Simulating Human-Robot Interaction in Microgravity Scenarios

Hugo Miguel Lopes Correia Ferro Nunes

Thesis to obtain the Master of Science Degree in
Aerospace Engineering

Supervisor(s): Prof. Rodrigo Martins de Matos Ventura
Prof. Duarte Nuno Jardim Nunes

Examination Committee
Chairperson: Prof. José Fernando Alves da Silva
Supervisor: Prof. Rodrigo Martins de Matos Ventura
Member of the Committee: Prof. Alfredo Manuel Dos Santos Ferreira Júnior

July 2019
Acknowledgments

I would like to express my gratitude to my supervisors, Rodrigo Ventura and Nuno Jardim Nunes, for their valuable advice, the knowledge shared and for reviewing the first draft of this thesis.

I wish to thank my family for their support: my parents Emídio Ferro and Ana Bela Correia; my uncle Paulo Ferro and my aunt Cláudia Ferro; my sisters Rafaela Ferro and Gisela Ferro.

Finally, I would also like to thank Bárbara Rosário, Bia Correia, Diana Carvalho, Diogo Rufino, Francisca Almeida, Gabriela Rocha, Inês Martins, Luísa Carvalho, Mariana Costa and Sebastião Costa for their friendship.
Resumo

O uso de robots de serviço no contexto da exploração espacial é essencial para aumentar a eficiência associada ao trabalho realizado pela tripulação da nave espacial. As responsabilidades envolvidas em gerir uma estação espacial são numerosas. Os astronautas podem beneficiar de um aumento em produtividade quando têm a possibilidade de delegar algumas tarefas repetitivas a robots. Estes robots necessitam de testes meticulosos para prevenir uma falha em circunstâncias nas quais os recursos necessários para resolver complicações inesperadas podem não estar disponíveis. Contudo, testar robots que irão operar em ambientes de microgravidade é caro e complexo. O programa de modelação Blender e o motor de jogos Unity são usados para criar um cenário a três dimensões que simula as condições da Estação Espacial Internacional. O robot Space CoBot é encarregado de transportar objetos entre dois astronautas ou entre um astronauta e um local de armazenamento adequado dentro da estação. O robot resolve vários problemas autonomamente, onde se inclui identificar o objeto que deve ser agarrado, abordá-lo e segurá-lo, encontrar um caminho entre dois locais que evite quaisquer obstáculos e determinar o ponto de entrega correto. É sensível nas suas interações com os humanos, ao observar os gestos que os astronautas fazem com os braços, as mãos e os dedos, para saber quando é que deve retirar ou entregar um objeto. Se os astronautas se movimentarem enquanto o robot está a realizar uma tarefa, o robot adapta o seu comportamento para refletir essa modificação.

Palavras-chave: microgravidade, interação humano-robot, Estação Espacial Internacional, Space CoBot, Blender, Unity
Abstract

The use of service robots in the context of space exploration is crucial to increase the work efficiency of the spacecraft crew. The responsibilities involved in managing a space station are numerous. The astronauts can benefit from an increase in productivity when offered the possibility to delegate some routine tasks to robots. These robots require meticulous testing to prevent a failure in circumstances in which the resources needed to solve unexpected complications may be unavailable. However, testing robots that will operate in microgravity scenarios is complex and expensive. The modeling software Blender and the game engine Unity are used to create a three-dimensional environment that simulates the conditions of the International Space Station. The Space CoBot robot is tasked with transporting an object between two astronauts or between an astronaut and an adequate storage place inside the station. The robot solves several problems autonomously, which include identifying the object that should be grabbed, approaching it and securing it, finding a path between two locations that avoids the existing obstacles and determining the correct delivery point. It behaves sensibly when interacting with humans, by observing the gestures the astronauts execute with their arms, hands and fingers, to know when it should remove or release an object. If the astronauts move while the robot is performing a task, the robot adapts its behavior to reflect that modification.

Keywords: microgravity, human-robot interaction, International Space Station, Space CoBot, Blender, Unity
Contents

Acknowledgments i
Resumo ii
Abstract iii
List of Tables vii
List of Figures ix

1 Introduction 1
  1.1 Topic overview ...................................................... 1
  1.2 Motivation .......................................................... 1
  1.3 Objectives .......................................................... 1
  1.4 Thesis outline ....................................................... 2
  1.5 State of the art ...................................................... 2

2 Blender 4
  2.1 Software overview .................................................. 4
  2.2 The astronaut ....................................................... 5
    2.2.1 The polygon budget ......................................... 5
    2.2.2 The mesh topology ........................................... 6
    2.2.3 The model ..................................................... 7
    2.2.4 The deforming bones of the armature ....................... 8
    2.2.5 The vertex weight groups of the armature ................. 10
    2.2.6 The control bones of the armature ......................... 11
    2.2.7 The animations ............................................... 15
  2.3 The robot .......................................................... 18
    2.3.1 The body ....................................................... 18
    2.3.2 The arm ........................................................ 18
    2.3.3 The docking station ........................................ 22

3 Unity 23
  3.1 Software overview ................................................ 23
    3.1.1 Interface ...................................................... 23
    3.1.2 Scenes, game objects and components ...................... 24
    3.1.3 Bounds ........................................................ 24
    3.1.4 Colliders ..................................................... 24
    3.1.5 Rigidbodies ................................................... 25
    3.1.6 Prefabs ....................................................... 27
    3.1.7 Layers ........................................................ 28

iv
3.1.8 Scripts .................................................. 28
3.1.9 Time and framerate management ........................................... 29
3.1.10 Physics functions .................................................. 30
3.1.11 Performance .................................................... 30
3.2 The simulation .................................................... 31
3.2.1 The interactions ............................................... 31
3.2.2 The managers .................................................. 37
3.3 The station .......................................................... 41
3.3.1 The game object ............................................... 41
3.3.2 The main manager ............................................. 43
3.4 The canvas .......................................................... 53
3.4.1 The game object ............................................... 53
3.4.2 The location manager ......................................... 53
3.4.3 The commands manager ..................................... 54
3.4.4 The constraints manager .................................... 55
3.5 The player .......................................................... 57
3.5.1 The game object ............................................... 57
3.5.2 The main manager ............................................. 59
3.5.3 The body manager ............................................. 64
3.5.4 The arm manager .............................................. 65
3.5.5 The eyes manager .............................................. 66
3.6 The robot .......................................................... 67
3.6.1 The game object ............................................... 67
3.6.2 The main manager ............................................. 68
3.6.3 The body manager ............................................. 73
3.6.4 The arm manager .............................................. 75
3.6.5 The propellers manager .................................... 77

4 Conclusion .......................................................... 78
  4.1 Achievements ...................................................... 78
  4.2 Future work .......................................................... 79

References .......................................................... 80

A Appendix .......................................................... 81
  A.1 The modeled mesh for the astronaut and for the robot .................. 81
  A.2 The game object hierarchy for the player and for the robot ................ 82
  A.3 Frustum culling and occlusion culling .................................... 82
  A.4 The robot navigation system ......................................... 85
  A.5 The robot cancellation system ..................................... 95
## List of Tables

2.1 Animations ......................................................... 18
3.1 Public variables in the Command class ..................................... 38
3.2 Public variables in the Instruction class .................................... 38
3.3 Public variables in the State class ........................................ 39
3.4 Public variables in the Orders class ....................................... 40
3.5 Public variables in the Share class ....................................... 41
3.6 Public variables in the Location class from the Station namespace ........ 43
3.7 Public variables in the Item class from the Station namespace .......... 49
3.8 Public variables in the Areas class from the Station namespace .......... 51
3.9 Public variables in the Point class from the Station namespace .......... 52
3.10 Public variables in the Line class from the Station namespace .......... 52
3.11 Public variables in the Paths class from the Station namespace .......... 52
3.12 Public variables in the Module class from the Station namespace ........ 52
3.13 Public variables in the Station class from the Station namespace ........ 53
3.14 Station Commands .................................................. 54
3.15 Player Commands .................................................... 54
3.16 Robot Commands .................................................... 54
3.17 Public variables in the SubCommand class from the Station namespace .... 55
3.18 Public variables in the Command class from the Station namespace ........ 55
3.19 Public variables in the Commands class from the Station namespace ........ 55
3.20 Examples of state updates for the translation constraints .................... 56
3.21 Examples of state updates for the rotation constraints ..................... 56
3.22 Public variables in the SubInstruction class from the PlayerMain namespace .... 60
3.23 Public variables in the Instruction class from the PlayerMain namespace .... 62
3.24 Public variables in the State class from the PlayerMain namespace ........ 62
3.25 Public variables in the Search class from the PlayerMain namespace .......... 63
3.26 Public variables in the Share class from the PlayerMain namespace ........ 64
3.27 Public variables in the SubInstruction class from the RobotMain namespace .... 70
3.28 Public variables in the Instruction class from the RobotMain namespace .... 72
3.29 Public variables in the State class from the RobotMain namespace .......... 73
3.30 Public variables in the Share class from the RobotMain namespace .......... 73
3.31 Public variables in the DockingStation class from the RobotMain namespace .... 73
3.32 Public variables in the Target class from the RobotBody namespace ........ 73
3.33 Public variables in the SubInstruction class from the RobotBody namespace .... 74
3.34 Public variables in the Instruction class from the RobotBody namespace ........ 74
3.35 Public variables in the Orders class from the RobotBody namespace ........ 74
3.36 Public variables in the State class from the RobotBody namespace ........ 75
3.37 Public variables in the Axes class from the RobotArm namespace ............................................. 75
3.38 Public variables in the Target class from the RobotArm namespace ............................................ 76
3.39 Public variables in the SubInstruction class from the RobotArm namespace ............................... 76
3.40 Public variables in the Instruction class from the RobotArm namespace ..................................... 76
3.41 Public variables in the Orders class from the RobotArm namespace ............................................ 76
3.42 Public variables in the State class from the RobotArm namespace .............................................. 76
3.43 Public variables in the Target class from the RobotPropellers namespace .................................. 77
3.44 Public variables in the SubInstruction class from the RobotPropellers namespace ...................... 77
3.45 Public variables in the Instruction class from the RobotPropellers namespace ............................. 77
3.46 Public variables in the Orders class from the RobotPropellers namespace .................................. 77
3.47 Public variables in the State class from the RobotPropellers namespace ................................... 77
A.1 Mesh complexity indicators ............................................................................................................ 82
A.2 Public variables in the Navigate class from the Station namespace ............................................. 88
A.3 Public variables in the SearchModulesNode class from the Station namespace ........................... 90
A.4 Public variables in the SearchModules class from the Station namespace ..................................... 90
A.5 Public variables in the SearchPathsNode class from the Station namespace ............................... 91
A.6 Public variables in the SearchPaths class from the Station namespace .......................................... 91
A.7 Public variables in the SearchGridNode class from the Station namespace ................................... 94
A.8 Public variables in the SearchGrid class from the Station namespace ............................................ 94
List of Figures

2.1 Blender interface with eight different areas ............................................. 4
2.2 Mesh geometry possibilities for the joints, with rotations of 0, 30, 60, 90 and 120 degrees . 6
2.3 Colleague model ....................................................................................... 7
2.4 Player model ............................................................................................. 8
2.5 First version of the astronaut armature ...................................................... 9
2.6 Second version of the colleague armature .................................................. 9
2.7 Second version of the player armature ....................................................... 9
2.8 Bending a finger with different sets of roll angles ....................................... 10
2.9 Vertex weight groups on a mesh animated by a two-bone armature .......... 10
2.10 Vertex weight groups for each bone of the colleague armature ................. 11
2.11 Vertex weight groups for each bone of the player armature .................... 11
2.12 Implement the leg IK chain: first step ..................................................... 13
2.13 Implement the leg IK chain: second step ................................................ 13
2.14 Implement the leg IK chain: third step .................................................... 14
2.15 Implement the leg IK chain: fourth step ................................................ 14
2.16 Implement the leg IK chain: fifth step .................................................... 14
2.17 Bone constraints for the player armature ................................................. 15
2.18 Animation for the colleague ...................................................................... 16
2.19 Animations for the player ........................................................................ 18
2.20 Robot’s arm ............................................................................................. 19
2.21 Mesh of the elements that form the arm of the robot (with different scales) (right and left names omitted for duplicated components) ............................................. 20
2.22 How to open the arm of the robot .......................................................... 20
2.23 How to grab an object with the arm of robot ........................................... 21
2.24 Valid and invalid dispositions for the arm of the robot (1) ....................... 21
2.25 Valid and invalid dispositions for the arm of the robot (2) ....................... 22
2.26 Docking station for the robot .................................................................... 22
3.1 Unity interface with seven different areas ................................................ 23
3.2 Robot grabs an object from the hand of the player ................................... 35
3.3 Robot delivers an object to the hand of the player .................................... 36
3.4 International Space Station layout: real model 4341 43 ............................. 41
3.5 International Space Station layout: virtual model .................................. 42
3.6 Item grabbed by the player ....................................................................... 43
3.7 Possible rotations for the item that is grabbed by the robot ....................... 44
3.8 Possible positions in the Z-axis for the item that is grabbed by the robot .... 44
3.9 Set of colliders used for the robot ............................................................ 45
3.10 Possible positions in the Y-axis for the item that is grabbed by the robot ... 46
3.11 Space correction for two different items .............................................. 46
3.12 Representation of the process used to find the arm radius (start position in green, end position in yellow, path in blue) .............................................. 47
3.13 Representation of the process used to find the approach point (start sphere in green, hit point in blue, solutions in orange, end sphere in yellow) ................. 48
3.14 Representation of the process used to find the new radius of the influence sphere (robot sphere in blue, item spheres in green) .............................................. 48
3.15 Areas for module Kibo 1 (zero in blue, one in red, two in green, three in yellow) .... 51
3.16 Paths created for module Node 2 (frontier points in red, interior points in yellow, lines in green) ................................................................. 52
3.17 Two possible coordinate systems for the player ...................................... 56
3.18 Player colliders (arm) ........................................................................... 58
3.19 Player colliders (all) ............................................................................. 59
3.20 Target used to search for an object to grab ........................................... 63
3.21 Moving the arm using animations ......................................................... 65
3.22 Moving the arm using scripting ............................................................. 66
3.23 Allowed rotation paths for the arm ....................................................... 66
3.24 Robot body colliders ........................................................................... 68
A.1 Astronaut mesh .................................................................................... 81
A.2 Robot mesh .......................................................................................... 81
A.3 Game object hierarchy for the robot and for the player ......................... 82
A.4 Representation of the frustum culling concept ...................................... 83
A.5 Frustum culling in the station ............................................................... 84
A.6 Representation of the occlusion culling concept .................................... 84
A.7 Occlusion culling in the station ............................................................. 85
A.8 Basic shapes that can be used to navigate ............................................. 87
A.9 Boxes used to translate ......................................................................... 87
A.10 Sphere used to rotate ........................................................................... 87
A.11 Testing the CanOccupyPoint function ............................................... 88
A.12 Testing the CanMoveToPoint function ............................................... 88
A.13 Testing the CanRotateInPoint function ............................................. 89
A.14 Uniform cost search algorithm ............................................................. 89
A.15 Two examples of a failed grid search .................................................. 92
A.16 First example of a successful grid search ............................................ 93
A.17 Second example of a successful grid search ........................................ 94
A.18 Example of a main manager cancellation ........................................... 98
A.19 Example of a body manager cancellation ........................................... 100
1 Introduction

1.1 Topic overview

The use of robots to perform repetitive tasks is increasingly common. Humans and robots can share the responsibilities involved in completing a job, allowing it to be concluded faster and with greater efficiency. The humans working along with robots are not burdened with the routine aspects of an action and have more time to dedicate themselves to activities that require intuitive decisions. The robots should carry out their assignments autonomously, with as little input from the humans as possible.

This is especially important in the context of space exploration. The astronauts operating a space station can benefit from the help of assistance robots in a variety of situations. Robots can be used not only to perform tasks deemed too dangerous for a human, but also to free the astronauts from the monotonous chores necessary to keep the onboard operations functioning correctly. The spacecraft crew is always limited in number and constantly busy with all the duties that living in space entails, but the use of robots can make up for some deficiencies in elements and time, with the overall goal of improving the productivity of the existing crew members.

1.2 Motivation

Any robot requires intensive testing before being considered apt to perform the job it was designed to do. The testing phase is even more crucial for a robot that will be used in a space station. The vehicles that travel between the station and the Earth, carrying people and supplies for each expedition, are infrequent and costly. If a robot that is already operating in space malfunctions, the spacecraft may not have the resources necessary to repair it. The astronauts would have to wait weeks or months for the necessary assets to be sent from the ground, or even more time if it were not possible to fix the robot on board and it had to go back to Earth to be redesigned. The purpose of a service robot is defeated if the crew spends more time worrying about it than being aided by it. For these reasons, any robot should only be sent to the space station when there is a high degree of certainty that it will work as it is supposed to.

The main complication of complying with this requirement arises from the particular nature of the environment the robot will operate in. The robot is designed and tested on Earth, in significantly different conditions from the ones in which it will be deployed. In order for the robot to be adequately tested, the gravity conditions experienced in space have to be reproduced on Earth, but the simulation of microgravity scenarios is difficult and expensive.

1.3 Objectives

The goal of this thesis is to develop a virtual reality simulation of a microgravity environment inside a space station that can be used to test the robot, instead of trying to replicate that scenario in real life. This involves the creation of a virtual model of the International Space Station (ISS), which was the spacecraft chosen to implement the simulation. It also requires the creation of a model of an astronaut
that the robot can interact with. The robot that will be used, the Space CoBot, was already designed by Pedro Roque and Rodrigo Ventura for a previous project [1].

The model of the station should strive to resemble its real-life counterpart, with believable and accurate graphics, that provide an immersive and reasonably realistic environment to test the robot.

The model of the astronaut needs to be able to roam the station, moving his body as a human would when experiencing microgravity circumstances. It should also have some liberty to use his arms, in order to grab objects that are lying around in the station. The astronaut will be replicated several times, allowing the user of the simulation to take the place of one of the members of that crew and control his actions.

The purpose of the robot is to transport small or medium-sized objects inside the station. The robot should be able to grab an object, move around the station with that object attached, and finally drop it in the appropriate place. In order to achieve the final objective, the robot needs to solve several problems. These include identifying the object that should be grabbed, deciding the best way to approach it and secure it, finding the shortest path from the initial point to the final point, without hitting the humans, the walls or the other objects inside the station, and choosing where to deposit the object. After a given object is delivered, the robot should make itself available to transport another object that may be requested.

The objects can be carried between the place where they are stored in the station and one of the astronauts, in any combination. For example, the astronaut might ask the robot to get an object from the station and bring it to him, or to pick up an object that the astronaut is holding and store it back in the adequate location. The astronaut might also ask the robot to deliver an object to another astronaut, or to fetch an object that is in the possession of a determined colleague.

When interacting with the astronauts, the robot should carefully observe their gestures. This entails waiting for the astronaut to open his fingers before removing an object from his hand, and also waiting for him to close his fingers around the object before releasing it to his care. If the astronaut moves while the robot is trying to catch or deliver an object, it should recognize that change and react accordingly.

1.4 Thesis outline

This thesis can be divided into two main challenges. The first is to create the 3D assets that will be employed in the simulation, using the modeling software Blender. They include the modules of the station, some of the objects that can be found inside the station, the astronauts and the robot. The second is to import these assets into the game engine Unity and program them to act in the desired way. All three major elements of the simulation - the station, the astronauts and the robot - require programming to achieve the intended functionalities. It is also necessary to create an interface for the user to interact with the simulation.

1.5 State of the art

There are several free-flyer robots that were designed to operate in the ISS.
The SPHERES system started being developed in 1999, by master’s students at Massachusetts Institute of Technology (MIT). It has been in the ISS since 2006; once there, the National Aeronautics and Space Administration (NASA) took ownership of the program. The acronym SPHERES stands for Synchronized Position Hold, Engage, Reorient, Experimental Satellite. The system is used to test a diverse range of hardware and software and to record data. It includes three identical robots, each with a diameter of 20 cm. There are several methods available to test the robots with gravity. One possibility is to use flat floor labs: the robot rides on top of a platform with low air friction, allowing for translation in the X and Y axes, as well as rotation about the Z axis. This is called a 3 degree of freedom platform (DoF). Another possibility is to use a crane structure - a 6 DoF platform - that allows the robot to float as if it were in space. It is also possible to test the robot in a sinusoidal 0 G aircraft ride: the aircraft flies up and down and it offers up to 30 seconds of reduced gravity when in a dive [2].

The Astrobee system, designed by NASA, was installed in the space station in February 2019. It was developed with the knowledge acquired during the years the SPHERES system has been aboard the station. Once this new system is fully commissioned, it will take over as the main robotic test facility in the ISS. The Astrobee system is used to take inventory, document experiments and transport objects throughout the station. The robots can work autonomously or under the remote control of astronauts in the station or researchers on the ground. The system includes a docking station and three collaborative robots, named Honey, Queen and Bumble. The robots are shaped like cubes, with a length of 30 cm. Each robot includes a perching arm that can grab the station handrails, in order to conserve energy or assist astronauts. The Astrobee system is tested using Gazebo, a robotics simulator that can be used to assess algorithms, design robots and train artificial intelligence systems using realistic scenarios. It includes a physics engine and offers high-quality graphics. The simulation built for the Astrobee system allows researchers to test software before implementing it on a real robot. It simulates the robot propulsion system, the perching arm, the cameras, the inertial sensor and the environment, which is provided by a CAD (Computer Aided Design) model of the ISS. Each simulator plug-in mimics the real hardware by offering the same ROS interfaces: topics, services and actions [3] [4] [5].

The JEM Internal Ball Camera robot, informally known as Int-Ball, was designed by the Japan Aerospace Exploration Agency (JAXA) and delivered to the ISS in June 2017. This robot is able to move around the station to take pictures and record videos. The goal is to reduce to zero the time the crew spends taking photographs, which amounts to about 10% of their working hours without any robotic assistance. The Int-Ball robot is shaped like a sphere with a diameter of 15 cm [6].

The CIMON robot was developed by Airbus and IBM for the German Aerospace Center (DLR) and sent to the ISS in June 2018. The acronym CIMON stands for Crew Interactive Mobile Companion. The robot was programmed to answer verbal questions and to give instructions to astronauts in order to help them carry out experiments. This robot is shaped like a sphere with a diameter of 32 cm.

---

2 Blender

2.1 Software overview

Blender is an open source software developed by Blender Foundation. It offers a wide range of features, that include modeling, sculpting, rigging, animation, visual effects, simulation, node based material creation, texturing, UV mapping, rendering, lighting, video game creation, video editing and scripting. It is ideal to create realistic looking objects with a low polygon count, fit to be used later in a game engine.  

The interface is divided into a number of rectangular areas, each one containing a particular editor. The ones most commonly used in this project, shown in Figure 2.1, were:

- 3D View (outlined in blue): used to visualize the mesh, for modeling, sculpting, animation, etc.
- Graph Editor (outlined in red): used to modify the animation properties through F-Curves.
- Timeline Editor (outlined in yellow): used to play or pause the animation, skip several frames or choose a specific one.
- UV/Image Editor (outlined in green): used to view and modify 2D assets like UV maps.
- Node Editor (outlined in purple): used to create node-based materials.
- Properties Editor (outlined in orange): used to edit data and properties for the active scene and the active object.
- Outliner (outlined in pink): used to view an organized list of all objects in the file.

![Figure 2.1: Blender interface with eight different areas](https://example.com/blender-interface.png)
2.2 The astronaut

The astronaut character represents the humans living in the space station. It is the most complex asset that needs to be created, requiring not only modeling but also animations. It is given as an example of the workflow that was at least partially adopted when creating the remaining elements of the scene.

2.2.1 The polygon budget

The goal when creating a model is to make it appear as close to reality as possible without compromising the game performance. This requires control of the vertex or polygon counts: convincing looking models are more complex and have higher values, but lower ones are essential if the game is to run smoothly. The vertex count is the total number of vertices in the mesh. The polygon count usually means the triangle count, not the face count. The face count is the sum of all polygons, regardless of their number of sides. The triangle count is the number of faces after the entire model is triangulated: one for a three-sided face, two for a four-sided face, three for a five-sided face, etc. The vertex and polygon counts follow the same tendency, but have some degree of independence, due to breaks in the mesh.

The two most important questions are how many vertices can be used in the complete scene and how should they be distributed between the elements of that scene.

The total available amount depends on the power of the game engine and the console. The Unity manual advises to aim for no more than 100 000 vertices on mobile; on PC, even if it handles well several million vertices, it is still preferable to keep the value as low as possible. The scene will be complex, with several station modules and the components that fill them; the environment will burden the performance and shrink the margin available for the character.

The distribution of vertices between the models relies on four major aspects: importance, distance, interactivity and quantity [7]. The astronaut grades high on the first three. As the main character, it will always be close to the camera and the user will constantly interact with it, which means that it should receive a generous share of the total budget. However, since there will be more than one astronaut in the scene, that amount cannot be as high as it could be if there was just one character.

Even with these guidelines, it is difficult to pinpoint an adequate number. To have a better idea of the common practice, the best way is to research the websites of video games companies and the portfolios of their developers, focusing on recent PC games similar in design to the one intended for this project. This research found that characters conceived by professionals usually add up to anywhere between 15000 and 60000 polygons.  

---

2.2.2 The mesh topology

The topology is defined by the arrangement of vertices, edges and faces that belong to a mesh.

The main concern when modeling a game asset is to use quads (faces with four sides) that are as close to squares as possible; ngons (faces with more than four sides) are entirely ruled out and tris (faces with three sides) should be scarce and exclusive to meshes that will not be animated [7]. Several tools only work well with quads. When adding a new edge loop to a mesh, it gets blocked if it finds a tris or a ngon in the way. Shading options can also uncover problems, when switching from flat to smooth, making the mesh appear pinched, with odd light reflections. Animation programs are better at deforming quads and cannot properly distort other polygons. Sculpting details usually requires a subdivision modifier, which also displays its best results when applied to quads with balanced proportions.

Some additional concerns pertain specifically to the areas of the mesh that will deform with the armature. The geometry of these joints affects the quality of the animation. Before starting to model, several possibilities were tested, to decide on the type of topology to use on all points where the mesh is going to bend. A cylinder was used for these tests, with the joint positioned in the middle. Its shape resembles most of the parts of the astronaut that will be animated: torso, legs, arms and fingers. The topology chosen here will be replicated around all articulations: waist, ankles, knees, shoulders, elbows, wrists, knuckles.

![Figure 2.2: Mesh geometry possibilities for the joints, with rotations of 0, 30, 60, 90 and 120 degrees](image)

Different aspects should be considered when evaluating the results displayed in Figure 2.2.

**Enough geometry** The mesh needs to have at least some vertices around the joint. Figure 2.2a is the most obvious example of this problem, with 2.2b as a close second. Without preexisting vertices, the mesh does not have enough information to guess how it will look like when deformed.

**Hidden side appearance** The compressed side of the mesh endures different rotations before collapsing. Figure 2.2e retains the faces at 60 degrees, while Figure 2.2i smashes into itself.

**Exposed side appearance** The stretched side of the mesh should have a rounded look. Figures 2.2c, 2.2e and 2.2g do not accomplish this, because they only have two loop cuts, which creates a flat surface when deforming.

**Volume conservation** Both halves lose their volume when there is only one loop cut in the back,
as seen in Figures 2.2g and 2.2h.

**Unnecessary geometry** The front side requires more geometry than the back, which means some loop cuts can be merged in the transition to save vertices, as shown in Figures 2.2f, 2.2g and 2.2h. However, while some joints have these sides clearly defined, others do not. A knee only bends one way, but the same cannot be said about the neck.

**Mesh flow** Merging loop cuts disturbs the topology, creating triangles and making it harder to neatly edit the mesh. Figure 2.2d shows the option that suits all kinds of articulations and keeps the faces even. This was the basic topology upon which the astronaut’s joints were built, with additional loops for the ones that required more detail.

### 2.2.3 The model

When modeling the astronaut, the first step was to block out its major elements with basic solids, as shown in Figure 2.3a. After getting right the generic shape and proportions, all the necessary details were added, until reaching the final stage, displayed in Figure 2.3b. The astronaut was modeled in a T pose: standing up, with the legs slightly apart and the arms spread away from the torso. This makes it easier to reach all places in the mesh [7].

![Figure 2.3: Colleague model](image_url)

The astronaut will be used as two different kinds of character, the colleague and the player. The user will see the station from the eyes of the player, so he will be able to see the colleagues, which are the non-playable characters controlled by the game engine to follow a programmed behavior. But he will not be able to see himself, or at least most of himself during most of the time. He will still be able to see his own arm, when he raises it to grab an object; other than that, it can be reasonably assumed that he never sees any other part of his body. Even when the player looks down, to where his legs and torso would normally be, their absence can be justified on the specific nature of this environment. A person
walking on Earth will certainly find their legs when looking down, but an astronaut in the space station can be floating on his stomach and with his legs pointing backwards. This circumstance allows us to eliminate everything but the arm for the player character, saving up considerably on vertex count and therefore performance. When choosing which arm to keep, it was taken into account that the majority of the population is right-handed and tends to grab things with that dominant arm, so the left arm was the one deleted.

![Figure 2.4: Player model](image)

### 2.2.4 The deforming bones of the armature

Meshes are animated using armatures, which are composed of one or more bones. Each bone has three elements: the start joint or head, the body, and the end joint or tail. The bones in the armature can be linked in parent-child relationships, forming one or more chains. The position and rotation of two connected bones are not independent: if one of them changes, so will the other, to keep the integrity of the chain. The first bone in a chain is called the root bone and the last one is the tip bone.  

Bones can be broadly classified into deforming bones and control bones. Deforming bones are the ones that interact directly with the mesh. Control bones only move the mesh indirectly, by moving first the deforming bones. They make the animating process more intuitive and save time that would otherwise be spent on some repetitive adjustments.

The astronaut is a humanoid character, therefore its armature should mirror a human skeleton. Bones need to be laid out where real ones would be, with the body of the bone positioned along the rigid part of the limb and the joints coinciding with the articulations. All bones should belong to at least one chain, and all chains should eventually converge to one single root bone [7].

Figure 2.5 represents the first version of the armature, created to meet the basic motion needs of a person, but not necessarily one that is dressed in a space suit. This armature can be simplified after considering the restrictions imposed by this specific garment. In order to know which bones can be deleted, one has to think which movements of the person wearing the suit are reflected on the outside. For example, if the person raises an arm, the arm of the suit is also raised. However, if the person looks around, the helmet stays in the same place. This means that the Head Bone and the Neck Bone are useless and can be eliminated. For simplicity reasons, all Finger Bones were also removed from the main armature, while the Fore Arm Bone was removed from the secondary armature. This means that a colleague astronaut will not able to use his fingers and the player astronaut will not be allowed to bend his arm around the elbow.

---


Figure 2.5: First version of the astronaut armature

Figure 2.6: Second version of the colleague armature

Figure 2.7: Second version of the player armature
After placing the joints of the bones, their roll angle - the rotation along the longitudinal axis - needs to be adjusted. It is important to keep the roll angles consistent because of the rotation constraints that will be added later \cite{7}. For example, if the roll angles are all the same for the three bones of a finger, all phalanges will bend together towards the palm of the hand when copying the first one, as shown in Figure 2.8a. However, if the roll angles are different, each one will bend in its own direction, as displayed in Figure 2.8b.

![Figure 2.8: Bending a finger with different sets of roll angles](image)

\subsection*{2.2.5 The vertex weight groups of the armature}

Creating an armature in the same location as the mesh is not enough to animate it. In order for the mesh to follow the armature, it needs to know how each bone affects each vertex. This influence is conveyed by a percentage, with zero meaning that the vertex stays in the exact same place when the bone is moved or rotated and a hundred that the vertex acquires the same position and rotation of the bone, keeping its offset to it null. Intermediate values make the vertex partially follow the bone, interpolating between the two extreme cases. The animator has to select a bone, assign a percentage to each vertex in the mesh in relation to that bone, and then repeat the process for all remaining bones in the armature. This means that each vertex can be influenced by more than one bone at the same time \cite{7}.

Figure 2.9 exemplifies the process in a simple case of a cylinder animated by a two-bone armature. For each step, on the left is represented the top bone weight group, in the middle the bottom bone weight group and on the right the mesh reaction when the top bone is rotated. Red is for maximum influence and blue for minimum. Three major conclusions can be drawn from this example. First, two bones that share a joint usually have weight groups that are approximately symmetric in relation to the central point. Second, the influence of a bone on a given vertex decreases with the distance between that vertex and the central part of the bone. Third, it is preferable to assign a gradient of values instead of having abrupt changes, in order to increase the smoothness of the deformation.

![Figure 2.9: Vertex weight groups on a mesh animated by a two-bone armature](image)

Figures 2.10 and 2.11 show the weight groups created for the astronaut armatures.

![Figures 2.10](image)

(a) Root  (b) Spine 01  (c) Spine 02  (d) Upper Arm

(e) Fore Arm  (i) Foot

(f) Hand  (g) Thigh  (h) Shin

![Figure 2.11](image)

(a) Arm  (b) Thumb 01  (c) Thumb 02

(d) Index 01  (e) Index 02  (f) Index 03  (g) Middle 01

(h) Middle 02  (i) Middle 03  (j) Ring 01

(k) Ring 02  (l) Ring 03  (m) Pinky 01

(n) Pinky 02  (o) Pinky 03

Figure 2.10: Vertex weight groups for each bone of the colleague armature

Figure 2.11: Vertex weight groups for each bone of the player armature

Only the deforming bones require vertex weight groups, since they are the ones responsible for directly moving the vertices. Control bones, on the other hand, do not need weight information [7].

2.2.6 The control bones of the armature

Two different techniques were used to pose the armature: forward kinematics and inverse kinematics. Forward kinematics makes the conversion between the joint space and the work space: it receives
as input the angles of the joints and returns as output the position and the rotation of the last element in chain. This is the default option and does not demand any control bones. With this method, the stream of influence follows the traditional path, from the parent to the child. Posing the tip bone of a chain using forward kinematics requires sequential posing of all the previous bones, starting at the first bone that will have any influence on the extremity. For example, to position the hand in front of the body, as if to grab something, first the upper arm needs to be rotated around the shoulder, then the forearm around the elbow and finally the hand around the wrist. The child bones inherit their parent’s position and rotation and are themselves positioned and rotated from there. It may be difficult to predict precisely what the rotation and position of the shoulder and elbow must be in order for the hand to end up where it is desired. Every time the parent bone is slightly adjusted, all its children need to be adjusted as well. Small and time-consuming changes have to be made up and down the chain, until the final result is achieved. The workload will depend on the length of the chain and on the position of the target bone along it. Forward kinematics should be used when the final position of the hand or the foot is not important and can be decided by how the animator transforms the root bones of the arm or the leg [7].

Inverse kinematics makes the conversion between the work space and the joint space: it receives as input the position and the rotation of the last element in chain and returns as output the angles of the joints. This process requires less effort and is more intuitive. With this method, influence is reversed and it is the child bone that affects the parent. Stretching the hand forward requires the user to do just that, since the program will retrace the adequate position and rotation of the parents up the hierarchy. This technique is the one that better reflects how a person thinks when grabbing something. Inverse kinematics should be used when the position of the hand or foot is important and the arm or leg are only required to compensate in the most natural way to achieve that final result [7].

To set up inverse kinematics, three main settings have to be decided: which bone will host the constraint, which bone will be the target and what will be the length of the inverse kinematics chain (IK chain). To invert the default parent-child behavior, the constraint needs to be placed on the parent bone and the target has to be its child, so that the parent will now follow the child. The length of the chain determines how many bones, counting up from the one with the constraint, will be affected when the target bone moves.

The first IK chain built was the one working the leg, and the first step was to decide which bone the leg will follow. The choice falls on one of the last bones in the chain, but not necessarily the very last. In Figure 2.12a the leg follows the toe position. The toe is first translated up and forward, which makes the knee, ankle and the ball of the foot bend in an impossible way. Then the toe needs to be rotated just to revert the boot to its default look. This choice makes the leg joints behave unnaturally and complicates the control of the boot roll, since a null rotation no longer means that the boot is straight. In Figure 2.12b the leg follows the foot position instead. With this choice the knee now functions as it should and the boot is not automatically deformed. Without any deliberate action from the animator, the sole of

---

the boot is kept parallel to the ground. The ankle and the ball of the foot will only rotate if the user decides to do so. The best option is therefore to make the leg follow the foot.

![Figure 2.12: Implement the leg IK chain: first step](image)

(a) Toe Bone          (b) Foot Bone

The previous results would implicate that the Foot Bone should be the target and its parent, the Shin Bone, should receive the IK bone constraint. However, if this option is implemented, nothing happens when trying to move the foot. The reason for this is that the bones are now looped in parenting relations: the foot is the child of the shin because of the original connection and shin is acting as the child of the foot because of the inverse kinematics, so they both look at each other to know where to go. Moving the foot would move the shin, which then would move the foot, and then again moving the shin, in an infinite positive feeding loop. To prevent this, the program blocks the foot in place. One solution would be to disconnect the foot from the shin, cutting the original parent-child relation, as shown in Figure 2.13a. The problem is that now the foot is not constrained by distance and can break away from the leg. Another solution is to make a duplicate of the foot, the Foot IK Bone, and delete the parent from that duplicate instead. This duplicate is the bone that should be dragged to move the leg, which is now possible. If the duplicate is moved too far away, the leg stops following it after reaching its maximum extent, as shown in Figure 2.13b.

![Figure 2.13: Implement the leg IK chain: second step](image)

(a) Without the Foot IK Bone          (b) With the Foot IK Bone

However, as shown in Figure 2.14a, this duplicate lost the control the original bone had over the rotation of the foot. Rotating the duplicate mismatches the bones and does nothing in the actual foot; it is necessary to go to the original bone in order to roll the boot by the ankle. To solve this, a second bone constraint, this time a Copy Rotation, needs to be added to the Foot Bone, targeting the Foot IK Bone. Now the foot responds when the duplicate bone is rotated, as shown in Figure 2.14b. The Foot Bone is paired with the Foot IK Bone and becomes a sort of invisible bone, that is never directly manipulated but is still doing important work in the background.
The next step is to decide the length of the chain. Figure 2.15a shows what happens when all bones in the chain are counted in: the shin, the thigh and the root. Moving the foot up also moves the root, which in turn moves the whole body of the astronaut. Figure 2.15b shows the effect of the same movement when the IK chain ends in the thigh, and in Figure 2.15c the chain ends in the shin. The desired result is the middle one, where the whole leg and just the leg moves with the foot.

The final step is to create a pole target bone for the IK chain, the Knee Control Bone, which decides the orientation of the knee. For example, when pulling the leg up, the knee will point forward, as shown in Figure 2.16a, but the animator might want it pointing sideways, as shown in Figure 2.16b. Though this could be accomplished by rotating the Shin Bone, it may be difficult to figure out the rotation that will achieve the desired result. With the Knee Control, the animator only has to translate this bone and the leg will rely on its position to compute the necessary rotations.
In a process similar to the one used for the leg, a new IK chain was built for the arm. The Hand Bone was chosen as the target and the Fore Arm Bone as the one holding the IK constraint, with a length of two bones, to include the Upper Arm Bone. The Elbow Control Bone was created to serve as the pole target. The colleague armature has a total of four IK chains, one for each arm and leg.

No control bones were added in the player armature, only bone constraints on the deforming ones. Without any constraints, closing each finger would require adjustments to each phalanx. After a Copy Rotation constraint was added to all second and third phalanges, targeting the previous one, only the first phalanx of each finger has to be rotated to close the hand.  

![a) Without bone constraints](image1.png) ![b) With bone constraints](image2.png)

Figure 2.17: Bone constraints for the player armature

### 2.2.7 The animations

An animation has a given number of frames, and each frame has associated a position and rotation for each bone. This does not mean that the animator has to fill in each frame. For example, to raise an arm, it is not necessary to increment its rotation by tiny amounts, from the start position until the final one is reached. The animator only has to choose how the arm looks like when it is down, and how it looks like when it is up. The first pose is saved on the first keyframe and second pose on the last. For all the unassigned frames between them, the program will interpolate to know how the bones should be located. The decision of how many empty frames separate each keyframe depends on whether the movement should be abrupt or smooth. As the distance between keyframes increases, the longer the bones will have to reach their final pose and the gesture becomes more relaxed. If the animation is played in a loop, the initial and final poses have to be the same to avoid discontinuities [7].  

When a model has more than one animation, they have to be ordered sequentially in the same timeline, to be divided later into clips inside the game engine.  

Only one animation was created for the colleague astronaut. This idle animation will always be playing in a loop. To decide what the colleague should do in this animation, it is necessary to take into account the environment and which interactions the colleague will be required to have with the robot.

The colleague does not experience the effects of gravity, so its pose should give an appearance of weightlessness. His legs do not have to be straight, with the feet flat and touching the ground. It is much more realistic to bend them, with each knee turned slightly outwards, and make the sole of the boots face backwards. Leaning the torso adds to the impression that the colleague is floating in space. One of

---


the arms can be relaxed, by the side of the body, with the hand pointing down, but the other one needs to be stretched forward, so that the robot can place objects on that hand. The arm has to be extended enough for the robot to have space to maneuver without colliding with the torso. Since the colleague does not have finger bones, his hand is always open, with the palm facing upwards, ready to receive the objects the robot wants to deliver. After the robot leaves the object on the hand of the colleague, he will not place them somewhere else, or lower the arm, so the object will still be in the same position if the robot wants to retrieve it later.

This pose could now be saved into a single frame animation and the colleague would be ready to fulfill its function in the scene. However, it does not look realistic to have a character that is always perfectly still, so a secondary pose was built, based on the first one. The feet were pulled upwards and arms were rotated out, away from the body. The whole torso was pushed forward, to suggest a balancing motion. To emphasize this idea, the entire model was also slightly translated.

The first pose was saved on the first and last keyframes of the animation, while the second one was saved in the middle. The animation has to be long enough to avoid lurching, robotic and obviously repeating movements, but not so long that these become imperceptible. The duration was settled on four seconds.

To decide which animations are required for the player, first it must be established how the arm will behave. The user should be able to lower the arm when he is not using it, hiding it away from its field of vision, so that he can move more easily around the station and approach the walls. But he also needs to raise the arm when grabbing an object or interacting with the robot. Aside from bending around the elbow, the arm should have all the mobility that a real arm has, which means it should rotate around all three axis. The goal is to offer the user the possibility to move it up and down, to the right and to the left, and rotate it on itself so that the palm is facing all possible directions. Finally, the fingers also need to open and close when grabbing objects. Just as it happens for the whole arm, the bending of the fingers can be done in numerous different ways, depending on the size and format of the object.

Considering the infinite amount of possibilities for the arm and the fingers, it was decided that some of the animations would be created in Blender and others in Unity. Though it is possible to animate bones directly in the game engine, the complexity of that process increases with the number of bones. Furthermore, it is only justifiable to create the animations by scripting if all the fine control that will then be available can be put to use. The rotation of the arm will be almost exclusively controlled by the player during the simulation, but the fingers will close automatically when there is an object that needs to be grabbed. For the purpose of this project, it would be senseless to ask the user exactly how he wants to translate and rotate each phalanx of each finger. After implementing the same precise control that
the player has over the arm on the fingers, the way they close would also have to be programmed. This would require evaluating the volume and shape of each object to guess how a person would grab it, a task that was deemed outside of the scope of this project. For this reason, movements that only require the Arm Bone will be handled later in Unity, while movements that only require the Finger Bones, or both Arm Bone and the Finger Bones, will be animated now in Blender. This means that the whole arm will retain a wide range of motion, but the behavior of the fingers has to be simplified.

The first animation created was the gesture the player makes when receiving an object from the robot. When the player is preparing itself to use the arm, but the robot is still at a considerable distance, he may raise the arm in front of his body, but not open the palm of the hand right way. In the first pose of this animation, the arm is raised horizontally, with the palm facing down and the fingers slightly closed in a relaxed position. In the last pose, the palm is facing up and the fingers are completely stretched. Both the Arm Bone and the Finger Bones are animated for this action. The initial and final stages can be seen in Figures 2.19a and 2.19b, respectively.

The remaining animations created concern the movements done after the robot places the object on the hand and the player secures it. There will be several objects available, which means that the fingers will not always close in the same way. In order to provide a reasonable amount of flexibility without increasing too much the number of animations, it was decided that all five fingers would rotate simultaneously and with the same angle. This closing angle was incremented by ten degrees for each animation, from zero to ninety degrees. The player has the option of not closing the fingers at all, when the object is very large, or closing them completely until the fingertips touch the palm, for objects with small diameters. For objects with intermediate sizes, the option that best fits them should be chosen from the available ones. Though the first animation has no visible effect, since the fingers start open and remain open, it was created to keep things consistent when programming the arm inside the game engine. In all these actions, only the Finger Bones were animated. The initial stage for these animations is shown in Figure 2.19b, and the final ones in Figures 2.19c through 2.19l.

The player also has to revert the arm to its relaxed position and to open the fingers when releasing an object. To avoid doubling the number of animations created in Blender, the ones that already exist will be reversed by scripting later in Unity.

All the animations created for the player are actions, which means that they will not be played in a loop and the final pose can be different from the initial one.
Table 2.1: Animations

<table>
<thead>
<tr>
<th>Model</th>
<th>Name</th>
<th>Start frame</th>
<th>End frame</th>
<th>Frame rate [fps]</th>
<th>Duration [s]</th>
<th>Played in a loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colleague</td>
<td>Idle</td>
<td>1</td>
<td>65</td>
<td>24</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>Player</td>
<td>Hand</td>
<td>1</td>
<td>25</td>
<td>24</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>Player</td>
<td>Fingers 00</td>
<td>26</td>
<td>50</td>
<td>24</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>Player</td>
<td>Fingers 10</td>
<td>51</td>
<td>75</td>
<td>24</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>Player</td>
<td>Fingers 20</td>
<td>76</td>
<td>100</td>
<td>24</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>Player</td>
<td>Fingers 30</td>
<td>101</td>
<td>125</td>
<td>24</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>Player</td>
<td>Fingers 40</td>
<td>126</td>
<td>150</td>
<td>24</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>Player</td>
<td>Fingers 50</td>
<td>151</td>
<td>175</td>
<td>24</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>Player</td>
<td>Fingers 60</td>
<td>176</td>
<td>200</td>
<td>24</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>Player</td>
<td>Fingers 70</td>
<td>201</td>
<td>225</td>
<td>24</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>Player</td>
<td>Fingers 80</td>
<td>226</td>
<td>250</td>
<td>24</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>Player</td>
<td>Fingers 90</td>
<td>251</td>
<td>275</td>
<td>24</td>
<td>1</td>
<td>No</td>
</tr>
</tbody>
</table>

2.3 The robot

The Space CoBot is a small aerial robot for inhabited microgravity environments, such as orbiting space stations. Its design is modular, comprising an hexrotor based propulsion system, and a stack of modules including batteries, cameras for navigation, a screen for telepresence, a robotic arm, space for extension modules, and a pair of docking ports, that can be used for docking and for mechanically attaching two Space CoBots together. The kinematics is holonomic, and thus the translational and the rotational components can be fully decoupled [1].

2.3.1 The body

The body of the robot was already modeled in SolidWorks [1], but there were some problems with the original mesh concerning its use in Unity. It had too many vertices, when it is particularly important that the robot, which will be constantly moved around the scene, to have as few as possible. The topology of the mesh was also inadequate for some of the tasks that have to be completed before exporting the model. It was triangulated and had discontinuities, giving it a jagged look when adding modifiers. It also makes it harder to mark seams and unwrap the mesh for texturing. For these reasons, the robot was remodeled, with the same dimensions and shape, but with less vertices and a quad based topology. The results can be seen in Appendix A.1.

2.3.2 The arm

The new arm of the robot was designed from scratch, because the original one did not observe all of the requirements established. The arm has to able to recoil, in a way that it can fit inside the opening on the bottom of the robot. The robot docks at his station with its lower part, so the arm cannot be
sticking out. There is not a single defined type of object that will be grabbed; these will be varied in size and shape. This means that the arm should be able to open as much as possible, to increase the pool of objects that can be picked up. The part of the arm that makes contact with the object should be somewhat generic, meaning that it must not be adapted to a specific object shape. An arm based on a forceps or a claw might not be adequate to secure all objects.

The arm has eleven components: Right Support, Left Support, Big Main, Big Right, Big Left, Small Right Bottom, Small Right Middle, Small Right Top, Small Left Bottom, Small Left Middle and Small Left Top. The Support Components are fixed to the body of the robot and never move or rotate. The Big Main Component can rotate 90 degrees around the supports, to close or open the arm. The Big Right and Big Left Components can slide along the Big Main, in and out, to increase the span of the arm and allow it to grab bigger objects. The Small Bottom Components slide along the Big Right and Big Left Components, dragging the Middle and Top Components with them. The Middle Components can rotate 90 degrees around their articulations with the Bottom Components, while the Top Components do the same around the articulations with the Middle Components, so that the arm fits the opening in the robot body when it is time to close it.
To open the arm, the first step is to rotate the Big Main Component, from Figure 2.22a to Figure 2.22b. The second step is to translate both the Small Right Bottom and the Small Left Bottom Components, until they are at their maximum span, touching the end points of the Big Right and Big Left Components, as displayed in Figure 2.22c. When the arm is closed, they have to be at the position shown in Figure 2.22b, slightly pulled to the center, so that they do not hit the rim of the opening in the body of the robot. However, in order to open the small arms, they have to be pulled back, otherwise the small right arm would hit the left one when rotating. They would also hit each other if they were opened at the same time, so first the Small Right Middle Component is rotated, as displayed in Figure 2.22d, and only then the Small Left Middle Component can do the same, as shown in Figure 2.22e. The final step is to rotate both Small Top Components, so that the arm is fully distended and ready to be used. These two actions are performed at the same time, because the two opposing components are distanced enough to avoid a collision. The arm can be seen completely open in Figure 2.22f.

To close the arm, the Big Right, Big Left, Small Right Bottom and Small Left Bottom Components have to slide to the positions displayed in Figure 2.22f. After this, all the steps mentioned before are repeated backwards.

Figure 2.21: Mesh of the elements that form the arm of the robot (with different scales) (right and left names omitted for duplicated components)

Figure 2.22: How to open the arm of the robot
To grab an object, the Big Right, Big Left, Small Right Bottom and Small Left Bottom Components are translated to the adequate positions, depending on the dimensions of the object. The arm can be at 0 cm, when the pads of the small arms are touching each other, or at 17.5 cm, when all four components are fully distended, each side at a radius of 8.75 cm from the center.

(a) Small at minimum, Big at minimum
(b) Small at midpoint, Big at minimum
(c) Small at maximum, Big at minimum
(d) Small at maximum, Big at midpoint
(e) Small at maximum, Big at maximum

Figure 2.23: How to grab an object with the arm of robot

One side can also be at a different radius from the other, when the object is not symmetric or when it is not centered with the robot, as displayed in Figure 2.24a. However, the situation represented in Figure 2.24b is not allowed. When the size of the object crosses the limit that forces the arm to use the Big Components, the Small Components should slide to their maximum, so that the Big Components only have to be extruded by the smallest amount necessary, as it was done in Figure 2.23d. The arrangement in Figure 2.24c is also invalid, because the small arms cannot slide out of the big ones.

(a) Valid
(b) Invalid
(c) Invalid

Figure 2.24: Valid and invalid dispositions for the arm of the robot (1)
All the previous images show both sides of the arm with a positive radius, but it will be allowed to move them across the central plane of the robot, one at a time. The previous principle of dislocating the big component by the smallest possible amount is kept here. The position of the big component that crosses should be aligned with the one of its small component, so that the local position of that component is zero, as shown in Figure 2.25b. The only case when this is not observed is when it would cause the big component to overlap the opposite one. In those cases, it should go the maximum extent permitted, keeping a continuous bridge for the small component, as shown in Figures 2.25a and 2.25c.

(a) Valid  
(b) Valid  
(c) Valid

Figure 2.25: Valid and invalid dispositions for the arm of the robot (2)

The ability to have different and negative radius for the sides of the arm might seem an unnecessary complication, since the robot could grab that same object by centering the arm with the same diameter. However, increasing now the complexity of the arm behavior will allow later the simplification of the decision process used to position and rotate the body of the robot to grab an object.

2.3.3 The docking station

The docking station is where the robot rests and recharges when it is not being used. It will be fixed to a wall of the space station.

(a)  
(b)

Figure 2.26: Docking station for the robot
3 Unity

3.1 Software overview

Unity is a game engine developed by Unity Technologies. It offers a wide range of features, that include the creation of three-dimensional games and virtual reality simulations. 25

3.1.1 Interface

The Unity interface is divided into areas, as displayed in Figure 3.1.

![Figure 3.1: Unity interface with seven different areas](image)

The ones most commonly used in this project were: 26

- Project Window (outlined in green): displays the library of assets imported into the project that can be used to build the game. There several different types of assets, including animations, audio clips, fonts, materials, meshes, models, prefabs, scenes, scripts and textures.
- Scene View (outlined in blue): used to visually navigate and edit the current scene, by interacting with its objects. The perspective in the Scene View is independent from the camera that renders the game.
- Hierarchy Window (outlined in purple): lists the names of all objects in the scene. If an object is added or removed, it will also appear and disappear from the hierarchy.
- Inspector Window (outlined in yellow): used to view and edit all the properties of an object. It is context sensitive; different types of objects have different properties, so the content of this window depends on the currently selected item.


Game View (outlined in red): used to preview and test the game while in the editor, without having to publish it. The user can play, pause or step frame by frame in the game. It is also possible to change the properties of the objects and observe the result without restarting everything. Even if the developer adds or deletes objects, these changes are temporary, and are always reverted after exiting play mode.

Console Window (outlined in orange): shows warnings, errors and debug messages generated by the user in scripts.

Toolbar (outlined in pink): contains the transform tools, the gizmo toggles, the play controls and the layers and layout menus.

### 3.1.2 Scenes, game objects and components

A project can have several scenes, and each scene can have several game objects. The game object is the basic element in Unity. They can be grouped together in nested parent child relationships. Each object can have many children, but it can only have one direct parent.  

Each game object behavior depends on the components it holds. All game objects, even the empty ones, have a Transform Component, which defines the position, rotation and scale of that object. The values that appear in the inspector are the local ones (the local position, rotation and scale of the game object in relation to its parent), but the global position and rotation (the values that would appear in the inspector if the game object was placed in the root of the hierarchy but keeping its location and orientation in the scene) can be accessed in scripts. Other types of components can be added to achieve the desired functionality. For example, a piece of scenery would have a Mesh Filter and a Mesh Renderer, assigned with the meshes designed previously in the modeling program and the material that contains the images created in the texturing program. It would also have a Collider Component.

### 3.1.3 Bounds

The bounds represent a bounding box aligned with the world’s coordinate axes, fully defined by its center and size. It is possible to obtain the bounds of an object in its local space or in world space. Both of them are boxes that completely enclose the object, but the first considers the space occupied by the object when it is at the origin of the scene, with null rotation and normalized scale, while the second takes into account the current transform of the object.

### 3.1.4 Colliders

Colliders are invisible components used in collisions to represent the actual physical space that the game object occupies. Unity offers four main types of colliders: the Box Collider, the Sphere Collider, the Capsule Collider and the Mesh Collider. The first three are called primitive colliders and have defined geometries that can be positioned and scaled in the settings. The last one has the shape of the attributed mesh, previously modeled and imported into Unity. When deciding which collider to use, the first option...

---

is to consider only one primitive collider. The second option is to use a combination of primitive colliders, called a compound collider. The third option is to use a simplified mesh based on the mesh of the object. The fourth and last option is to use the exact same mesh that is used to render the object. These options are ordered by their toll on performance and the next one should only be considered when the previous one is not satisfactory. Most of the time, the collider does not have to be in the exact same shape of the object and a rough approximation goes unnoticed by the user.  

All primitive colliders are convex and can collide with any type of collider. A mesh collider might not be convex; to turn it into a convex collider, a box can be checked in the settings. For some colliders - the ones that were originally concave -, doing this will change their shape, so that all depressions are filled in. Two concave mesh colliders do not collide with each other. For that to happen, at least one of them needs to be marked as convex.

To optimize performance, a mesh used only as a collider can have its normals disabled when imported, because it will not be rendered.

Usually, when two objects with colliders hit each other, the physics engine will make the necessary calculations to simulate a collision and compute the new positions and rotations for each object in a way that avoids having parts of them occupy the same volume in space. However, in some situations, it may be desirable to detect when a collider enters the space of another one, without actually causing a collision. If a collider is marked as a trigger, another collider (whether or not it is a trigger itself) will pass through that first collider instead of bumping into it. This can be detected in scripts and the necessary actions taken by code.

### 3.1.5 Rigidbodies

Rigidbodies components are used to mark game objects that should be taken into account by the physics engine.

Not all game objects in the scene should be rigidbodies. A rigidbody needs to have a collider, so that it can interact with other physics objects; it does not make sense to have a rigidbody without knowing the physical space that object occupies. However, not all objects with colliders should also be rigidbodies. A game object with a collider can either be a static collider, if it has no rigidbody, or a dynamic collider, if it has one. Dynamic colliders can also be divided into kinematic colliders and non-kinematic colliders.

A static collider is used for scene geometry that is never moved during the game and does not respond to collisions, such as trees and walls. These objects will cause others to collide with them, but will not be affected by those collisions themselves. For example, a character that goes against a wall will be stopped, but the wall should not be shoved out of its place when sustaining that bump. When the game starts, all static colliders are checked once by the physics engine and never checked again, unless one of them moves during the game. In that case, not only that one but all static objects have to be checked again. This takes a toll on performance and can leave the collider in an undefined state that

---


causes incorrect results, for example when casting rays or awakening rigidbodies hit by those moving static colliders. For these reasons, a static collider should not be changed during runtime, either by moving it, rotating it, scaling it, or enabling/disabling it. 29

A kinematic collider is used for objects that are moved during the game but should not respond to collisions. Examples of these are rotating fans or sliding platforms. The solution to the limitations mentioned for static colliders is to use kinematic rigidbodies. Just like static colliders, kinematic colliders are not affected when slammed by rigidbodies, but the physics engine assumes that they can be moved and the major burden on performance explained before is avoided. 29 These objects are not controlled by the physics engine; they are moved by scripts, using their transform in the Update function. 31

A non-kinematic collider is used for objects that are moved during the game and should respond to collisions. The most obvious example is the player game character. These objects can receive forces and torque to act in a realistic way and are fully simulated by the physics engine. They react to static colliders (for example, a character that comes to a halt when walking into a wall), to kinematic colliders (for example, a character that is elevated in the air if he steps onto a platform that moves upwards) and to other non-kinematic colliders (for example, two characters that bump into each other). A non-kinematic collider is sometimes called a rigidbody collider. It can be assumed that this expression refers to a normal rigidbody unless it is explicitly stated that the rigidbody is marked as kinematic. 29 Non-kinematic objects can be moved both by collisions and by scripting, using their rigidbody in the FixedUpdate function. The transform should not be used to move this type of collider, because it may cause problems with physics calculations. 31

A rigidbody can have a primitive collider, a convex mesh collider or a combination of these. Concave mesh colliders are not supported. Usually, the best option is to use mesh colliders, concave if necessary, for the scenery, and compound primitive colliders for moving objects, such as the character. 31

When a game object has children, only the parent needs a rigidbody component, which will take into account the parent’s collider and the children’s colliders. 31

Some of the settings that affect the behavior of each rigidbody are: the mass, the drag, the angular drag, the use gravity property, the is kinematic property, the interpolation, the collision detection and the constraints. 31

The mass is used to compare how two rigidbodies react when colliding with each other: the rigidbody with higher mass will react less than the one with lower mass. It is used as a ratio; in fact, the actual mass of the rigidbody is less important than the size of the mesh. If the game object is behaving incorrectly, the best option is to scale the mesh before considering the mass. This scaling should be done in the modeling software, so that the imported mesh has a uniform scale in Unity, because objects with non-uniform scales require more work from the physics engine. For a game that uses physics, it is crucial to make the dimensions of the objects similar to the ones they have in the real world. For example, a human character could be 5 cm tall and the user would not find it strange, assuming that the whole scene had been scaled down to keep the proportions. However, it would be noticeable if the game used physics, because a 5 cm sized object is not affected by gravity and other forces in the same way a 5 m one is. For this reason, Unity advises to make humanoid characters around 2 meters tall in the modeling software. 31
The drag and the angular drag measure how much the air resistance affects the object when moving and rotating it using forces and torque, respectively. Low values make the object seem heavy and high values make it seem light. Zero means no resistance at all, so if a force is applied to that object, it will always keep moving, even after the interaction ended. A positive value will allow the object to cover some distance, slowing it down until it stops; the loss rate in velocity will depend on how high is the value. It can also be set to infinity, which means that the object will stop moving as soon as the force is no longer applied. However, it should be noted that it is not possible to stop an object from rotating just by setting its angular drag to infinity.  

The use of gravity can be unchecked to prevent gravity from affecting the object, defined in Unity as a force with a value of -9.81 in the Y axis.  

The kinematic box can be checked so that the game object, in spite of having a rigidbody, does not react to physics but can be properly moved during the game.  

The interpolation can be used to prevent bumpiness in the motion of the object, based on the transform of the previous frame or the predicted transform of the next frame. It should be set to none if the object already moves smoothly without any interpolation.  

The collision detection can be adjusted to prevent high speed objects from going through other objects so fast that their colliders are not detected.  

The constraints allow the developer to restrict the motion of the rigidbody, by freezing its position or rotation in a combination of axes.  

### 3.1.6 Prefabs

When the file exported from the modeling software is imported into Unity, it is called a model. This model can be dropped in the scene to create a new game object. This object already has the mesh components (with the mesh assigned and the material unassigned), but the collider has to be created and scaled. This is acceptable if there is only one instance of this model in the entire scene. However, some models will be used several times, which requires adding a new collider with the same settings as the first one to each duplicate. If the collider needs to be adjusted later, the changes must be repeated for all objects. This time-consuming process can be avoided by using prefabs.  

A prefab is a game object configured in a specific way, with all its components and properties, that is stored in the project and can be instantiated in the game at any time. It can be created from any kind of game object; picking up from the previous example for a piece of scenery, after the first model is instantiated in the scene and the collider is added, the game object needs be dragged and dropped back into the project to create a prefab. Now, when a new clone with this mesh is required, the prefab should be the one that is dropped in the scene, not the model. Any changes made to the prefab, such as adding, removing or changing components, will be replicated to the copies of that prefab, automatically synchronizing all the instances. However, this does not mean that all duplicates have to be exactly the same. The developer can modify one them directly and choose to override the original settings to make this instance unique in some aspects. For example, instances of the same prefab can have a different

---

material or scale.  

If a model is accidentally deleted from the scene, all the work done in the components is lost. A prefab, on the other hand, can have no instances at all in the scene when the game starts. It can be instantiated only at runtime, which is useful for game objects that are created as consequence of an action of the player, for example, a projectile when shooting a gun.  

3.1.7 Layers

Layers can be used to selectively ignore colliders when using the cast functions from the Unity Physics library. Each game object can be assigned only to one layer. There are 32 layers in Unity; of these, 8 are builtin and cannot be changed, but the remaining 24 are available to the user, who can name them how he sees fit.  

3.1.8 Scripts

Scripts are written in C# using the Scripting API that comes with Unity. Some scripts are behavior components that can be added to game objects, just like the types of components mentioned before. These scripts can modify the game object and its other components during runtime. They can also modify other game objects in the scene (if provided with a reference to them), add new objects or delete the existing ones. A behavior script stored in the project is only a blueprint of a component and has no effect unless it is added to a game object. Different game objects can have instances of the same script; those instances share the same template, but have different references. Changing a variable on one of them will not affect the others.  

Usually a behavior script is organized with the global variables at the top, the event functions in the middle and additional functions at the bottom.

The global variables are the ones that can be used by all functions in that script, as opposed to the ones that are declared inside each function. As for their scope outside of the script, they can be declared as public or private. It is good practice to make all variables private except the ones that have to be accessed by other scripts and the ones that the developer wishes to make visible in the inspector. These can be assigned directly in the editor, without having to open the script, which is useful to make quick adjustments to the speed of a game object or to create references to other game objects, for example.

The event functions have established names and are called at specific times during the game. A script in Unity does not follow the conventional idea of a program that is executed, always has control and runs continuously until reaching its end. Instead, Unity will call a certain function of a script when the time is right and then reclaim the control of the game as soon as that function finishes. A script is not executed uninterruptedly; it is called intermittently when it contains the correct event function for the current state of computation. The most commonly used event functions can be divided into initialization events, regular update events and physics events.  

---

Initialization events are called only once in the lifetime of a script attached to the game object. The Awake function is called first, when the scene is loaded, even if the script is not enabled. The Start function is called after Awake and before any update events, but only if the script is enabled. Though the Awake and Start functions on different game objects are called randomly, the last Awake function has to be finished before the first Start function is called. The Awake function should be used to initialize variables that will be accessed by multiple scripts in the Start function, to make sure that they are all synchronized. The Start function can also be used to delay any initialization that depends on the state of the game for an object that is not active when the game starts.  

Regular update events are called several times if the script is enabled. The Update function is called once per frame and it should be used to receive input from the user and move non-physics objects. The LateUpdate function is also called once per frame, after all Update functions for all enabled scripts have been called. This function is useful to keep variables synchronized, just like it was mentioned before for the Awake and Start functions. Updates that require coordinated information from more than one script should be done in this function, to avoid making calculations with part of the information already updated, relative to the next frame, and the other part still relative to the previous one. The FixedUpdate function is called once per physics step and it should be used to move physics objects. 

Physics events are called when the physics engine reports a collision against the object that holds the script. These functions are OnCollisionEnter, OnCollisionStay and OnCollisionExit, called when contact is made, held and broken, respectively. If the collider is checked as a trigger, the functions OnTriggerEnter, OnTriggerStay and OnTriggerExit are called instead. Unlike the FixedUpdate function, these functions are not called regularly. 

Finally, the additional functions can be used to organize the code when it is long and it would be impractical to fit it all inside the previous functions. They are only executed if called by one or more of the event functions. 

There are also scripts that are not behavior components and are not attached to any object in the scene. These can be used to store classes or static methods, both of which can be implemented by behavior scripts. The classes created can be organized into different namespaces. There cannot be two classes with the same name inside the same namespace, even if they belong to different files. 

3.1.9 Time and framerate management

The frame updates and physics updates do not occur with the same frequency. 

The Update function is called in irregular intervals, depending on how long each frame takes to process. When an object is translated with constant speed using its transform inside the Update function, it cannot be moved by a fixed distance each frame. The render time of the frame needs to be taken into account to make sure the object is shifted by different amounts but at the same speed. 

The FixedUpdate function is called at consistent intervals, which means that the time between calls is always the same. When an object is translated with constant speed using its rigidbody inside the

---

FixedUpdate function, it is not necessary to adjust for the render time of the frame.  

3.1.10 Physics functions

The Unity Physics library offers a set of functions that can be used to cast or overlap shapes in the scene and check for collisions against objects with colliders. These functions return information on what was hit and where it was hit. They can receive a layer mask to take into account only the objects that belong to a set of layers. The ones used in this project can be divided into cast functions and overlap functions.  

The cast functions are used to sweep the scene, along a ray with a given origin and direction, and optionally a maximum length. It is possible to cast only a ray or to cast a shape along that ray, which can be thought of as a thicker ray. The Raycast function requires only the parameters mentioned previously, but the BoxCast function also requires the size and the rotation of the box, while the SphereCast function requires the radius of the sphere.  

The overlap functions are used to inspect a given volume in the scene and check if there are any colliders totally or partially inside that volume, or even only touching the volume. The BoxOverlap function requires the center, the size and the rotation of the box. The SphereOverlap function requires the center and the radius of the sphere.  

3.1.11 Performance

A game can be understood as a movie whose frames - both their number and their content - are not determined when it starts. The film industry also uses 3D modeling programs, but movies can achieve a realistic look that is impossible for a game to reach. To make a movie, film animators model and place characters and objects in a lighted scene. A virtual camera then captures their movements, prompted by animations and simulations. Each frame taken by this camera can take hours or days to render [7].  

For example, it took about 29 hours to render a single frame of Monsters University by Pixar, using one of the top 25 supercomputers in the world, composed of 2000 machines and 24000 cores. With 24 to 60 frames per second on a two hour long movie, 100 million CPU hours were required; with only one, it would amount to ten thousand years. For Pixar’s render farms, it took a couple.  

In video games, the player controls the character’s actions; each frame will depend on the choices made and therefore will have to be rendered in real time. Just like in a film, the camera captures the shapes and textures of the models, the lights and the shadows and the interactions between game objects. This is why it is so important to keep performance goals in mind when working on these elements. To guarantee that the frame flow runs evenly, the developer needs to simplify the meshes, the materials, the collisions and the lights, but without compromising too much the quality of the game.  

3.2 The simulation

3.2.1 The interactions

The simulation is organized into four major elements: the station, the canvas, the player and the robot.

The station encompasses every piece of scenery, including the modules, the objects that form the content of those modules, some of which can be grabbed, and the colleague astronauts, which are non-playable characters and therefore considered part of the environment, entirely controlled by the game. It interacts with the player and the robot by providing a scenario that bounds them to a certain volume in space. If they try to breach the frontiers of that volume, the station originates collisions in order to stop them.

The canvas displays texts on top of the screen, such as the menus the user can explore to affect the behavior of the station, the player and the robot. It interacts with the remaining three elements when it transmits a request from the user, already complete with all the information necessary. It also protects them by screening for some erroneous user input; however, it only checks for generic errors, that can be made for any of the three elements.

The player is the astronaut that fulfills the role of the only playable character in the game. The user roams the station using the body of the player, grabs objects using his arm and sees everything around him through his eyes.

The robot is able to transport objects between their original places in the station and the hands of each of the astronauts. It obeys in part to requests made by the user and to its own internal programming. The user is offered the possibility of issuing only high-level requests. He is not be bothered with having to make decisions about actions concerning low-level operations, such as opening the arm or positioning the body of the robot. The user only needs to indicate the origin and the destination of the object; from that point, the robot handles autonomously the problems posed by the request and the steps necessary to accomplish it.

The most complex interaction in the entire simulation is the one between the player and the robot, when the robot either receives an object from the player or delivers an object to the player. Several problems have to be solved in order to complete both of these tasks successfully.

The first step is to locate the player inside the station. More specifically, the robot needs to decide which position, somewhere above the palm of the hand of the player, is appropriate to carry out the remaining operations. These operations differ on the volume of available space the robot requires in order to perform them without colliding with the player. After approaching him, the orientation of the robot may not be the appropriate one to either grab or drop an object. This means that the robot needs to rotate, until finding an orientation that suits the hand of the astronaut and the shape of the object that is being exchanged. In order to rotate freely, without hitting the player, the robot needs to be at some distance from the hand, leaving enough empty space between the two. However, the actions that follow this step require the robot to be closer to the hand. It cannot grasp the object or put it down in the palm of the hand if it is too far from the player.
The robot is therefore tasked with finding not one location but two: both of them above the palm of the hand of the player, one slightly more distant than the other.

The first location, the one farther from the hand, is an approach position that the robot should take to carry out the initial actions involved in exchanging an object. The robot enjoys a greater degree of freedom while at this position, since it can rotate to any orientation or spread the arm to its maximum span without colliding with the astronaut. This could also be the target position when the robot finishes an action. The robot can return to the approach point after picking up or dropping off an object. This guarantees that the robot is far enough from the player to avoid disturbing him while he performs other actions unrelated to the robot, but close enough to respond promptly if the player makes a new request.

The second location, the one closer to the hand, is the contact position, where the robot carries out the intermediate actions required to transfer an object. The robot cannot perform broad movements while at this point. Only more delicate operations, such as slight modifications to the opening span of the arm, should be executed. Even at this position, the robot is not supposed to touch the astronaut. It should be close enough to the hand to be able to make contact with the object when grabbing it, or to cause the object to contact with the player when dropping it. There can be contact between the player and the object, and between the object and the robot, but not between the player and the robot.

The second step is to decide which orientation the robot should have when reaching for the player’s hand. Some of the requirements associated with this stage are specific to when the robot is either grabbing or dropping an object, while others are shared by both situations.

Whether the robot is grabbing or dropping an object, its arm should be facing the direction of the palm of the hand, because it is not possible to exchange the object if the bottom of the robot is facing the opposite direction. Therefore, the orientation of the robot depends greatly on the orientation of the player. If the player is standing upright, with his hand turned to the top panels of the station, the robot should point its arm in the direction of the bottom panels. This is the most natural and obvious configuration. However, due to the microgravity conditions experienced in space, the astronaut is able to adopt a wide range of postures. For example, he can be upside down, with his feet pointing to the top panels of the station and his hand turned to the bottom panels. In this case, the robot needs to rotate to match the astronaut, turning its arm in the direction of the top panels. He may also position himself sideways, floating horizontally. In this case, the robot would usually have to perform a quarter of a turn in relation to its default orientation. All other intermediate situations should be considered, enabling the robot to respond to any player movement. It should be noted that, although the direction of the hand depends on the orientation of the body, the arm is also able to move independently from the rest of the body. What really matters is the direction the palm of the hand is pointing to. For example, the player may be in the upright position but his hand can be facing up, down, right, left or any direction in between. When the hand is not pointing up, the robot is required to maneuver in order to match it, even if the astronaut is standing straight.

When the robot is grabbing an object, it needs to acquire an orientation that allows the arm to frame an adequate part of the object, while avoiding hitting the hand of the astronaut. The robot cannot try to grab a portion of the object that is covered by the fingers or that is larger than the maximum span.
span of its arm. It should also try to orient itself with the symmetry axes of the object, if it has any, and center the object with its body, bringing it as close to the object as possible. The robot should not grab the object sideways, because that makes it easier for the object to slip; it should also not grab it by its tip, because that maximizes the volume of the object that is outside the space occupied by the robot, making it harder to navigate the station while transporting the object.

When the robot is dropping an object, it has to find an orientation that allows it to deliver the object in the most natural way for the astronaut to properly grab it. The player should not be required to make any adjustments with his hand; it is the robot that has the responsibility of identifying how a person would grab an object if they were picking it up, and then mimicking that orientation. Several complications have to be prevented at this stage. The robot cannot place the object too close to the tips of the fingers, or too close to the wrist. The object should be placed in the middle of the palm of the hand, to make it easier for the astronaut to secure it. The robot needs to put it down gently, with a reasonable amount of contact between the surface of the object and the palm. The object should not be left hovering above the hand; it should also not try to go further than the hand, pushing it down and possibly hurting the astronaut. Finally, the robot should rotate in a manner that avoids getting its arm in the path of the astronaut’s fingers when they close around the object.

The third step is to find the correct value for the opening of the arm. While the robot is holding an object, the two sides of the arm should be closed tightly around it. This means the arm has to slide in to secure an object, and slide out to release it.

When the robot is grabbing an object, it could reach for it with the arm fully distended, and then close each side until making contact with the surface of the object. However, this operation has to be performed at the contact position; one of the goals of the interaction between the player and the robot is to minimize the amount of time and the range of movements the player performs at that location, in order to limit the probability of a collision or conflict between the two. It would be preferable if the robot was able to evaluate the diameter of the object while at some distance from it, from the approach position. The robot could then slide the arm beforehand, setting it to the size of the object plus a small margin. At the contact point, the arm could be rapidly adjusted to cover the added margin, closing firmly around the object.

When the robot is dropping an object, the same requirements established in the previous paragraph also apply. The robot needs to slide each side of the arm out, in order to release the object. But it is not necessary or desirable to distend it until reaching the maximum span, because that would require the astronaut to wait longer before taking possession of the object and being able to start using it. The robot only has to open the arm enough to avoid scratching the object when retreating, with the same margin used to grab the object. After the robot backs away from the player, it can reposition the arm to another opening span. The default configuration, that the robot acquires while it is idle, is with the big components completely closed and the small components completely open. The big components should not be extruded when they are not needed, because they increase the volume of the space occupied by the robot and reduce its navigation flexibility. The arm components should be distended because that configuration is the midpoint between the minimum and maximum span values. Most of the objects
that can be grabbed by the robot have dimensions that require the arm to be approximately at that
configuration. If the robot restores the arm to that opening span after working with an object, it will
usually reduce the time consumed with adjustments later, when grabbing a new object.

The fourth step is to take into account the movements of the player. When an object is exchanged
between the two, the player also performs a few gestures in order to release or accept an object. These
gestures can be grouped into two main categories: finger gestures and arm gestures.

When the player is giving an object to the robot, he has to open his fingers in order to let the robot
retreat with the object. The robot should not try to forcibly extract the object from the hand. When
the player is receiving an object from the robot, he has to close his fingers before the robot slides the
arm out. The robot should not let go of the object without making sure that the player has secured it.
Without this precaution, there could be loose objects floating around the station, if the astronaut asked
the robot to bring him an object but then failed to grab it, either by accident or because he decided he
no longer wanted that object.

When the player is giving or receiving an object, he can also perform some arm gestures that
expedite the process of exchanging an object. For example, if the astronaut is in the most common
position, standing upright, he can rotate his arm in order to turn his hand up while interacting with the
robot. This prevents the robot from having to rotate to find the palm of the hand. The player is more
agile than the robot; he needs less time to turn his hand than the time the robot takes to rotate around
it. After the interaction ends, the astronaut can turn the hand down again, to a more natural position.
This gesture can also be used to communicate to the robot whether or not the astronaut is currently
interested in exchanging the object, even after requesting to do so. For example, when bringing an object
to the player, the robot can stop above the back of his hand, waiting for the player to turn it up. If the
player does not turn it up immediately, it means that he might still intend to receive the object, but not
just yet. If that happens, the robot should wait near the hand, until the astronaut finishes the other
task that is keeping him occupied and is finally available to receive the object. At that moment, the
player can turn the palm of his hand in the direction of the robot, with his fingers fully stretched. The
robot interprets this gesture as an invitation to put down the object and only then tries to complete the
current action. This method prevents the robot from abruptly approaching the astronaut and forcing an
object into his hand, even if he has his fingers partially or completely closed, without any regard about
his receptiveness to that intent.

The requirements established can be organized into a sequence of actions that the robot should
take in order to adequately receive an object from the player or deliver an object to the player. These
actions are displayed in Figures 3.2 and 3.3. The images do not include any station reference points, to
emphasize the notion that the interaction can occur for different orientations. However, it is possible to
deduce that these images concern the default player posture - standing vertically, with the head pointing
to the top panels -, by observing the initial and final stages. When the robot is idle, its default orientation
is parallel to the bottom panels of the station, with the arm pointing down. The rotation angle about
the vertical axis depends on the last action performed and can take any value. For example, in Figure
3.2a, if the astronaut had his head pointing towards the bottom panels of the station, he would see the
robot upside down.

Figure 3.2 displays the main stages involved in grabbing an object. In Figure 3.2a, the player is holding an object and the robot is at some distance from the player, waiting to be given a task. In Figure 3.2b, the robot locates the player and positions itself at the approach point. In Figure 3.2c, the robot rotates to an orientation that allows it to grab the object. In Figure 3.2d, the robot slides the arm to match the diameter of the object, with an added margin. In Figure 3.2e, the robot reaches for the object, positioning itself at the contact point. In Figure 3.2f, the robot slides the arm to close it completely around the object. At this stage, the robot should wait for the player to open his fingers, for as long as it takes. In Figure 3.2g, the player opens his fingers. In Figure 3.2h, the robot retreats to the approach point. In Figure 3.2i, the robot rotates to the default orientation, but the change is imperceptible because the robot was already close to the desired value. The interaction is completed; the player can turn his hand down again and move away from the robot.

Figure 3.2: Robot grabs an object from the hand of the player
Figure 3.3 displays the main stages involved in dropping an object. In Figure 3.3a, the robot is holding an object, waiting to be told where to transport it, and the player has his hand empty, facing down, with the fingers slightly closed in a relaxed position. In Figure 3.3b, the robot locates the player and positions itself at the approach point. In Figure 3.3c, the robot rotates to an orientation that allows it to properly deliver the object. That orientation depends on a prediction made about the future location of the palm of the hand when the player turns it up. At this stage, the robot should wait until the player rotates his arm. In Figure 3.3d, the player turns the hand up and stretches his fingers. In Figure 3.3e, the robot places the object in the palm of the hand, positioning itself at the contact point. At this stage, the robot should wait until the player closes his fingers. In Figure 3.3f, the player closes his fingers. In Figure 3.3g, the robot releases the object, by sliding out each side of the arm. In Figure 3.3h, the robot retreats to the approach point. In Figure 3.3i, the robot rotates to the default orientation and slides the arm to the default span. The interaction is completed; the player can move away from the robot.

Figure 3.3: Robot delivers an object to the hand of the player
The remaining robot interactions are more simple than the one with the player. They make use of some of the concepts involved in exchanging an object with the player, but ignore the unnecessary ones. The robot can also receive an object from a colleague astronaut or deliver it to him; and it can pick up an object stored inside the station or put it back in its original place. All of these interactions involve identifying the pick-up or drop-off location - both the approach point and the contact point -, rotating to an adequate orientation and maneuvering the arm. However, they do not need to take into account any finger or arm gestures: the station for obvious reasons and the colleague astronaut because those parts of his body were not animated.

3.2.2 The managers

All four major elements of the simulation are controlled by one or more dedicated scripts called managers. A few key principles were observed when designing these scripts. First, each manager is tasked with a specific functionality or a group of closely related functionalities. A manager should not be overloaded with a diverse range of responsibilities. The functionalities are not repeated in another manager; they are clearly separated between each manager to avoid duplicating the purpose of a manager or overlapping between two of them. Without this separation of duties, there would be confusion on which manager had the responsibility of carrying out a given task, or which manager was causing problems when a task failed. Finding the source of errors would also be harder if there were fewer managers, each of them burdened with a greater number of tasks. Secondly, each manager knows as little as possible about the internal workings of the other managers. If the need for two managers to interact is unavoidable, the interface is kept simple and reduced to the absolute minimum required. This facilitates making changes on each manager, without having to worry about altering the other managers as well, as long as the interface remains the same. Finally, the same naming convention and programming style were maintained throughout the project. The managers share similarities in the names of the classes and in the structure of the methods, which is important to keep them consistent and to make it easier for someone not familiar with the project to get a quicker grasp on how it is organized [8] [9].

Some concepts are shared by several managers: the command, the instruction, the state, the order and the share classes, to name the most important ones. Their common characteristics are described in the following paragraphs. When they are mentioned again in the sections specific to each manager, only their differences from the basic structure are discussed. For a complete picture of the concept, it is necessary to refer back to this section.

A command is created when the user goes through the menus to issue a request, targeting the station, the player or the robot. It is used to receive complex input information from the user, that cannot be conveyed using a single keyboard shortcut. It can be evaluated on two parameters used to control the flow of commands between the managers: whether it is complete and whether it was requested. A command is marked as complete only when all options required were chosen by the user. It is considered incomplete while it is being generated by the canvas; when that process finishes, the command follows from the canvas to one of the three main managers, which are always listening for a complete command with the correct target. Upon detecting one, the manager checks if it is already following another instruction,
generated by a previous command. If not, the manager creates a new instruction based on the specifics of
the command. The command leaves the canvas manager marked as not requested. When the target main
manager sees that command, it always marks it as requested, even if it will not actually be obeyed, either
because the manager is still following another one or because it is not valid in the current circumstances.
This means that the command does not get suspended, waiting for the manager to finish or for the right
set of conditions. That could be confusing for the user, since a command issued some time ago could
suddenly awake, after the user had forgotten about it and when it was no longer wanted.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string target</td>
<td>The main manager targeted by this command.</td>
</tr>
<tr>
<td>bool isComplete</td>
<td>True if the command is complete.</td>
</tr>
<tr>
<td>bool wasRequested</td>
<td>True if the command was requested.</td>
</tr>
<tr>
<td>List&lt;SubCommand&gt; subcommands</td>
<td>A list with all subcommands.</td>
</tr>
</tbody>
</table>

Table 3.1: Public variables in the Command class

An instruction is a command translated to a language that the manager can understand. It can be
evaluated on two parameters used to control the flow of instructions inside each manager: whether it is
valid and whether it was requested. It is considered valid if all conditions necessary to follow through
with this specific request are met. Marking an instruction as invalid prevents the manager from trying to
follow through with a user command that it cannot obey due to the current circumstances of the game.
The instruction is considered requested if the manager started obeying it. It is marked as not requested
when created and as requested on the frame its starts getting processed. This prevents the manager
from trying to feed the state machine an instruction that is already being followed. Any instruction is
divided into subinstructions, which are ordered steps that the manager has to take in order to complete
the instruction. Only when the current subinstruction is fulfilled can the manager move on to the next.
Each manager has several different types of subinstructions, with specific responsibilities.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int number</td>
<td>The number of subinstructions.</td>
</tr>
<tr>
<td>string name</td>
<td>The name of the instruction.</td>
</tr>
<tr>
<td>bool isValid</td>
<td>True if the current state of the game meets all required conditions.</td>
</tr>
<tr>
<td>bool wasRequested</td>
<td>True if the manager started obeying this instruction.</td>
</tr>
<tr>
<td>Dictionary&lt;int, SubInstruction&gt; subinstructions</td>
<td>A dictionary with all subinstructions.</td>
</tr>
</tbody>
</table>

Table 3.2: Public variables in the Instruction class

The state characterizes the current condition of the manager. Each newly created instruction
enters a state machine, that decides if it should proceed with it. When it does, it progresses through the
instruction taking up one subinstruction at a time, in sequential order, until the last one is completed.
The current situation of the state machine is identified with a number. All managers with state machines
have at least five situations. In regular circumstances, the state machine only sees those first five, using
them to initiate and successfully finish an instruction. Some managers have additional situations, that
happen when it is necessary to deal with abnormal events.

In situation one, the manager is idle, waiting for a new instruction to arrive. In situation two, the
manager prepares to start obeying an instruction, by setting the isFollowingInstruction state variable to true, assigning the new instruction to the state instruction and the first subinstruction to the state subinstruction. In situation three, the manager is following a subinstruction, which might take one or several frames to complete. In situation four, the manager has finished a subinstruction, but not the last one in the instruction. The next subinstruction in the docket is assigned to the state subinstruction. In situation five, the manager has successfully completed the last subinstruction and therefore the full instruction. The isFollowingInstruction state variable is set false.

From situation one, the manager can switch to situation two if a new valid instruction is detected. From situation two, the manager always switches to situation three one frame later. From situation three, the manager can switch to situation four, if the current subinstruction is complete but it is not the last one, or to situation five, if the current subinstruction is complete and is the last one. From situation four, the manager always switches back to situation three one frame later. From situation five, which also lasts only one frame, the manager always goes to situation one.

These different situations can be used to take different actions in the frame at the start of the subinstruction (initialization tasks, that should only be executed once), in the frames that occur during the subinstruction (recurrent tasks, that have to be executed more than one time), and in the frame at the end of the subinstruction (finalization tasks, that should only be executed once).

The block of code that executes a subinstruction is divided into sections. Each section is appropriate to deal with a given type of subinstruction during a given situation. These two parameters - the subinstruction type and the situation number - are used to guide the execution through the adequate path, entirely ignoring the sections of the code that are not relevant in the current frame.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int situation</td>
<td>The situation number.</td>
</tr>
<tr>
<td>bool isFollowingInstruction</td>
<td>True if the manager is currently following an instruction.</td>
</tr>
<tr>
<td>Instruction instruction</td>
<td>The current instruction.</td>
</tr>
<tr>
<td>SubInstruction subinstruction</td>
<td>The current subinstruction.</td>
</tr>
</tbody>
</table>

Table 3.3: Public variables in the State class

An order is used to exchange information between a main manager and one of its sub-managers. When the user interacts with the canvas, an external request - the command - is created, which then prompts an instruction inside the main manager. Some of the subinstructions of this instruction are internal requests - orders - directed to the sub-managers. An order can trigger a local instruction inside the sub-manager, which runs parallel to the global instruction in the main manager. Only when the local instruction finishes can the global subinstruction that initiated it be considered complete. The main manager then follows to the next subinstruction on the docket. This process assures that the main manager never tries to issue an order to any of the sub-managers while they are still dealing with an instruction deduced from a previous order.

Each order belongs to a given sub-manager and is based on that script, which means that the order does not require a variable with the name of the target. It only needs the name of the order, which must
be the name of a local instruction recognized by that sub-manager. The two managers also use a set
of switches to inform each other on the current state of the order. When the main manager wants to
give an order to a sub-manager, it writes the name of that order on the current order variable. Then,
it sets the was requested switch to true, to signal that there is a new order waiting for the sub-manager
to examine it. When the sub-manager detects that the switch is on, it turns it off so that the same
order is not evaluated twice. It then creates an instruction based on that order. If the instruction is not
valid, the sub-manager sets the was denied switch to true and does nothing else with that instruction. If
the instruction is valid, the sub-manager sets the was approved switch to true. This switch is detected
by the internal state machine of the sub-manager, which initiates the instruction when it sees that the
switch is on. It is turned off right after the instruction starts, to prevent the same instruction from being
executed more than one time. When the sub-manager finishes the instruction, it sets the was completed
switch to true. After issuing the order, the main manager is always checking both the was denied and the
was completed switches. Only one of them will be turned on. If the order is denied, the main manager
quits the global instruction. If the order is completed, the main manager has successfully completed the
current subinstruction and can continue to the next one. While the sub-manager is executing an approved
instruction, it may happen that the main manager decides that it no longer wants the sub-manager to
go through with that order. The main manager will then turn on the was cancelled switch and the
sub-manager will quit the order, without finishing it.

<table>
<thead>
<tr>
<th>Orders</th>
</tr>
</thead>
<tbody>
<tr>
<td>string currentOrder</td>
</tr>
<tr>
<td>bool wasRequested</td>
</tr>
<tr>
<td>bool wasApproved</td>
</tr>
<tr>
<td>bool wasDenied</td>
</tr>
<tr>
<td>bool wasCancelled</td>
</tr>
<tr>
<td>bool wasCompleted</td>
</tr>
</tbody>
</table>

Table 3.4: Public variables in the Orders class

The share class was created to exchange information between two main managers, when the execution
of two commands with different targets requires coordination between the actions of each manager.
It is possible to have only one share system based on one of the managers, when the second manager
needs information about the activities of the first one, but the first does not require any from the second.
However, they were implemented in pairs - two symmetric systems, one based in each manager - because
the managers that require them are mutually dependent.

The coordination is achieved with bool variables that act as switches. The variables that belong
to one manager are set to true in that manager and set to false in the other manager, when it receives
the message. They inform the receiving manager that the sending manager is waiting for it to complete
a task, or that the sending manager has completed a task the receiving manager was waiting for it to
complete. This task can be to start or to end a given instruction or subinstruction.

40
The waiting variable is used to inform the receiving manager that the sending manager is waiting for it to initiate an instruction. This variable is set to false when the instruction starts, after the user issues the correct command. The completed variable is used to tell the receiving manager that the sending manager completed a subinstruction. This variable is set to false during a wait subinstruction in the receiving manager, after it had been waiting on the variable and a true value finally occurs.

These variables are simple switches that carry no information about which task they refer to. That information is either unnecessary, because there is only one possible task they could be referring to, or necessary and found through other mechanisms that evaluate the current game circumstances.

<table>
<thead>
<tr>
<th>Share</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool waiting</td>
<td>True when this manager is waiting for the other manager to complete a given task.</td>
</tr>
<tr>
<td>bool completed</td>
<td>True when this manager has completed a task the other manager was waiting for it to complete.</td>
</tr>
</tbody>
</table>

Table 3.5: Public variables in the Share class

3.3 The station

3.3.1 The game object

A total of 11 modules of the International Space Station were modeled: the Columbus Research Laboratory (Columbus), the Harmony Node (Node 2), the SpaceX Dragon Docking Vehicle (Dragon), the Japanese Experiment Module (Kibo 1 and Kibo 2), the Destiny Module (US Lab), the Unity Node (Node 1), the Quest Joint Airlock (Airlock), the Orbital Cygnus Docking Vehicle (Cygnus), the Tranquility Node (Node 3), and the Permanent Multipurpose Module (PMM). Not all modules of the ISS were modeled, but the ones that were are all connected to each other, to avoid dividing the station into isolated portions. Only the interior part of the modules was modeled, since the player will always be inside the station. The assets are faithful to reality when it comes to dimensions, proportions, colors, hollows and protuberances, position and orientation of the hatches.

(a) All modules

(b) Modeled modules

Figure 3.4: International Space Station layout: real model

A total of 72 objects were modeled, some structural, like hatches, frames, handrails, platforms, tables, and grids; some that would be movable in reality but are fixed for the purpose of this simulation, like bags, books, flags, laptops, monitors and other equipment; and finally smaller objects that can be grabbed by the player or the robot, like cameras, headphones, flashlights, pliers, wrenches, pens, rolls of tape, food (fruit, cans, bagged solids and fluids, silverware) and toys (rocket models). All objects modeled are copies of real objects seen in reference images (though sometimes stylized and with different colors), but they were not placed in the same spots in which they can be found on those images. The workload of modeling every asset necessary to achieve such an accurate representation would be immense. For this reason, a decision was made to model a reasonable selection of the most common objects and duplicate them around the scene. Nevertheless, the original functionality of each module was observed; for example, storage modules were populated with bags, while work modules have computers.

All modules have Mesh Colliders with the same mesh as the Mesh Renderer. The option of using a simplified mesh was considered, but in that case the player and the robot would be unable to reach all nooks in the modules. Using the original mesh offers more flexibility to navigate inside the station.

Most of the objects that cannot be grabbed have primitive colliders. Only the ones with more complex shapes have simplified Mesh Colliders. This is important for objects that the robot will have to get real close to in order to pick up and drop objects. One example is the table; with a Box Collider, the robot would be stopped from reaching its top surface.

The objects that can be grabbed require more precise colliders, to allow the robot to close its arm tightly around them, without leaving any empty space that would look unrealistic. However, there are dozens of interactable objects in the scene and having them all with mesh colliders at the same time would significantly slow down the game. Since the robot and the player will only be able to interact with a few objects at the same time, a solution can be worked out that fits both situations. These objects have two colliders: a Box Collider and a Mesh Collider with a simplified mesh. The Box Collider is enabled when the object is at its original place in the station and that is enough to prevent the arm of the player or the robot of passing through the object while they are cruising the scene. The Mesh Collider is only enabled when either the player or the robot interact with it. Only a few interactable objects, the ones with shapes that resemble a box, do not have the additional Mesh Collider and are exempt from this switching process.
3.3.2 The main manager

The station main manager is responsible for attending to the commands issued by the user that target the station. There is only one possible command for the station; when the user asks to restart the station, the scene is reloaded, making it like it was when the game started. All changes made to the player, the robot or the objects are erased. Due to the simplicity of the available command, the station main manager uses neither the instructions system nor the state machine described in Section 3.2.

In addition to listening to commands, this manager also has the responsibility of working as a server that holds all the information about the station that other managers need to know during runtime. This information is built using a number of auxiliary classes, which are described below one by one.

An instance of the Location class receives a point in space and finds the location of that point in relation to other location related classes. This class is used mainly by the robot navigation system.

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector3 position</td>
<td>A point in a three-dimensional coordinate system. For a game object, it should be the global position of its origin.</td>
</tr>
<tr>
<td>Module module</td>
<td>The module where the point is located (null if it does not belong to any).</td>
</tr>
<tr>
<td>Line line</td>
<td>The line where the point is located (null if it does not belong to any).</td>
</tr>
<tr>
<td>Point referencePoint</td>
<td>The reference point where the point is located (null if it does not coincide with any).</td>
</tr>
</tbody>
</table>

Table 3.6: Public variables in the Location class from the Station namespace

An instance of the Item class holds all the information that both the player and the robot have to know in order to grab or drop a given item. These calculations are done with auxiliary game objects, inside an enclosed cube at some distance from the main scenery, to guarantee that the modules and objects in the station do not interfere with this process.

The player needs to know the position and rotation the item has when he is holding it, and also how much he should close his fingers to secure it. These three values were defined manually for each type of item, using a shadow arm that is a replica of the player arm. Every time the player needs to grab something, it will copy the way this secondary arm is holding that same item. For example, for the space shuttle toy, it was decided to place it as shown in Figure 3.6, with its nose sticking out so that the robot can put its arms around it even if the player has his hand closed.

Unlike the player, most of the robot information is calculated via code. The shadow robot is placed at the origin of the enclosed cube. Though this enclosure is not at the center of the scene, the local position and rotation of the shadow robot - in relation to the enclosure - are null. The robot is considered to have a reset transform: located at the origin (position zero), with its normal orientation (rotation zero) and not scaled (scale one). Later, all results need to be corrected for the deviation of the enclosure.

There are several variables that need to be defined to eliminate any ambiguity in how the robot grabs an item. The first one is the rotation of the item. The objects are exported from the modeling software with a rotation that aligns their planes of symmetry with the world axes, so the most natural
rotations will be the ones obtained by increments of 90 degrees around a combination of axes. This leaves a pool of 48 possibilities, which can be reduced to 4 when it is settled that the robot has to grab the item by positioning itself above the hand of the player. These are all the possible rotations that make the local axis of the item that is pointing up when the item is held by the astronaut coincide with the vertical axis of the robot. In the case displayed in Figure 3.7, two of them, 3.7b and 3.7d, can be discarded since the length of the space shuttle is greater than the span of the arms. Between the other two, 3.7a was chosen simply because it makes the nose of the space shuttle point in the forward direction of the robot. The rotation of the item is the only variable that is chosen manually. After the item is rotated to the desired value, it can be left in any position around the shadow robot, since everything else will be determined by code.

![Figure 3.7: Possible rotations for the item that is grabbed by the robot](image)

The second variable to be calculated is the position of the item, which involves knowing three values, one for each coordinate. The coordinate system used has its origin at the robot origin and its axes are represented in red, green and blue, for the X, Y and Z axes, respectively.

The first coordinate chosen is along the Z-axis, which is computed using the Box Collider. It is important to use the center of the Box Collider as the reference and not the origin of the item, because usually the origin of the mesh will not be in the same point as its center. The sphere of influence of the robot is the one centered in its origin with the minimum radius necessary to include all parts of the robot and also the item it is grabbing. This sphere is used for some navigation operations and the goal is to minimize it. The best option to do so is to center the item with the robot’s body, as displayed in Figure 3.8a. However, the robot’s arm is at a distance of 4.3 cm from that central plane, which means that it will miss some smaller objects if they are centered with the body, as shown in Figure 3.8c. These objects need to be centered with the arm instead; due to their small dimensions, they do not disturb the sphere radius even when in that position. Each item is centered either with the body or with the arm, depending on their length; the threshold is set at 10 cm.

![Figure 3.8: Possible positions in the Z-axis for the item that is grabbed by the robot](image)
The second coordinate chosen is along the X-axis, which is also computed using the Box Collider. All items are centered with the robot’s body.

The third coordinate chosen is along the Y-axis, which is now computed using the Mesh Collider, because the item needs to get as close to the robot as possible. The first step is to position the item 50 cm below the robot. The items that can be grabbed are all small and this is more than enough to guarantee that no part of the robot is touching them. The goal now is to find exactly how much the robot has to descend to hit the item. Seven boxes were used, one for each major part of the robot, as shown in Figure 3.9. The boxes for the movable parts of the arm are extruded to their maximum so that they do not hit the top of the item. If the item is slim, the arm boxes may never hit it on their way down. If the item is narrow at the top, but its width increases until it is thicker than the arm span at the bottom, the robot can still grab it by holding only its top part, but it needs to know the height at which the item becomes too large to grab, and that is when the arm boxes come in. The other three boxes are certain to hit any item.

![Figure 3.9: Set of colliders used for the robot](image)

These boxes are lowered one by one, using the BoxCast function, until they make contact with the Mesh Collider. The height of the point in the collider that was hit is used to compute the height of the center of the box (adding half the height of the box), and then the position of the robot (adding the deviation from the center of the box to the origin of the robot). In the case of the space shuttle, only the three first boxes hit the item, as shown in Figures 3.10a, 3.10b and 3.10c. The others passed on the sides without touching it.

Another hit point also needs to be found. The item will usually be resting on some kind of surface, be it the palm of the astronaut, the top of a table or a platform in the station walls. A plane that contains the bottom surface of the Box Collider - with the Y-axis as its normal and the same height as the lowest point of the item - is also checked for collisions, but using only the box for one of the small arms (the result is the same for both). The result for the space shuttle can be observed in Figure 3.10d.

From all these hit points, the highest one is chosen and a margin of 2 mm is added. Knowing how much the robot would have to be lowered to hit the item, the item can be elevated to match that position.
At this stage, the item has the position and rotation that it needs to have for a robot at the origin to grab it. This means that the transform of the item in relation to the robot is known; however, the desired one is the transform of the robot in relation to the item. During the game, the item will not be dislocated and oriented to suit the robot; it is the robot that will have to put in that work. To find those values, the robot and the item are translated and rotated as a whole until it is the item that has a position and rotation of zero. The current position and rotation of the robot can now be stored to use later. Of course that in the game the item will probably not have a reset transform, but these values can be adapted to take into account those deviations - that is only one additional and unavoidable computation step to make during runtime, instead of two.

The third and fourth variables to be computed are the radius of each side of the robot’s arm. This is done using the boxes for the small arms mentioned in the previous step. Each box is extruded a bit further than their maximum position so that an item with exactly the maximum width does not go undetected. Each one is then translated in the direction of the other, again using the BoxCast function, until they hit the Mesh Collider. The hit points returned tell us where each small arm should be. These radius values can be different or even negative, which could happen for an item that is not symmetric or that is shaped as a U, for example. Knowing these values, the item position in the X-axis could be corrected so that they were both the same, but then the Y-axis position would be invalid for this new
situation and would have to be computed again. In turn, the new Y-axis position could make each arm radius different from their previous values. This method would get the program stuck in an endless cycle of adjustments, which is why the robot’s arm was designed with the functionality described previously in Section 2.3.2.

![Figure 3.12: Representation of the process used to find the arm radius (start position in green, end position in yellow, path in blue)](image)

The fifth variable is the approach position. Up until now, all variables were related to the contact point with the item: the position, rotation and arm radius of the robot when it is grabbing the item. However, there needs to be an approach point, at some distance from the contact point, where the robot can rotate freely to the orientation it will have when grabbing the item. It cannot rotate when it is already at the contact point, because it would most likely hit the item. The approach point is the closest point to the item where a sphere centered at the origin of the robot and the minimum radius necessary to enclose all its parts makes a tangent to some point in the Mesh Collider, without ever passing through it.

To make this calculation, the robot-item ensemble is translated and rotated back, so that the robot is at the origin and the item is located in relation to it, for the contact point situation. It is established that the approach point is where the robot will rotate to the contact point orientation, which means that the robot will shift from the approach point to the contact point with that same rotation. All the previous work was done with the item as a subordinate to the robot; therefore, while simulating this situation inside the enclosure, the robot will translate from approach to contact with null rotation. Furthermore, since the bottom part of the robot will be in contact with some kind of obstacle and the Y position was computed considering that the robot descended straight down, it can be concluded that the approach point will be somewhere directly above the current robot position, in the form (0, y, 0), with y positive.

The closest point in the Mesh Collider is found using the SphereCast function. The origin of the sphere is at (0, 0.5, 0); 0.5 m is more than enough to make sure that the start sphere does not touch the item. The radius of the sphere is the distance of the farthest point in the robot’s mesh to its origin, adding a margin of 1 cm. This sphere is cast down until it hits the Mesh Collider. The hit point is now known, but not the position of the center of the sphere when it hits that point. If the center of the sphere is at the coordinates \((x_1, y_1, z_1)\) and the hit point at \((x_2, y_2, z_2)\), the distance \(d\) between the two equals...
the radius of the sphere. This equation can be solved using the quadratic formula, which returns two solutions, one above and one below the item. The correct one is the one with the greater value. The solution found can be used to position the item in the approach situation and then repeat the process explained before, to find the robot approach point in relation to the item.

\[
(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2 = d^2
\]

\[
x_2^2 + (y_2 - y_1)^2 + z_2^2 = d^2
\]

\[
x_2^2 + y_1^2 - 2y_1y_2 + y_2^2 + z_2^2 = d^2
\]

\[
y_1^2 - 2y_1y_2 - d^2 + x_2^2 + y_2^2 + z_2^2 = 0
\]

\[ax^2 + bx + c = 0, \text{ with } a = 1, b = -2y_2 \text{ and } c = -d^2 + x_2^2 + y_2^2 + z_2^2, \text{ all known}
\]

Figure 3.13: Representation of the process used to find the approach point (start sphere in green, hit point in blue, solutions in orange, end sphere in yellow)

The sixth variable is the radius of the grabbed item in relation to the robot. As mentioned before, the sphere of influence of the robot should be expanded when it is transporting an item that punctures its original sphere. The radius of an item is calculated in the contact situation, because it is with that position and orientation of the item that the robot will be hauling it from one place to another. It equals the distance from the origin of the robot to the farthest point among the vertices that form the Box Collider of the item. In Figure 3.14a, it is possible to see that the robot sphere is not big enough to contain the item. The spheres calculated for the item are shown in Figures 3.14b and 3.14c (the item is centered so each set of four vertices produces the same sphere). The correct one is in Figure 3.14c.

Figure 3.14: Representation of the process used to find the new radius of the influence sphere (robot sphere in blue, item spheres in green)
Finally, both the player and the robot need to remember where the item was if they want to put it back in its original place. Each type of item can have several instances in the scene. When the game starts, the position, rotation and parent game object is stored for all those instances.

<table>
<thead>
<tr>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>string index</td>
</tr>
<tr>
<td>string name</td>
</tr>
<tr>
<td>Bounds bounds</td>
</tr>
<tr>
<td>float grabbedRadius</td>
</tr>
<tr>
<td>float robotRightArmRadius</td>
</tr>
<tr>
<td>float robotLeftArmRadius</td>
</tr>
<tr>
<td>Vector3 robotContactPosition</td>
</tr>
<tr>
<td>Vector3 robotApproachPosition</td>
</tr>
<tr>
<td>Quaternion robotRotation</td>
</tr>
<tr>
<td>string astronautFingers</td>
</tr>
<tr>
<td>Vector3 astronautPosition</td>
</tr>
<tr>
<td>Quaternion astronautRotation</td>
</tr>
<tr>
<td>Dictionary&lt;Transform, Vector3&gt; originalPositions</td>
</tr>
<tr>
<td>Dictionary&lt;Transform, Quaternion&gt; originalRotations</td>
</tr>
<tr>
<td>Dictionary&lt;Transform, Transform&gt; originalParents</td>
</tr>
</tbody>
</table>

Table 3.7: Public variables in the Item class from the Station namespace

An instance of the Areas class holds information about the regions that make up one given module. An area is always in the shape of a box, positioned and scaled but never rotated, that includes some or all of the module bounds volume. It never crosses the module boundaries; the minimum point of an area is always equal or greater than the minimum point of the module bounds, while the maximum point is always equal or less than the maximum point of the bounds. Each area game object has a box collider marked as a trigger, but no renderer, which means that the user cannot see them. There are four different types of area: zero, one, two and three.

All modules have one and only one type zero area. This area is equal to the module bounds and it is used to display the player’s location on the screen. It interacts with a small sphere collider held by the player’s body, also marked as a trigger. When the player collider enters an area collider, the script attached to the area updates the name of the module printed on the canvas by the location manager.

All modules have one and only one type one area, and some have an additional type two area and/or a type three area. These areas are used to know which modules should be active based on the player’s location. This information is useful to improve performance, because the modules that the player is not seeing at the moment are not rendered, saving up on computation time in each frame. Unity also has
two built-in tools to disable the renderers of objects that are outside of view, named frustum culling and occlusion culling. These options were explored and the results can be consulted in Appendix A.3.

To demonstrate the process of creating areas for all modules, Kibo 1 will be given as an example. When the player is in Kibo 1, he can also see Columbus, Node 2, Dragon and Kibo 2. He cannot see US Lab or Node 1, even if he positions himself at the limit of the connection between Kibo 1 and Node 2. A type one area is created, containing the entire module bounds, and the names of the four modules mentioned previously are assigned to this area. However, the player cannot see all of these modules at the same time. If he is looking at the passage to Node 2, he can always see Node 2 and Columbus, even if he is in the middle or at the back of Kibo 1. This is possible because those two modules are directly aligned with the module the player is in. But the player is not able to see Dragon from all locations inside Kibo 1, even if he is looking at that connection. He needs to be close to the passage to Node 2, so that he can look down and take a peek at the entry of Dragon, because this module is transversely oriented in relation to the module the player is in. It is useless to render Dragon when the player is at the back of Kibo 1. An additional type two area was created to deal with this situation. It has the same height and width as the type one area, but the length is much shorter. Its value is found by approaching the connection, coming from the back of Kibo 1, as close as possible to the top panel and looking down. The first spot where the entry of Dragon becomes visible marks the beginning of area two. This area is centered in a way that makes its exterior plane cut through the middle of the connection between Kibo 1 and Node 2. Area two eats away at one of the extremities of area one, so the initial area one is resized to exclude the volume that area two took over. Finally, Dragon is unassigned from area one and reassigned to area two. This concludes the work for the connection to Node 2, but there is another connection in Kibo 1 that can also be optimized. Just as it happened with Dragon, it is not possible to see Kibo 2 from all locations in Kibo 1. This time, the player has to be closer to the back of the module to see a module that is directly connected to Kibo 1 but is rotated in a way that makes its longitudinal axis perpendicular to the longitudinal axis of Kibo 1. A type three area is created and its length is found by approaching the connection coming from the front of Kibo 1, as close as possible to the bottom panel and looking up. Area one has to be resized again to include only the central part of the module and Kibo 2 is delegated from area one to area three.

To know which modules are visible from each area, it is necessary to add the current module and the modules inherited from the main area. For example, when in area one the player can see Kibo 1 (current module), and Columbus and Node 2 (specific modules). In area two, he can see Kibo 1 (current module), Columbus and Node 2 (inherited modules), and Dragon (specific module).

The concepts illustrated by the previous example can be generalized for the whole station. For a given module, a type one area covers all the volume that is not included by any type two or type three areas. A type two area covers the volume close to a connection when there is a third module connected to the one this passage gives access to, with transverse orientation in relation to both the first and second modules. A type three area covers the volume close to a connection when there is a second module directly connected to this module, but with a transverse orientation. Most type two and three areas are caused by the nodes, because they have a constellation of four hatches distributed in a circle.
Unlike the type zero areas, these ones interact with the sphere collider marked as trigger that is attached to the player’s eyes. What matters is not the location of the player’s body, but what he is seeing. Since the center of the body and the center of the eyes are not in the same position, it is possible to have the body inside one area and the eyes inside another one.

![Area zero and Areas one, two and three](image)

Figure 3.15: Areas for module Kibo 1 (zero in blue, one in red, two in green, three in yellow)

| Dictionary<string, Transform> transforms | A dictionary with all type zero, one, two and three areas, for a given module. |
| Dictionary<string, List<Transform>> visible | A dictionary with all visible modules when in type one, two and three areas, for a given module. These are the modules that should be activated when the player enters the area. |
| Dictionary<string, List<Transform>> invisible | A dictionary with all invisible modules when in type one, two and three areas, for a given module. These are the modules that should be deactivated when the player enters the area. |

Table 3.8: Public variables in the Areas class from the Station namespace

The Point, Line and Paths classes are used to help the robot navigate the station. A few reference points were placed in each module: one in middle of each passage; one at some distance from the passage, when the panels get wider; and one in the middle of the module, but only for the ones that do not have all hatches aligned along a single axis. To connect the points with lines, each point tries to find its closest neighbour in all six semi axes. A continuous network of points and lines covers the entire station, forming the paths. There are no objects obstructing the paths and the robot knows that it can follow them without worrying about collisions with pieces of scenery.

For example, Node 2 has nine reference points: Frontier Node 2 Columbus, Frontier Node 2 Dragon, Frontier Node 2 Kibo 1, Frontier Node 2 US Lab, Interior Node 2 Columbus, Interior Node 2 Dragon, Interior Node 2 Kibo 1, Interior Node 2 US Lab and Interior Node 2 Junction. When the Interior Node 2 Junction point tries to form a line in the positive X semi-axis (which points in the direction of the Columbus module), it searches for all points that have a greater X coordinate and approximately the same coordinates in Y and Z (with a margin of 10 cm because the hatches are not perfectly aligned). It finds the points Frontier Node 2 Columbus and Interior Node 2 Columbus. Of these, the interior point is closer, so a line is formed between the two. Later, when building the lines for the Interior Node 2 Columbus point, the Interior Node 2 Junction point is ignored, because there is already a line connecting them. The paths created for this module are displayed in Figure 3.16.
Figure 3.16: Paths created for module Node 2 (frontier points in red, interior points in yellow, lines in green)

<table>
<thead>
<tr>
<th>Point</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>string name</td>
<td>The complete name of the point (for example, Interior Node 2 Columbus).</td>
</tr>
<tr>
<td>string type</td>
<td>The type name of the point (for example, Interior).</td>
</tr>
<tr>
<td>string module</td>
<td>The module name of the point (for example, Node 2).</td>
</tr>
<tr>
<td>string specific</td>
<td>The specific name of the point (for example, Columbus).</td>
</tr>
<tr>
<td>Transform transform</td>
<td>The transform of the point’s game object.</td>
</tr>
<tr>
<td>List&lt;string&gt; connections</td>
<td>A list with the names of all points this one is directly connected to (for example, Frontier Node 2 Columbus and Interior Node 2 Junction).</td>
</tr>
<tr>
<td>List&lt;List&lt;Line&gt;&gt; lines</td>
<td>A list with all the lines this point belongs to.</td>
</tr>
</tbody>
</table>

Table 3.9: Public variables in the Point class from the Station namespace

<table>
<thead>
<tr>
<th>Line</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>string name</td>
<td>The name of the line (for example, Frontier Node 2 Columbus - Interior Node 2 Columbus).</td>
</tr>
<tr>
<td>Point start</td>
<td>The start point of the line (for example, Frontier Node 2 Columbus).</td>
</tr>
<tr>
<td>Point end</td>
<td>The end point of the line (for example, Interior Node 2 Columbus).</td>
</tr>
</tbody>
</table>

Table 3.10: Public variables in the Line class from the Station namespace

<table>
<thead>
<tr>
<th>Paths</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dictionary&lt;string, Point&gt; points</td>
<td>A dictionary with all the points in a given module.</td>
</tr>
<tr>
<td>Dictionary&lt;string, Line&gt; lines</td>
<td>A dictionary with all the lines in a given module.</td>
</tr>
</tbody>
</table>

Table 3.11: Public variables in the Paths class from the Station namespace

An instance of the Module class holds information about one module in the station.

<table>
<thead>
<tr>
<th>Module</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>string name</td>
<td>The name of the module (for example, Columbus).</td>
</tr>
<tr>
<td>Transform transform</td>
<td>The transform of the main game object that is the parent of all game objects related to this module.</td>
</tr>
<tr>
<td>Bounds bounds</td>
<td>The bounds of the module in world space. They include half of each passage that connects this module to its neighbours.</td>
</tr>
<tr>
<td>List&lt;string&gt; connections</td>
<td>A list with the names of the modules directly connected to this one.</td>
</tr>
<tr>
<td>Areas areas</td>
<td>The areas of this module.</td>
</tr>
<tr>
<td>Paths paths</td>
<td>The paths of this module.</td>
</tr>
</tbody>
</table>

Table 3.12: Public variables in the Module class from the Station namespace

An instance of the Station class is created when the game starts. It holds all the information about...
the station that never changes during runtime and that several scripts need to know. It is made public so that any of them can consult it at any time.

<table>
<thead>
<tr>
<th>Dictionary&lt;&quot;string, Item&quot;&gt; items</th>
<th>A dictionary with all the items in the station, with their names as the key (for example, 137-prefab).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dictionary&lt;&quot;string, Module&quot;&gt; modules</td>
<td>A dictionary with all the modules in the station, with their names as the key (for example, Columbus).</td>
</tr>
</tbody>
</table>

Table 3.13: Public variables in the Station class from the Station namespace

3.4 The canvas

3.4.1 The game object

The canvas is the game object that parents all user interface elements. In this project, the canvas was set to screen space overlay mode, which means that it is rendered on top of everything else in the scene, without a feeling of depth. In this mode, the canvas automatically fills the entire screen, resizing itself if the aspect ratio is changed. The pixel perfect option was also selected, to sharpen the look of the texts written on the screen.

The UI elements can be texts, images, buttons, sliders and others. In this project, three main texts were used, one for the player location (anchored to the upper right corner), one for the commands (anchored to the upper left corner), and another one for the constraints (anchored to the lower left corner). All these elements have the raycast target box unchecked, because they are not interactive buttons that the user can click on.

3.4.2 The location manager

The location manager displays the name of the module where the player is currently located. The content of the text is set by this manager only once, when the game starts. After that, it is updated by the areas described in Section 3.3. During the game, this script controls not what is printed but the printing mode. There are three of them. In the first mode, the text is always hidden. In the second mode, the text shows up for three seconds when the user switches to this mode or when the location changes. In the third mode, the text is always showing. The user can toggle between modes by pressing the F key.

While in mode two, this manager uses a coroutine function to display the text. A regular function runs until it is completed, which means that it occurs within a single frame update. A coroutine function can pause its execution and later pick up where it left off. This type of function is declared with an IEnumerator return type and the statement yield return will appear somewhere in the middle of the code, to mark the place where the function should pause itself. The developer has the option of resuming the execution in the next frame or after a given number of seconds has passed. 44 The coroutine used activates the text, to make it appear on the screen, pauses for three seconds, and then deactivates the text, if the manager is still in the second mode. This verification is done to avoid hiding the text if the


53
user switched modes during the waiting period. Before the coroutine function is called, all coroutines are stopped, to prevent a previous coroutine from overlapping with the next, when the user is quickly switching between modes and/or modules.

### 3.4.3 The commands manager

The commands system is used to receive instruction related input from the user, which is usually more complex than making the player move forward, for example. There are three main menus that can be invoked: the station commands, pressing M, the player commands, pressing P, and the robot commands, pressing R. Then, the user can press a digit to choose an option. If that option requires some additional options, the next submenu pops up, making the user work his way through a tree-structured set of commands. When the user presses the final digit of a particular branch of commands, the menu automatically disappears. If the user has not yet pressed that final digit, it is possible to cancel by pressing again the letter for that menu, which would make the menu disappear, or by pressing the letter for another menu, which would make the current one disappear and the new one appear at its first level. If the user presses an invalid digit (for example, 9 for the player commands menu), or more than one key at the same time (for example, P and 1, or 2 and 3), nothing happens. All managers that listen for keyboard input are prepared to solve disputes when two conflicting keys are pressed at the same time.

1. Restart Station

| 1. Raise Arm. |
| 2. Lower Arm. |
| 3. Show Target. |
| 4. Hide Target. |
| 5. Grab Object. |
| 6. Drop Object. |
| 7. Accept Object. |

<table>
<thead>
<tr>
<th>Table 3.14: Station Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Leave The Docking Station.</td>
</tr>
<tr>
<td>2. Return To The Docking Station.</td>
</tr>
<tr>
<td>3. Grab Object.</td>
</tr>
<tr>
<td>4. Drop Object.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3.15: Player Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. From Player.</td>
</tr>
<tr>
<td>2. From Colleague.</td>
</tr>
<tr>
<td>3. From Station.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3.16: Robot Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. From Colleague Name 1.</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>n. From Colleague Name n.</td>
</tr>
</tbody>
</table>

The SubCommand class represents a node of a menu. Each subcommand has a description seen by the user, which can be changed without worrying about changing also the description used by the program. The visible description can be longer and better phrased. For example, an object that can be grabbed is referred to as an item in the code, because the word object is reserved, but that is the word that the user sees.
Table 3.17: Public variables in the SubCommand class from the Station namespace

The Command class represents a set of 0 to n subcommands, all of which have to belong to the same menu and be organized in the same order in which the user can reach them. A complete command is one that represents an entire branch - in other words, one that contains a leaf subcommand. Only when a command is complete the target for that command (the Station Manager, the Player Manager or the Robot Manager), will try to obey it. A requested command is one that was seen by the correct target manager and is being processed.

Table 3.18: Public variables in the Command class from the Station namespace

The Commands class represents an entire tree for a given menu. Only the first level of subcommands can be reached directly; the others can be found by looking for the children of each one of those initial subcommands.

Table 3.19: Public variables in the Commands class from the Station namespace

The way the commands manager was designed makes it easy to remove or add commands at any time. The necessary code is already in place if the developer later decides to build more complex requests, for example, to ask the robot to grab a certain type of item, like a pen, and then be prompted for that pen color or other preferences.

3.4.4 The constraints manager

The constraints manager is used to receive input from the user related to the player’s axes of translation and rotation. Two different coordinate systems can be used: the world axes (X, Y, Z) and the local axes (x, y, z). The world axes are static, but the local axes depend on the player’s rotation. The x-axis points to the right, the y-axis points up and the z-axis points forward.
The player can move right or left; up or down; forward or backward. For each of these directions, the user can choose whether they refer to the local axis or the global axis. The translation constraints can be any combination of three local or global axes. To switch between states, the user should press X, Y or Z. Some examples of how these keys work can be seen in Table 3.20.

<table>
<thead>
<tr>
<th>(x, y, z)</th>
<th>X</th>
<th>(X, y, z)</th>
<th>Y</th>
<th>(x, y, Z)</th>
<th>Z</th>
<th>(x, y, z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X, , )</td>
<td>X</td>
<td>(X, , )</td>
<td>Y</td>
<td>(X, , )</td>
<td>Y</td>
<td>(X, , )</td>
</tr>
<tr>
<td>others</td>
<td>X</td>
<td>(x, , )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(, y, z)</td>
<td>Shift+X</td>
<td>(, Y, z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(, y, Z)</td>
<td>Shift+X</td>
<td>(, Y, Z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(, , Z)</td>
<td>Shift+X</td>
<td>(, , Z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>others</td>
<td>Shift+X</td>
<td>(, , z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.20: Examples of state updates for the translation constraints

The player can also rotate around all six axes, but while he is able to translate along three different axes at the same time, he can only rotate around two simultaneously, or only one if he chooses so. The reasons for this restriction are explained in Section 3.5.3. The rotation constraints can be any combination of one or two local and global axes. To switch between states, the user should use the keys X, Y, Z and Shift. Some examples of how these keys work can be seen in Table 3.21.

<table>
<thead>
<tr>
<th>(x, , )</th>
<th>X</th>
<th>(X, , )</th>
<th>Y</th>
<th>(X, , )</th>
<th>Z</th>
<th>(X, , )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(, y, )</td>
<td>X</td>
<td>(X, , )</td>
<td>Y</td>
<td>(X, , )</td>
<td>Z</td>
<td>(X, , )</td>
</tr>
<tr>
<td>others</td>
<td>X</td>
<td>(x, , )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(, y, z)</td>
<td>Shift+X</td>
<td>(, Y, z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(, y, Z)</td>
<td>Shift+X</td>
<td>(, Y, Z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(x, , )</td>
<td>Shift+X</td>
<td>(X, , )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(x, , )</td>
<td>Shift+X</td>
<td>(X, , )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(x, , )</td>
<td>Shift+X</td>
<td>(X, , )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>others</td>
<td>Shift+X</td>
<td>(x, y, z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.21: Examples of state updates for the rotation constraints

Both the translation and rotation constraints use some of the same keys, so the user should press Tab to toggle between the two and lock the one he wishes to update. It should be noted that changing the constraints does not cause any movement in the player; it only indicates how he would move if the user asked him to.

By default, the constraints are (x, Y, z) for translation and (x, y, ) for rotation. This makes the player always move up or down and turn around considering the upwards direction of the station, instead of the direction the top of his head is currently pointing to. This is useful to avoid disorientation. Any combination that mixes the x, y, z and Y axes offers believable results. Using the X or Z axes can be confusing and it is not recommended, though those options were maintained for a question of consistency.
3.5 The player

3.5.1 The game object

The hierarchy of the player game object can be consulted in Appendix A.2. It has three major elements: the body, the arm and the eyes. Both the arm and the eyes are children of the body, to make sure they always have the same offset even when the player moves and rotates. The mesh of the arm is weighted to the armature, so it will deform when the bones move, even without having a parent-child relationship. The armature includes all bones created and also two additional game objects: the target and the touch point. Both are used to grab objects from the station and need to have always the same original offset in relation to the palm of the hand. This means that they have to be parented to the arm bone. Now, even if the arm is lowered or raised, or if the hand turns up or down, these two objects will be in the correct place, since all those animations are controlled by the arm bone. They are not affected by the animations of the finger bones, as it should be.

The body game object contains a Rigidbody Component, a Capsule Collider Component and a Sphere Collider Component. The rigidbody has both drag values set to infinity, to guarantee it immediately stops moving and rotating as soon as the user releases the input keys. Using other values would steal some of the control the user has over the player. To get the player to a certain position, he would have to stop pressing the keys before reaching that spot and wait for the player to decelerate until coming to a full stop, hopefully in the intended place. The Use Gravity box was unchecked and so was the Is Kinematic box, to allow the player to respond to collisions. The capsule collider, with a height of 30 cm and a radius of 12 cm, is considerably smaller than the body of any astronaut. For a character walking around a scene with gravity, the collider needs to include the soles of his feet, to prevent any part of his body from falling through the ground. Applying that same logic to this game, the player collider would need to be 1.80 m tall, to include his whole body. The modules have a height of about 2.30 m, so the player would have a margin of 0.5 m to go up and down. He would be able to get very close to the top panel, discounting only the thickness of his helmet, but he would not be able to hover down by the bottom panel, because the length of his legs would stop him. This is not how an astronaut feels in the space station; he experiences much more mobility than a person walking on Earth. Another problem comes up when the player tries to go through one the hatches, which are only 1.1 m tall. These difficulties could be overcome if the user first rotated the body of the player, to make it parallel to the ground, then rolled the eyes up to make the player look forward, and finally crossed the passage head first. But this solution is not obvious and places a tiresome burden on the user. The best option is to make a collider that includes only the head and the neck of the astronaut. Leaving out the torso and the legs is the same as saying the player automatically positions them in a way that allows him to obey the user instructions: for example, if the user wants to go down or cross a passage, the legs may be stretched behind the player. Finally, the sphere collider is used as a trigger, that interacts with the station areas to make known the player’s location, as described previously in Section 3.3.2.

The arm game object contains an Animation Component. This component has eleven slots, one for each animation clip made previously in Blender, which can be played at any time. An additional slot
will be created and deleted during runtime, when it is necessary to create an animation clip to raise or lower the arm bone.

The bones game objects have Box Collider Components, Sphere Collider Components and Capsule Collider Components. They have several colliders in order to achieve high precision, allowing the arm to get closer to pieces of scenery and to the robot. The arm has five capsule colliders and the hand has five box colliders. Each finger bone has one capsule collider for the phalanx and one sphere collider for the knuckle. The colliders have to be applied to the bones in order to make them move and rotate with those bones. If the colliders were on the mesh game object and the player closed his fingers, the colliders would not accompany that movement; they would still be stretched open, instead of making a fist. With this solution, the colliders are always synchronized with any animation, as shown in Figure 3.18e.

The eyes game object contains a Camera Component, a Post Process Layer Component and a Sphere Collider Component. The camera is responsible for rendering the scene. When it is attached to the player and positioned at the height his eyes would be, the user sees the world as the player would see it, in what is called a first-person perspective. In the settings, the camera projection was set to perspective, with a field of view of 70 degrees and clipping planes of 0.1 m and 25 m. The post process layer is used to add filters and effects to the image rendered by the camera before it is displayed to the user. Some of the effects added were: anti-aliasing, which smooths jagged lines by surrounding them with an adequate color gradient; ambient occlusion, which darkens corners that should not receive as much ambient light as exposed surfaces; motion blur, to simulate the blurring of the image seen by the eyes when they move too fast; and bloom, which increases the brightness of the lights. Finally, the sphere collider is not used for collisions, since the body collider already includes the eyes, but as a trigger to tell the station which modules should be active based on where the eyes of the player are located, as described previously in Section 3.3.2.


3.5.2 The main manager

The player main manager is responsible for attending to the commands issued by the user that target the player. It uses both the instructions system and the state machine generically described in Section 3.2 in order to process a command. It also uses the share system to communicate with the robot main manager. However, it does not use the orders system, because it does not communicate with the other player sub-managers, since none of the commands require any action from them. The player sub-managers take orders directly from the user based on other input distinct from the commands. The player main manager controls all player motions that come from animations and leaves to the sub-managers the control of the motions that come from scripting.

This manager has six different kinds of subinstructions.

The animation subinstruction is used to play one of the player’s animations. It is defined by the name of the animation and how it should be played: forward or backward. If the animation requested is to raise or lower the arm bone, first that animation clip needs to be created and only then can it be played. This animation leaves the fingers and the hand with their current local rotation and changes only the arm. For example, if the player lowers his arm while holding an object in his hand, the fingers will still be closed around the object as the arm rotates down. This subinstruction is complete when the animation reaches its end.

The item subinstruction is used to move an item from the station to the player’s hand, or from the hand to the station. It requires the translation speed with which the item will cover the distance between its initial and final position and the rotation speed with which it will cover the angle between its initial and final orientations. These are computed so that the item finishes translating and rotating at the same time. During this subinstruction, the item moves and rotates a bit closer to its target each frame, until finally reaching it and completing the subinstruction.

The target subinstruction is used to activate and deactivate the target. It is complete one frame after it starts.

The arm subinstruction is used to activate and deactivate the arm. When the arm is lowered, it gets out of the field of vision of the player and is therefore disabled. Before being raised, it needs to be enabled again. This subinstruction can also be used to reset any rotations added to the arm by the Player Arm Manager. If the user rotated the arm using the keys controlled by that manager, and then tries to lower or raise the arm using an animation, the results will not be the expected ones. The arm is brought
back to its original rotation along several frames, to make that movement smooth and natural. If this subinstruction is only to activate/deactivate the arm, it is complete one frame after it starts; otherwise, it is complete when the arm reaches its reset rotation.

The wait subinstruction is used to wait for a certain amount of time or until the robot has finished a given subinstruction. It is complete when the timer goes off, in the first case, or when the robot signals that it completed the relevant task, in the second case.

The share subinstruction is used to tell the robot that the player is waiting for it to complete a task or that the player has completed a task himself that the robot was waiting for him to complete. To complete this subinstruction, the player has only to turn on the correct switch variable, which means the subinstruction is complete one frame after it starts.

<table>
<thead>
<tr>
<th>SubInstruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>int index</td>
</tr>
<tr>
<td>bool isAnimation</td>
</tr>
<tr>
<td>string animationName</td>
</tr>
<tr>
<td>bool playForward</td>
</tr>
<tr>
<td>bool isItem</td>
</tr>
<tr>
<td>Vector3 itemPosition</td>
</tr>
<tr>
<td>Quaternion itemRotation</td>
</tr>
<tr>
<td>float itemTranslationSpeed</td>
</tr>
<tr>
<td>float itemRotationSpeed</td>
</tr>
<tr>
<td>string itemFingers</td>
</tr>
<tr>
<td>LayerMask itemLayer</td>
</tr>
<tr>
<td>Transform itemParent</td>
</tr>
<tr>
<td>Transform itemTransform</td>
</tr>
<tr>
<td>bool isTarget</td>
</tr>
<tr>
<td>bool targetActive</td>
</tr>
<tr>
<td>bool isArm</td>
</tr>
<tr>
<td>bool armActive</td>
</tr>
<tr>
<td>bool resetArmRotation</td>
</tr>
<tr>
<td>bool isWait</td>
</tr>
<tr>
<td>bool waitForRobot</td>
</tr>
<tr>
<td>bool waitForSeconds</td>
</tr>
<tr>
<td>float targetWaitTime</td>
</tr>
<tr>
<td>bool isShare</td>
</tr>
<tr>
<td>bool waitingForRobot</td>
</tr>
<tr>
<td>bool completedForRobot</td>
</tr>
</tbody>
</table>

Table 3.22: Public variables in the SubInstruction class from the PlayerMain namespace

These subinstructions can be combined in several ways to build complex instructions. The manager currently recognizes eight different kinds of instructions, one for each subcommand at the base level of the player commands menu.

The raise arm instruction is used to rotate the arm 90 degrees, from its lowered position by the side of the body to its raised position stretched in front of the player and pointing forward. The hand will come up empty or holding an object, depending on whether it already had an object when it was lowered. This instruction is valid if the arm is currently lowered. It has two subinstructions. The first is
an arm subinstruction, to activate the arm game object. The second is an animation subinstruction, to play forward the arm animation.

The lower arm instruction is used to rotate the arm 90 degrees, following the inverse path of the previous instruction. This instruction is valid if the arm is raised. It has three subinstructions. The first is an arm subinstruction, to bring the arm back to its reset rotation, pointing forward in front of the body of the player. The second is an animation subinstruction, to play backward the arm animation. The third is an arm subinstruction, to deactivate the arm.

The show target instruction is used to turn on the target used to grab objects. This instruction is valid if the arm is raised and the hand is facing down. Checking if the hand is facing down is the same as checking if the target is not already visible and if the hand is not holding an object, since the palm of the hand will be facing up in both of those situations and facing down otherwise. This instruction has two subinstructions. The first is an animation subinstruction, to play forward the hand animation, meaning that the hand starts with the palm facing down and relaxed fingers, and ends with the palm facing up and stretched fingers. The second is a target subinstruction, to activate the target.

The hide target instruction is used to turn off the target. This instruction is valid if the target is currently visible. It has two subinstructions. The first is a target subinstruction, to deactivate the target. The second is an animation subinstruction, to play backward the hand animation.

The grab item instruction is used to grab an object from the station, which can be located on a table, on a wall platform or in any other place. When the user grabs an object, it flies from where it is resting to the palm of the hand of the player. The way the player grabs an item was subject to some modifications during this project. The first version had the item simply disappear and reappear on the player’s hand. This last version is an improvement when compared to the first one, but still admittedly far from realistic. The main difficulties found came from the fact that it would be hard to position the palm of the hand touching the object, especially if it were located in a wall platform, due to the lack of control at the elbow. Since the player is incapable of bending the arm, he would have to stretch the arm above his head, with the palm facing forward, and then advance to the wall aiming for the object. Other problems would come up if the user grabbed the object with an orientation that does not suit the robot. Considering all this, the decision was to settle for the method described previously. This instruction is valid if the target is visible and the target ray is hitting an object. It has three subinstructions. The first is a target subinstruction, to deactivate the target. The second is an item subinstruction, to transport the item from the station to the player. The third is an animation subinstruction, to close the fingers with the angle required by this object.

The drop item instruction is used to drop an object back to its original place in the station. It is valid if the arm is raised, if the player is holding an object and if that object is within dropping distance. Checking that is arm is raised is necessary, because the player could be holding an item with the arm lowered and inactive, which is not a valid position to drop an object. To be within dropping distance means that the object is less than 1.5 m away from its destination. This has to be checked to prevent the user from trying to drop an item too far from its original place, which would make an already unrealistic method even worse. Without this restraint, the player could even drop an object from a different module,
which would have the item flying around the station. This instruction has three subinstructions. The first in an animation subinstruction, to open the fingers completely, leaving them stretched and the palm of the hand facing up. The second is an item subinstruction, to transport the object from the player to the station. The third is an animation subinstruction, to turn the palm of the hand down and relax the fingers.

The accept item instruction is used to receive an object from the robot. It is valid if the player is not holding an object already and if the robot has an item and is waiting to put it on the player’s hand. This is the same as saying that the user previously gave a command to the robot, ordering it to hand over to the player the item the robot is carrying. It has six subinstructions. The first is an animation subinstruction, to turn the palm of the hand up and stretch the fingers. The second is a share subinstruction, to tell the robot that the player has completed the task (opening the hand) the robot was waiting for. The third is a wait subinstruction, to wait for the robot to put the object on the player’s hand. The fourth is also a wait subinstruction, to wait half a second after the robot puts down the object, so that the next movement does not seem abrupt. The fifth is an animation subinstruction, to close the fingers around the object. The sixth is a share subinstruction, to inform the robot that the player has secured the item and that the robot can now open its arm and retreat away from the player.

The release item instruction is used to deliver an object to the robot. It is valid if the player is holding an object and the robot is waiting for the player to hand it over, with its arms already around the object. This is the same as saying that the user previously gave a command to the robot, ordering it to come get an item from the player. It has five subinstructions. The first is an animation subinstruction, to open the fingers completely. The second is a share subinstruction, to tell the robot that the player has completed the task (opening the fingers) the robot was waiting for. The third is a wait subinstruction, to wait for the robot to retreat away from the hand with the item. The fourth is also a wait subinstruction, to wait half a second before executing the next movement. The fifth is an animation subinstruction, to turn the palm of the hand down and relax the fingers.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>string name</th>
<th>The name of the instruction: Raise Arm, Lower Arm, Show Target, Hide Target, Grab Item, Drop Item, Accept Item or Release Item.</th>
</tr>
</thead>
</table>

Table 3.23: Public variables in the Instruction class from the PlayerMain namespace

The state machine used by the player main manager is the basic one, described in Section 3.2, with no additional situations.

<table>
<thead>
<tr>
<th>State</th>
<th>int situation</th>
<th>The situation number: from one to five.</th>
</tr>
</thead>
</table>

Table 3.24: Public variables in the State class from the PlayerMain namespace

Before the user can grab something, first he has to use the target to lock the desired item. The target is a small object in the shape of an aiming sight, as shown in Figure 3.20a, that appears slightly above the palm of the hand of the player. It searches for objects using an invisible ray that starts in the eyes of the player and passes through the center of the target. When the target is active, the ray is continually updated, using the current positions of the eyes and the target. While the ray is not detecting
an item, the target is yellow, as displayed in Figure 3.20b. If it hits an object that can be grabbed, the target turns red and a box appears around the locked item, rotated and scaled to neatly embrace it, as shown in Figure 3.20c. The box is also red, but semi-transparent, so that the user can still see the object underneath it. If the user moves on from that item, without grabbing it, the box disappears and the target turns yellow again. If the user grabs the item, both the box and the target disappear.

The target can also be used to ask the robot to pick up an item. With the object locked, the user should give a command to the robot to grab an object from the station. As soon as the command is issued, he can lower his arm and wait for the robot to comply with the request.

![Target object](image1)

(a) Target object  
![Target unlocked](image2)

(b) Target unlocked  
![Target locked](image3)

(c) Target locked

Figure 3.20: Target used to search for an object to grab

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string state</td>
<td>The current target state, which can be Hidden, Yellow or Red.</td>
</tr>
<tr>
<td>Transform targetTransform</td>
<td>The transform of the target game object.</td>
</tr>
<tr>
<td>Dictionary&lt;string, Color&gt;</td>
<td>A dictionary that makes the correspondence between the state and the color of the target.</td>
</tr>
<tr>
<td>Ray ray</td>
<td>Ray that starts at the player’s eyes and passes through the center of the target.</td>
</tr>
<tr>
<td>float rayLength</td>
<td>The maximum length of the ray, starting at the eyes. Set to 1.5 m.</td>
</tr>
<tr>
<td>LayerMask rayLayerMask</td>
<td>The layer mask used to search only for objects that can be grabbed. An object that is small enough to be grabbed but is currently being held by the robot does not get included in this mask. This prevents the user from trying to snatch an object from the robot.</td>
</tr>
<tr>
<td>bool rayHitSomething</td>
<td>True if the ray hit an object that can be grabbed in the current frame.</td>
</tr>
<tr>
<td>RaycastHit raycastHit</td>
<td>Information about the game object hit by the ray.</td>
</tr>
<tr>
<td>Transform eyesTransform</td>
<td>The transform of the eyes game object.</td>
</tr>
<tr>
<td>Transform box</td>
<td>The transform of the box used to highlight a locked object.</td>
</tr>
</tbody>
</table>

Table 3.25: Public variables in the Search class from the PlayerMain namespace

There are some instructions that require coordination between the robot and the player, in order to successfully complete a pair of complementary commands. The first case is when the user commands the robot to give an item to the player. Later, the user also has to command the player to accept that item, otherwise the robot command issued previously cannot be completed. The second case is when the user commands the robot to pick up an item from the player, which requires that the user also commands the player to release that item, in order for the robot to be able to follow through with its command. In order for these instructions to work correctly, there needs to be some form of communication between the player manager and the robot manager. The player manager needs to evaluate if an accept item or release item command from the user is valid, which requires knowing if the robot is currently following a drop item on the player or grab item from the player command, respectively. While two complementary commands
are running in each manager, additional coordination is required between some of their phases. Its job is
to guarantee that the robot does not try to put an object on the player’s hand while the palm is still
facing down or that the robot does not try to pry an object from the player’s hand while he still has his
fingers closed.

| Share |
|--------|---------------------------------|
| bool waitingForRobot | True when the player is waiting for the robot to complete a task. |
| bool completedForRobot | True when the player has completed a task the robot was waiting for him to complete. |

Table 3.26: Public variables in the Share class from the PlayerMain namespace

3.5.3 The body manager

The player body manager controls the position and rotation of the body of the player. The player
can move and rotate in any direction, without restrictions, just like an astronaut in the space station
would.

The user can control the translation using the following keys: **A** to go left and **D** to go right (X-axis); **Q** to go down and **E** to go up (Y-axis); **W** to go forward and **S** to go back (Z-axis). He can press
more than one key at the same time to move in a diagonal, but if he presses two conflicting keys the
player will not move in that axis.

The user can control the rotation using the mouse: **Mouse Y** if the player is rotating only in the
X-axis; **Mouse X** if the player is rotating only in the Y-axis; **Mouse X** if the player is rotating only
the Z-axis; **Mouse Y** and **Mouse X** if the player is rotating in the X and Y axes, respectively; **Mouse
Y** and **Mouse X** if the player is rotating in the X and Z axes, respectively; **Mouse Y** and **Mouse X**
if the player is rotating in the Y and Z axes, respectively. These values, **Mouse X** and **Mouse Y**, are
the deltas of the mouse in the horizontal and vertical axes, respectively. They do not depend on the
frame rate, which means that the rotation does not need to be adjusted for the time that passes between
rendering two frames. Using the mouse to manage the rotation allows for better control than when using
the keyboard. The angle of rotation will depend on how much the user moved the mouse in this frame.
The downside is that the player can only rotate in two axes at the same time.

If the user wishes to revert the rotation added to the body in the X and Z axes, to align the torso
of the player with the vertical direction of the station and make his head point up, he can press the **B**
key continuously. This will cause the body to rotate smoothly back to zero in those axes. If the user
releases the key before the body reaches its target, it will stop stabilizing. While the body is stabilizing,
the mouse is blocked. This stabilizing feature is useful if the user finds itself in an awkward rotation,
upside down or sideways, and wants to quickly return to a more natural orientation. It works on the
local axes of the player, even if the constraints are set for the global ones.

The axes of translation and rotation are calculated based on the current state of the constraints and
the current orientation of the player’s body. The keys pressed control the amounts added or subtracted
along each of these axes, in order to build the target. The target is the position and rotation the player’s
body will try to reach in the next frame. He may end up with different values, if these ones make him
collide with another object. To find the position, the deltas in all three axes are added to get a direction
This vector is normalized, so that the player does not get an advantage by moving in more than one axis at the same time.

3.5.4 The arm manager

The player arm manager controls the rotation of the arm of the player. It adds more flexibility to the orientation of the arm, building on top of the functionalities already offered by the animations.

Using animations, the user can raise or lower the arm and turn the palm of the hand up or down, as shown in Figure 3.21. The body collider and the arm local axes are also shown as a reference; the X-axis is red and points to the right, the Y-axis is green and points up and the Z-axis is blue and points forward. The rotations in the animations are accomplished with the arm bone game object, while the rotations coming from this manager are applied to the arm main game object. They have to be applied to different objects to guarantee that the bones remain with the same local space, otherwise the animations would not work correctly. The animations are used to achieve something else other than solely moving the arm. For example, lowering the arm with an animation is used to disable the collider and turning up the hand is used to make the target appear.

![Figure 3.21: Moving the arm using animations](image)

When the user just wants to move the arm, he should use the direct keyboard keys, instead of the commands: I and K for rotating around the local X-axis, the first for raising the arm and the second for lowering; J and L for rotating around the local Y-axis, the first to the left and the second to the right; U and O for rotating around the local Z-axis, the first counterclockwise and the second clockwise. Figure 3.22 shows the limits for the rotation around each axis while keeping null the rotation in the other two. Unlike the animations, that go from one state to another and cannot be interrupted, the orientations shown in this figure are just examples. They can be combined to position the hand in numerous ways along the surface of the sphere centered at the shoulder.

If the user wishes to revert the rotation added to the arm, he can press the V key continuously. This will cause the arm to rotate smoothly back to its base orientation, which depends on the last animation played. If the user releases the key before the arm reaches its target, it will stop stabilizing. While the arm is stabilizing, the other arm keys are blocked.
(a) Minimum rotation in X (-90°)  (b) Maximum rotation in X (90°)  (c) Minimum rotation in Y (-40°)

(d) Maximum rotation in Y (100°)  (e) Minimum rotation in Z (-180°)  (f) Maximum rotation in Z (90°)

Figure 3.22: Moving the arm using scripting

These rotations have to be done along the valid path for each axis. For example, the user is not allowed to rotate the arm from Figure 3.22f to Figure 3.22e taking the shortest route, because a real arm cannot bend that way. These paths are also enforced when the program automatically stabilizes the arm.

Figure 3.23: Allowed rotation paths for the arm

The limit values indicated in Figure 3.22 concern the arm in the situation shown in Figure 3.21b, when the last animation played left the arm raised and the hand down. However, these values need to be corrected to consider other animations that involve the arm bone, which means that the limits have to be different so that the paths remain the same. The situation displayed in Figure 3.21a is not considered, because the arm is inactive and cannot rotate, but the one shown in Figure 3.21c requires an adjustment of 180 degrees in the Z-axis.

3.5.5 The eyes manager

The player eyes manager controls the rotation of the eyes of the player. It allows the player to look around to inspect the station only by moving his eyes, without having to rotate his whole body.

He can look up and down, by pressing the Up Arrow key and the Down Arrow key; to the left and to right, by pressing the Left Arrow key and the Right Arrow key. He can change the rotation of his eyes on both axes at the same time. For example, if the user wants to look to the upper right corner
of his field of vision, he can press the Up Arrow key and the Right Arrow key at the same time. If the user presses both the Up Arrow key and the Down Arrow key, or the Right Arrow key and the Left Arrow key, nothing happens. The rotation of the eyes is clamped to 30 degrees in all directions.

If the user wishes to revert the rotation added to the eyes, making the player look straight forward again, he can press the C key continuously. This will cause the eyes to rotate smoothly back to their original orientation. If the user releases the key before the eyes reach their target, they will stop stabilizing. While the eyes are stabilizing, the other eye keys are blocked.

3.6 The robot

3.6.1 The game object

The hierarchy of the robot game object can be consulted in Appendix A.2. It has three major elements: the body, the arm and the propellers. All components that move with some degree of freedom from the others have their own game object. With the exception of the body, no component is completely independent of the rest of the robot. Both the arm and the propellers were parented to the body, because they need to move along with it when the robot translates or rotates. Each part of the arm also has its place in the hierarchy. Some additional game objects were created, to work as reference rotation points, and placed on the same level as the component they refer to.

The body game object contains a Rigidbody Component. The settings used are the same ones that were described for the player rigidbody, with one exception. The robot rigidbody was marked as kinematic, to prevent the player from plucking the robot out of its docking station, or throwing it around the modules while the robot is obeying an instruction.

The body game object also contains three Mesh Collider Components. Three options were considered when creating the collider. The first option was to use the same boxes that the robot uses to navigate the scene and check for collisions, displayed in Figure 3.24a. The robot will never move or rotate in a way that causes another object to enter the space defined by those boxes. However, the player can approach the robot and touch it with his arm; it would be unrealistic if he were stopped by invisible corners sticking out of the visible mesh. The second option was to use a single mesh collider with three cylinders that resemble more closely the general shape of the body. Unlike the player, the robot has no animations, which means that the mesh does not deform. The problems that would be faced when trying to use a mesh collider for the player are not found in this case. However, the robot is also a rigidbody, which requires any mesh collider to be marked as convex. While the original collider looks like the one displayed in Figure 3.24c, the convex collider takes up more space, as shown in Figure 3.24b. To avoid this, the original mesh was divided into three separate cylinders. These are already convex shapes, therefore they suffer no changes when marked as convex, as displayed in Figure 3.24c.

Each arm game object has a Box Collider Component. All parts of the arm are approximately shaped as cuboids and a single box collider is enough to convincingly represent the occupied volume.
An item game object will sometimes be added to the robot hierarchy, parented to the body, when the robot is carrying that object.

### 3.6.2 The main manager

The robot main manager is responsible for attending to the commands issued by the user that target the robot. It uses both the instructions system and the state machine generically described in Section 3.2, in order to process a command. It also needs the share system to communicate with the player main manager and the orders system to communicate with the robot sub-managers. This manager distributes work to the sub-managers and keeps them all coordinated with each other. The user does not interact directly with any of the robot sub-managers. He can only communicate with the main manager, which then decides if it should contact the sub-managers, and when it should do it. This difference in behavior between the player managers and the robot managers reflects the greater control the user has over the player, while the robot is mainly controlled by its own internal programming.

This manager has six different kinds of subinstructions. Three of these subinstructions - the ones that move the components - are used to give orders to the sub-managers and the other three - the ones that handle the item and coordinate with the player - are solved by this manager itself.

The body subinstruction is used to give an order to the robot body manager. The main manager can ask the body to translate or to rotate. This subinstruction is defined by the value wanted for the position or the rotation, by the layer mask that can be used to selectively ignore colliders when checking for collisions and by the transform of the item related to the current instruction. After giving the order when the subinstruction starts, the main manager needs to wait for a reply from the body manager. The subinstruction is completed only when the sub-manager informs that the order received is either finished or was denied.

The arm subinstruction is used to give an order to the robot arm manager. The main manager can ask the arm to open, close or grab. The arm manager does not need any additional information to open or close, because those positions and rotations are predefined, but it needs to know the position required for the arm when grabbing an object. The subinstruction is completed on the same conditions stated for the body subinstruction.

The propellers subinstruction is used to give an order to the robot propellers manager. The main
manager can only ask the propellers to change their rotation speed and it has to inform the sub-manager of the new desired value. The subinstruction is completed on the same conditions stated for the body subinstruction.

The item subinstruction can be used with two different goals. First, it can be used to switch the item’s layer and parent. This has to be done when the control of the object changes as it circulates between the station, the player and the robot. Secondly, it can be used to enable or disable the object’s collider. Most of the items have two colliders, a box collider and a mesh collider, and only one of them should be active at any given moment. The box collider can be active when the object is at its original place in the station, but the mesh collider should be switched on if the player or the robot start interacting with it. The problem with this approach is that both the player and the robot are rigidbodies. When they interact with an object, that object is included in the game object hierarchies of one of them, depending on which one is holding it. A rigidbody cannot have a non-convex mesh collider assigned to itself or any of its children. On the other side, the mesh collider of the object cannot be marked as convex, because many of its details would be lost and the robot would not be able to completely close its arms around it. The solution found was to always leave the mesh collider disabled, and create a temporary clone of the object, outside of the rigidbodies hierarchies, with the mesh collider enabled. The auxiliary item is used to check for collisions before the robot moves and its creation falls under the responsibility of the sub-managers. For example, when the robot checks if it can move to the contact point with the object, the body manager will ignore the box collider and consider only the mesh collider. The answer would usually be negative for the box collider, but it is positive for the mesh collider. When the robot closes in, it will violate the space of the box collider, but not the one of the mesh collider, which means that the user will not notice this transgression. The sub-managers steer the robot towards the position that would otherwise be unreachable, but it is the main manager that has to deal with the consequences of invading the space of another collider. That fallout is only registered if one of the colliders has a non kinematic rigidbody, entirely controlled by the physics engine. The robot is kinematic, which means its collider offers no resistance when trying to break the surface of another collider. As for the object, it depends on the command issued by the user. The robot can grab/drop an item from/on the player, one of the colleagues or the station. There is no problem when the object is parented to one of the colleagues or to the station, because neither of them is a rigidbody. The same cannot be said of the player. If the robot is grabbing an object from the player and attempts to breach the box collider, it will push away both the object and the arm of the player, in its effort to reach the target position. Though it will succeed, it will also move the object in the process, which means that there is a new contact position, different from the original one. When the robot tries to reach the updated contact point, what happened before happens again, in an endless cycle. A similar problem would occur if the robot was trying to drop an object on the player’s hand. The target position for the object is close to the mesh, so that the user does not notice any empty space, but that position usually makes the colliders overlap, since they are only rough approximations. In this situation, the object would repeatedly bump into the hand, always pushing it away, without ever managing to get close enough. To prevent this, the box collider is disabled for the duration of a robot instruction that involves the player. Since all preventive collision checks are made
with the mesh collider, the robot never overlaps with the mesh of the object. The item subinstruction is always completed one frame after it starts.

The wait and share subinstructions are used with the same goals described previously for the player subinstructions with the same name.

<table>
<thead>
<tr>
<th>SubInstruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int index</td>
<td>The index of the subinstruction in relation to the set that forms an instruction.</td>
</tr>
<tr>
<td>bool isBody</td>
<td>True for a body subinstruction.</td>
</tr>
<tr>
<td>string bodyInstruction</td>
<td>The name of the body instruction (Translate or Rotate).</td>
</tr>
<tr>
<td>Vector3 bodyWanted</td>
<td>The desired position or rotation for the body of the robot.</td>
</tr>
<tr>
<td>LayerMask bodyLayerMask</td>
<td>The layer mask used to selectively ignore colliders when testing for collisions.</td>
</tr>
<tr>
<td>Transform bodyItemTransform</td>
<td>The transform of the item involved in this command.</td>
</tr>
<tr>
<td>bool isArm</td>
<td>True for an arm subinstruction.</td>
</tr>
<tr>
<td>string armInstruction</td>
<td>The name of the arm instruction (Open, Close or Grab).</td>
</tr>
<tr>
<td>float armRightRadius</td>
<td>The desired radius for the right arm of the robot in a grab instruction.</td>
</tr>
<tr>
<td>float armLeftRadius</td>
<td>The desired radius for the left arm of the robot in a grab instruction.</td>
</tr>
<tr>
<td>bool isPropellers</td>
<td>True for a propellers subinstruction.</td>
</tr>
<tr>
<td>string propellersInstruction</td>
<td>The name of the propellers instruction (Change).</td>
</tr>
<tr>
<td>float propellersSpeed</td>
<td>The desired rotation speed for the propellers.</td>
</tr>
<tr>
<td>bool isItem</td>
<td>True for an item subinstruction.</td>
</tr>
<tr>
<td>bool switchItemCollider</td>
<td>True if the subinstruction is used to switch the item’s collider.</td>
</tr>
<tr>
<td>string newColliderStatus</td>
<td>The new collider status, either enabled or disabled.</td>
</tr>
<tr>
<td>bool switchItemLayerAndParent</td>
<td>True if the subinstruction is used to switch the item’s layer and parent.</td>
</tr>
<tr>
<td>LayerMask newItemLayer</td>
<td>The new layer (player, robot or station).</td>
</tr>
<tr>
<td>Transform newItemParent</td>
<td>The new parent (player, robot or station).</td>
</tr>
<tr>
<td>bool isWait</td>
<td>True for a wait subinstruction.</td>
</tr>
<tr>
<td>bool waitForPlayer</td>
<td>True if the robot should wait for the player to complete a task.</td>
</tr>
<tr>
<td>bool waitForSeconds</td>
<td>True if the robot should wait for a given number of seconds.</td>
</tr>
<tr>
<td>float targetWaitTime</td>
<td>The time the robot should wait, in seconds.</td>
</tr>
<tr>
<td>bool isShare</td>
<td>True for a share subinstruction.</td>
</tr>
<tr>
<td>bool shareWaitingForPlayer</td>
<td>True if the robot should turn on this switch, meaning that it is waiting for the player to complete a task.</td>
</tr>
<tr>
<td>bool shareCompletedForPlayer</td>
<td>True if the robot should turn on this switch, meaning that it has completed a task the player was waiting for it to complete.</td>
</tr>
</tbody>
</table>

Table 3.27: Public variables in the SubInstruction class from the RobotMain namespace

The manager recognizes four different kinds of instructions, one for each subcommand at the base level of the robot commands menu. When a command has more than one subcommand, two instructions of the same kind can differ from each other, based on those additional options.

The leave docking station instruction is valid only if the robot is currently at the docking station. This instruction has four subinstructions. The first is a propellers subinstruction, to make the propellers start rotating. The second is a body subinstruction, to translate the robot to the departure position of the docking station. The third is a body subinstruction, to rotate the robot to a clean orientation, since it will not be parallel to the bottom panels if the docking station is fixed to any panels other than those. The fourth is an arm subinstruction, to open the arm.

The return to docking station instruction is valid only if the robot is not at the docking station and is not holding an object. This instruction has five subinstructions. The first is a body subinstruction, to translate the robot to the departure position of the docking station. The second is an arm subinstruction,
to close the arm. The third is body subinstruction, to rotate the body to the resting rotation of the docking station. The fourth is a body subinstruction, to translate the robot to the arrival position of the docking station. The fifth is a propellers subinstruction, to stop the propellers.

The grab item instruction has three different first level options: the player, the colleague and the station. Neither option is valid if the robot is at the docking station or is already holding an item. Additionally, a player option is valid if the player is holding an object and his arm is up; a colleague option is valid if the colleague is holding an item; and a station option is valid if the player’s target is hitting an object. The original values for the contact position, the contact rotation and the approach position are consulted in the station manager and converted according to the current item transform. The radius the arm should have to grab this item is also retrieved. This subinstruction has fourteen subinstructions for the player option and eight for the others. Six of the subinstructions (the first, the seventh, the eighth, the ninth, the twelfth and the fourteenth) are included only for the player option. The first is an item subinstruction, to disable the item’s collider. The second is a body subinstruction, to translate the robot to the approach position. The third is a body subinstruction, to rotate the robot to the contact rotation. The fourth is an arm subinstruction, to set the arm radius to the grab values, with a margin of 1 cm added to each side. The fifth is a body subinstruction, to translate the robot to the contact position. The sixth is an arm subinstruction, to set the arm radius to the grab values. The seventh is a share subinstruction, to tell the player that the robot is waiting for him to release the item. The eighth is a wait subinstruction, to wait for the player to open his fingers. The ninth is a wait subinstruction, to wait for half a second after the player has finished opening the fingers. The tenth is an item subinstruction, to switch the item to the robot’s parent and layer. The eleventh is a body subinstruction, to translate the robot to the approach position. The twelfth is a share subinstruction, to tell the player that the robot retreated and that he can now turn down the hand and relax is fingers. The thirteenth is a body subinstruction, to rotate the robot until it is parallel to the bottom panels. The fourteenth is an item subinstruction, to enable the item’s collider.

The drop item instruction has three different first level options: the player, the colleague and the station. Neither option is valid if the robot is at the docking station or is not holding an item. Additionally, the player option is valid if the player is not already holding an object, if his arm is up and if he is not using the target; and the colleague option is valid if the colleague does not have an object in his hand. The item’s target position and rotation depend on the option: for the player and for the colleague, it depends on the transform of their hands; for the station, the original position and rotation of the item, before the player removed it, are consulted in the station manager. The contact position, the contact rotation and the approach position are converted based on the item’s target transform. This subinstruction has sixteen subinstructions for the player option and eight for the others. Eight of the subinstructions (the first, the fourth, the fifth, the sixth, the eighth, the ninth, the tenth, and the sixteenth) are included only for the player option. The first is an item subinstruction, to disable the item’s collider. The second is a body subinstruction, to translate the robot to the approach position. The third is body subinstruction, to rotate the robot to the contact rotation. The fourth is a share subinstruction, to tell the player that the robot is waiting for him to accept the item. The fifth is a wait subinstruction, to wait for the player to
turn his hand up and stretch his fingers. The sixth is a wait subinstruction, to wait for half a second after
the player has finished opening his hand. The seventh is a body subinstruction, to translate the robot to
the contact position. The eighth is a share subinstruction, to tell the player that the robot has finished
placing the object on the palm of his hand. The ninth is a wait subinstruction, to wait for the player to
close his fingers around the object. The tenth is a wait subinstruction, to wait for half a second after the
player has secured the object with his fingers. The eleventh is an arm subinstruction, to slide each small
arm 1 cm away from the object. The twelfth is an item subinstruction, to switch the item’s parent and
layer, depending on the option (player, colleague or station). The thirteenth is a body subinstruction,
to translate the robot to the approach position. The fourteenth is a body subinstruction, to rotate the
robot until it is parallel to the bottom panels. The fifteenth is an arm subinstruction, to position the arm
in its resting configuration, with the small arms fully open and the big arms fully closed. The sixteenth
is an item subinstruction, to enable the item’s collider.

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>string name</td>
</tr>
<tr>
<td>The name of the instruction: Leave Docking Station, Return Docking Station, Grab Item, Drop Item.</td>
</tr>
</tbody>
</table>

Table 3.28: Public variables in the Instruction class from the RobotMain namespace

The state machine created for the robot main manager has three additional situations when com-
pared to the basic one described in Section 3.2. They happen only when a subinstruction is either denied
or cancelled. A subinstruction is denied when the manager issues an order to one of the sub-managers,
but they refuse to obey it because it originates an invalid instruction. Additional information on what
causes a denial can be found in Sections 3.6.3, 3.6.4 and 3.6.5. A subinstruction is cancelled when rele-
vant assumptions made to create an instruction have changed. Additional information on what causes a
cancellation can be found in Appendix A.5.

When a subinstruction is denied or cancelled, so is the entire instruction. A denial means that
the instruction is not completed successfully and the manager does not try to reissue it. There is no
point in trying to run the exact same instruction again, because the circumstances of the game have not
changed. Whatever reason led the sub-manager to refuse it would still be present for the new instruction,
which would be denied as before. A cancellation also means that the current instruction is not completed
successfully, but a new instruction, one that is computed taking into account the new circumstances,
might be. In this case, the manager reissues the instruction, with the same options used to create the
original one. The new instruction follows the path of a regular instruction caused by a user command.
If it is invalid, the state machine gives up on this instruction and returns to its idle state. If it is valid,
the state machine initiates the instruction and tries to see it through.

Situations six and seven are triggered by a denial and by a cancellation, respectively. Situation
eight is triggered when a reissued instruction is invalid.

In addition to the state transitions enumerated in Section 3.2, situation three can now originate
a situation six or seven. From situation six, the manager can only go to situation one. From situation
seven, the manager can go to situation two, if the new instruction is valid, or to situation eight, if it is
invalid. From situation eight, the manager always goes to situation one.
It is considered that the manager is still following an instruction while in situation seven, during the transitional period between two instructions. This is done to prevent the user from issuing a new command while the manager is still trying to obey a previous one.

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>int situation</td>
</tr>
</tbody>
</table>

Table 3.29: Public variables in the State class from the RobotMain namespace

The robot share variables have the same functionality as the ones used by the player.

<table>
<thead>
<tr>
<th>Share</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool waitingForPlayer</td>
</tr>
<tr>
<td>bool completedForPlayer</td>
</tr>
</tbody>
</table>

Table 3.30: Public variables in the Share class from the RobotMain namespace

The docking station is defined by the position and rotation the robot should have in order to dock or leave it.

<table>
<thead>
<tr>
<th>DockingStation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transform transform</td>
</tr>
<tr>
<td>Vector3 arrivalPosition</td>
</tr>
<tr>
<td>Vector3 departurePosition</td>
</tr>
<tr>
<td>Vector3 restingRotation</td>
</tr>
</tbody>
</table>

Table 3.31: Public variables in the DockingStation class from the RobotMain namespace

3.6.3 The body manager

The robot body manager is responsible for translating and rotating the body of the robot. It uses the instructions system and the state machine described in Section 3.2, as well as the orders system to communicate with the robot main manager.

The target defines the desired position and rotation of the body that the subinstruction must achieve. During the subinstruction, the body translates or rotates gradually towards the target, with constant speed. The subinstruction is completed when the body is on target. In addition to the target, the subinstruction has auxiliary variables that help steer the execution of the code. For a translation subinstruction, the part of the code responsible for rotating the body is ignored; and the opposite happens for a rotation subinstruction. This measure is not essential, since a translation subinstruction has the target rotation set to the current rotation of the body, while a rotation subinstruction has the target position set to the current position of the body. However, it helps improve performance, by preventing the manager from executing the sections of the code that would have no effect on the robot. It should also be noted that a subinstruction can be both a translation and a rotation at the same time, though no instructions that implement that functionality were defined.

<table>
<thead>
<tr>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector3 position</td>
</tr>
<tr>
<td>Vector3 rotation</td>
</tr>
</tbody>
</table>

Table 3.32: Public variables in the Target class from the RobotBody namespace
SubInstruction

| int index | The index of the subinstruction. |
| bool isTranslation | True for a translation subinstruction. |
| bool isRotation | True for a rotation subinstruction. |
| Target target | The target of the subinstruction. |

Table 3.33: Public variables in the SubInstruction class from the RobotBody namespace

The translate instruction is used to move the body from its current position to another one, without changing its current rotation. It is valid only if a path can be found between the initial and final points, without hitting any objects. This path may be formed only by the start and end points, if the robot can move between them in a straight line, or it can have any number of intermediate points if such is required to avoid obstacles. The number of translation subinstructions equals the amount of intermediate points plus the final point. Additional information on how the robot finds a valid path can be found in Appendix A.4.

The rotate instruction is used to rotate the body from its current orientation to another one, without changing its current position. It is valid only if the robot can rotate in the point it currently is without hitting any objects. This instruction has only one rotation subinstruction. Additional information on how the robot checks if can rotate in a given point can be found in Appendix A.4.

Instruction

string name | The name of the instruction: Translate or Rotate. |

Table 3.34: Public variables in the Instruction class from the RobotBody namespace

The order received from the main manager is defined by the value of the position or rotation wanted for the body, the layer mask and the item transform.

The layer mask can be used to ignore some colliders when searching for a path or checking if the robot can rotate in a point. The default mask excludes the colliders in the robot layer, to prevent the robot from considering its own collider in the scene and signal a collision against itself. Some situations require an additional layer to be excluded. For example, when the robot returns to the docking station, the last main manager subinstruction, that takes the robot from the departure point to the arrival point, asks the body to ignore the collider of the docking station. This is necessary because both the robot and the docking station’s colliders are rough approximations that overlap with each other when the robot docks, even though the meshes do not. When the robot approaches the docking station with the correct rotation, it knows that it is safe to ignore the collider of the docking station, because the two objects will not visually overlap.

The item transform is necessary to improve the precision of the collision checking process, by allowing this manager to consider the mesh collider of the object, instead of its box collider.

Orders

string currentOrder | The name of the order: Translate or Rotate. |
| Vector3 wanted | The final position or rotation of the body. |
| LayerMask layerMask | The layer mask used to ignore colliders that belong to certain layers. |
| Transform itemTransform | The transform of the item related to the instruction that is running in the main manager. |

Table 3.35: Public variables in the Orders class from the RobotBody namespace

74
The state machine has three additional situations when compared to the basic one described in Section 3.2. They happen only when a subinstruction is cancelled by the main manager or by the body manager. When a subinstruction is cancelled, so is the entire instruction.

A cancellation caused by the main manager happens for motives unknown to this sub-manager, and therefore it does not try to reissue the instruction. When the main manager wants to cancel an ongoing instruction in the body manager, it turns on the was cancelled switch. While executing the instruction, the sub-manager is always watching this switch and will stop what it is doing upon detecting a true value. No further communication between the managers is required.

On the other hand, a local cancellation is still under the purview of the body manager, which means that the manager will always try to complete the original order using a new instruction. If the new instruction is invalid, the was denied switch is turned on. The main manager will only know that the sub-manager was not capable of seeing the order through. It is not relevant if the sub-manager denied the order right at the start, because the first instruction was invalid, or after it had already obeyed part of the first instruction, because that instruction was cancelled and the second one was invalid. Additional information on what causes a cancellation can be found in Appendix A.5.

Situation six is triggered by a main manager cancellation. Situation seven is caused by a body manager cancellation. Situation eight happens when a reissued instruction is invalid. The additional state transitions are the same as the ones enumerated for the robot main manager.

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>int situation</td>
</tr>
</tbody>
</table>

Table 3.36: Public variables in the State class from the RobotBody namespace

### 3.6.4 The arm manager

The robot arm manager is responsible for translating and rotating every component of the arm of the robot. It uses the instructions system and the state machine described in Section 3.2, as well as the orders system to communicate with the robot main manager.

Each arm component translates or rotates along only one local axis.

<table>
<thead>
<tr>
<th>Axes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dictionary&lt;string, char&gt; translation</td>
</tr>
<tr>
<td>Dictionary&lt;string, char&gt; rotation</td>
</tr>
</tbody>
</table>

Table 3.37: Public variables in the Axes class from the RobotArm namespace

The target defines the desired position and rotation of each arm component that the subinstruction must achieve. Each subinstruction can move and/or rotate one or all components at the same time. The instructions implemented use mostly subinstructions that move only one component at a time, though some move two or even four. The subinstruction is completed when the position and rotation of all components equal the values stored in the target. There are also auxiliary variables that signal the components being adjusted, which helps reduce the computation time of the subinstruction, for the same reasons described for the robot body manager.
Dictionary\langle string, float \rangle

A dictionary with the names of all components and their target position along their local translation axis.

Dictionary\langle string, float \rangle

A dictionary with the names of all components and their target position along their local rotation axis.

Table 3.38: Public variables in the Target class from the RobotArm namespace

SubInstruction

int index

The index of the subinstruction.

Dictionary\langle string, bool \rangle

componentsToTranslate

A dictionary with the names of all components and a bool variable that is true if the current position of the component is different from the target position.

Dictionary\langle string, bool \rangle

componentsToRotate

A dictionary with the names of all components and a bool variable that is true if the current rotation of the component is different from the target rotation.

Target target

The target of the subinstruction.

Table 3.39: Public variables in the SubInstruction class from the RobotArm namespace

Instruction

string name

The name of the instruction: Open, Close or Grab.

Table 3.40: Public variables in the Instruction class from the RobotArm namespace

Orders

string currentOrder

The name of the order: Open, Close or Grab.

float rightRadius

The radius of the right side of the arm.

float leftRadius

The radius of the left side of the arm.

Table 3.41: Public variables in the Orders class from the RobotArm namespace

State

int situation

The situation number: from one to six.

Table 3.42: Public variables in the State class from the RobotArm namespace
3.6.5 The propellers manager

The robot propellers manager is responsible for rotating all six propellers. It uses the instructions system and the state machine described Section 3.2, as well as the orders system to communicate with the robot main manager.

Rotating the propellers does not actually move the robot; it is only a visual complement to add realism to its movements, all effectively controlled by the body manager. The target sets the rotation speed that should be achieved at the end of the subinstruction. The manager changes the speed gradually, with constant acceleration. The time that a subinstruction takes to complete depends on the difference between the current and target speed, but it is usually no more than a few seconds.

<table>
<thead>
<tr>
<th>Target</th>
<th>The target rotation speed for all the propellers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>float speed</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.43: Public variables in the Target class from the RobotPropellers namespace

<table>
<thead>
<tr>
<th>SubInstruction</th>
<th>The index of the subinstruction.</th>
</tr>
</thead>
<tbody>
<tr>
<td>int index</td>
<td></td>
</tr>
<tr>
<td>Target target</td>
<td>The target of the subinstruction.</td>
</tr>
</tbody>
</table>

Table 3.44: Public variables in the SubInstruction class from the RobotPropellers namespace

The change instruction is the only one available. It has only one subinstruction, with its target set to the new rotation speed. This manager does not work in the same way as the body or the arm manager. Those managers only alter the transform of the components while the state machine is in situation three, obeying a subinstruction. This manager is constantly changing the position and rotation of the propellers, even when the state machine is idle (unless the current rotation speed is zero). While in situation three, the propellers manager keeps changing the transform of the propellers, but the point of executing an instruction is to change the rate at which those transforms are altered.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>The name of the instruction: Change.</th>
</tr>
</thead>
<tbody>
<tr>
<td>string name</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.45: Public variables in the Instruction class from the RobotPropellers namespace

This manager only needs to know the desired rotation speed. All additional information, such as the longitudinal axis of each propeller, is computed internally.

<table>
<thead>
<tr>
<th>Orders</th>
<th>The name of the current order: Change.</th>
</tr>
</thead>
<tbody>
<tr>
<td>string currentOrder</td>
<td></td>
</tr>
<tr>
<td>float speed</td>
<td>The desired rotation speed for all the propellers.</td>
</tr>
</tbody>
</table>

Table 3.46: Public variables in the Orders class from the RobotPropellers namespace

The state machine used by this manager is identical to the one implemented by the arm manager.

<table>
<thead>
<tr>
<th>State</th>
<th>The situation number: from one to six.</th>
</tr>
</thead>
<tbody>
<tr>
<td>int situation</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.47: Public variables in the State class from the RobotPropellers namespace
4 Conclusion

4.1 Achievements

The main objectives established when starting this project were accomplished, although with some simplifications along the process.

The station provides an immersive scenario with several modules and a diversity of interactive objects. The models are reasonably realistic and accurate when compared to images of the real ones. The player is stopped when he strikes an object or when he tries to cross through a wall. However, some objects that should react to collisions do not. In real life, the bags would spin in the air and the laptops would shake if the astronaut collided with them, to give some examples. This simplification was made to avoid overburdening the physics engine.

The player astronaut offers a considerable amount of possible movements, combining the actions of the body, the arm and the eyes. He is able to move around the station and rotate in the air as an astronaut would in a microgravity environment. He can make several gestures with his arm and fingers, in order to manipulate objects. He is also able to look around, without moving his body, only his eyes. The colleague astronauts can receive and hold on to objects. However, they do not experience the same liberty of movements the player does. They are always in the same place, executing the same short-range gestures. This simplification was made to avoid increasing the complexity of the system that manages the non-playable characters. A system able to simulate realistically the actions of several astronauts would take a long time to design and implement. It would also decrease the performance standards because it would have to share the available computing resources with other systems, more crucial to the main goals of this simulation.

The robot is able to perform the delineated tasks autonomously, depending on the player only with respect to the indispensable input. It is possible to ask the robot to grab an object that is in the possession of the player, of one of the colleagues or stored in the station. It is also possible to ask it to deliver an object to those same recipients. These actions can be combined in numerous ways to transport several objects throughout the station. The robotic system includes a docking station, where the robot can recharge while it is not in use. The robot is able to leave and return to the docking station when ordered to do so.

The robot pays attention to the gestures of the player astronaut, to determine the right occasion to secure or release an object and to approach or retreat away from the hand. It can find a path between any two locations inside the station, if at least one possible path exists. The robot is able to occupy tight hollows in the station walls, cross through narrow passages and navigate close to obstacles without colliding with them. It also has the capacity to reevaluate the validity and the steps of an action when the circumstances of the simulation change. These changes are caused by unexpected player movements, such as moving his hand while the robot is trying to deliver an object or crossing the path of the robot while it is navigating the station.

However, some simplifications were made when computing the variables that each robot system requires in order to function correctly. Some of these variables are obtained by exploring the benefits of
running a test in a virtual simulation, instead of performing it in real life. For example, the robot is able to follow the gestures of the player by relying on information provided by its manager or by checking directly the values of the transform of its game object. Only in a simulation can the robot find if the hand of the astronaut is facing up by evaluating an angle value that is readily available. It would be much more complex to build a system offering the same capabilities in a real-life scenario. Such a system would use other sources of input, such as the robot’s cameras, and it would implement algorithms able to interpret that data in order to reach the same answer.

4.2 Future work

Several aspects could be improved in order to obtain a simulation with greater quality.

The robot could be given more liberty to choose where to store an object that the astronaut no longer needs. Instead of putting it back in the place where it came from, the robot could try to find a closer location, among objects of the same type.

The battery time of the robot could be taken into account. After the simulation had been running for a while, the robot would run out of power and automatically return to the docking station to recharge.

The physics interactions could be more realistic, by making the robot respond to collisions and programming it to return to its previous orientation, after being disturbed by an astronaut. The movement of the robot could also include position and rotation noise and be computed as the result of thrust and torque applied by each propeller, instead of having a single input source.

The range of responsibilities of the robot could be increased. For example, the robot could be programmed to document experiences, take inventory and inspect the station systems.

The interface could be designed to simulate more accurately the way an astronaut would issue requests to the robot in real life, for example with voice commands.

The number of robots in the scene could be increased and they could be programmed to collaborate with each other and agree on priority schemes. This could involve passing objects between them, deciding who should cross a passage first when there is a conflict, or stacking themselves on top of each other to recharge at the same docking station, to give some examples.

The number of users that can take part in the same simulation could be expanded. Instead of having only one player at the same time, there could be several, each controlling one astronaut and seeing the same station from different points of view.
References


A Appendix

A.1 The modeled mesh for the astronaut and for the robot

Figure A.1: Astronaut mesh

Figure A.2: Robot mesh
A.2 The game object hierarchy for the player and for the robot

![Game object hierarchy for the robot](image)

![Game object hierarchy for the player](image)

Figure A.3: Game object hierarchy for the robot and for the player

A.3 Frustum culling and occlusion culling

While developing this project, it was possible to have a privileged view over the station, in order to see all the modules at the same time and position them correctly. The player, on the other hand, will see the station from a different point of view. He will always be inside it, which means that he will never be able to see all the modules simultaneously. Rendering objects that the user will not see places a needless burden on performance. During runtime, the goal is to render the minimum amount possible of objects, without the user noticing that the some of the elements in the scene were not drawn for the current frame. The difficulty lies in knowing which objects the player would see if everything in the scene were rendered and the ones he would not. This prediction should be done by excess: it is preferable to render some objects that the user does not see than to ignore objects that the user should be seeing. Unity offers two tools that help the camera disregard irrelevant objects: frustum culling and occlusion culling.
Frustum culling is automatic and does not require any additional work from the developer. It is the process of disabling the renderers of the objects that are outside of the camera’s volume.

The scene volume rendered by the camera is a solid delimited by six faces, as displayed in Figure A.4, known as the camera frustum. It is defined by the field of view and the clipping planes. The clipping planes determine the distances at which the camera starts and stops rendering. The near clipping plane (in green) contains the closest points rendered by the camera and the far clipping plane (blue) contains the furthest. The objects that are completely outside of the camera frustum are not rendered (yellow cubes), while the ones that are partially (orange cubes) or totally inside (red cubes) are rendered. Frustum culling is one of the reasons why it is important to export the meshes from the modeling software broken down into smaller parts, even if they all belong to the scenery and never move during the game. If the cubes in Figure A.4 were all part of the same model, they would all be rendered, even the yellow ones, because that model would be partially inside the frustum. After modeling the station in Blender, the modules were imported one by one into Unity, where they were positioned together again. This allows the game engine to selectively render parts of the station, instead of always rendering it entirely.

![Frustum Culling Diagram](https://docs.unity3d.com/Manual/OcclusionCulling.png)

(a) Perspective view, without objects  
(b) Perspective view, with objects  
(c) Side view, with objects  
(d) Top view, with objects

Figure A.4: Representation of the frustum culling concept

To illustrate the process of frustum culling, the player was placed in the central module of the station and rotated around in the same spot, until completing a full circle. The results are shown in Figure A.5, with the visible modules, other than the one where the player is, named in the captions. Without frustum culling, everything in the scene is rendered, no matter the direction the player is facing, as shown in Figure A.5a. With frustum culling, the modules are rendered only if they are inside the frustum, as displayed in Figures A.5b through A.5h. As the player rotates, some modules are no longer contained by the frustum and stop being rendered, while new ones are included and show up on screen.

![Frustum Culling Examples](https://docs.unity3d.com/Manual/OcclusionCulling.png)

(a) All  
(b) Columbus, Node 2, Dragon  
(c) Columbus

---

Occlusion culling is not automatic, but it can be enforced after some set up work. It is the process of disabling the renderers that are inside the frustum but totally hidden behind other objects. The two culling tools are not mutually exclusive; when using one it is possible to still benefit from the other. 47

To build the image shown in a given frame, the objects farther away from the camera are rendered first and the closer ones are rendered last. When drawing some objects on top of others, in a process known as overdraw, a pixel can be assigned several values in the same update, which is undesirable since only the last one will be seen by the user. The goal is to identify the objects that would be totally occluded by nearest ones and prevent the game engine from rendering them. In Figure A.6, the yellow cubes are not rendered, because they are hidden by the bigger orange solid. The red cubes, however, are all rendered, even the one that is partially obscured. 47

To illustrate the process of occlusion culling, the same test done before for frustum culling was repeated and the results can be seen in Figure A.7. In Figure A.7, only A.7a has occlusion culling inactive, but frustum culling active. All other images are using both culling tools. The results are encouraging: occlusion culling removes up to three modules in some situations, renders only the ones
that are absolutely necessary and does not remove any that the player should be seeing. These results were replicated successfully in most of the tests, with the exception of a few isolated incidents. These could be modules that were rendered when they should not be, or the opposite problem, modules that were not rendered when they should be. There were also some objects popping in and out of view in front of the player. Tweaking the parameters helped keep these erroneous results to a minimum. However, without full confidence in this process, it was decided to set up the alternative system of areas described in Section 3.3.2.

(a) Columbus, Node 2, Dragon, Kibo 1
(b) Node 2
(c) None
(d) None
(e) Node 1
(f) None
(g) None
(h) Node 2

Figure A.7: Occlusion culling in the station

A.4 The robot navigation system

Before outlining the robot navigation system, it is necessary to establish which functionalities the robot should have. The main question rests on how to balance the flexibility of the robot - how many places in the station it can reach - with the need for collision checking - how it can reach those places while avoiding station obstacles, or if it can reach them at all. Two different approaches were considered.
The first possibility is more simple than the second one. It prioritizes the avoidance of collision checking over navigation flexibility. This approach has the robot moving only along previously defined paths. These paths are formed by reference points and straight lines that link them to each other, building a network that spreads across the entire station. They cut through the middle of the modules and the passages, approximately centered in relation to the four sets of panels that form the top, bottom, left and right walls of each module. The objects, such as the tables and the laptops, are fixed to the walls, leaving the central volume of the modules unobstructed. When the robot follows the paths, it is as if it were inside an invisible pipe circuit, reserved exclusively for the robot to use, where it does not have to worry about colliding with pieces of scenery. The limitations of this approach are obvious. Even if the robot is confined to the paths, the player is not. When the player has an object that he wants the robot to grab, the robot could approach the player as much as possible without leaving the paths, but it would fall to the player the task of covering the remaining distance. The player would then have to carefully position his hand under the robot, framing the object within the robot’s arms, and wait motionless until the robot had it secured. Another problem arises from the mobility range that the player enjoys. Though it can be assured that no objects break the surface of those invisible pipes, the same cannot be done for the player. The user could choose to place itself directly in front of the robot, completely blocking its way. Since the robot only has one available path to reach the intended destination, it would either get stuck, waiting for the player to move aside, or even collide with him, if no preventive collision checks were put in place at all.

The second possibility is more complex than the first one. It prioritizes navigation flexibility over avoiding to check for collisions. With this approach, the robot can move to any point inside a module, as long as there is enough empty space around that point to fit in the robot and also as long as there is at least one possible path for reaching that point. When the target point cannot be reached in a straight line, it is necessary to find an alternative, preferably the shortest one, that circumvents all objects between the current point and the final one. This eliminates the burden placed on the user of having to go to the robot. Now, it is the robot that has the responsibility of finding the player and adapt its position and rotation in order to grab the object that he is holding. The main challenge of this approach is to guarantee that the robot does not collide with anything while trying to reach the desired destination.

The first step taken to build the navigation system was to find how much space the robot occupies. The answer needs to be an approximation by excess, because it is neither feasible nor significantly rewarding to consider every curvature or small protuberance. It also needs to be based on the basic shapes offered by the functions in the Unity Physics library - the box, the sphere and the capsule - because those functions will be used later. These three shapes are displayed in Figure A.8, scaled and positioned in a way that minimizes their volume, but does not exclude any part of the robot. The box and the sphere are the ones with the smaller volume; the capsule does not bring any advantage, because the robot is not significantly bigger along one axis. But even the box and the sphere take up considerable amounts of empty space, which would stop the robot from reaching some tighter spots, even though it could.
Another solution, displayed in Figure A.9, is to use more than one shape. Eight different boxes were created, three for the body of the robot and the others for the arm components. There needs to be at least one box for each component, so that each individual box can be moved and rotated along with that component. An exception was made for the small arms. There is only one box for three components, because the robot never navigates with the arm half closed. The robot can either navigate with the arm completely open, and therefore the components in the small arms always have the same offset to each other and can share a box; or it can navigate with the arm completely closed, which means that the whole arm is included inside the body bottom box. If the robot is carrying an object, a ninth box is added to these ones, with the shape of that object. These boxes define the space that the robot occupies when it translates.

The boxes cannot be used when the robot rotates, because there is no function available to check if a box rotated from a given orientation to another collides with an object. A sphere needs to be used instead, centered at the origin of the robot, because that is the pivot point it uses when rotating. The radius equals the distance between the origin and the farthest point in the robot mesh. It can be seen in Figure A.10 that the robot never crosses the surface of the sphere, for several different rotation values. When the robot is carrying an object, the radius of the sphere is increased if necessary.
<table>
<thead>
<tr>
<th>Navigate</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>float radius</strong></td>
</tr>
<tr>
<td>Radius of the sphere that contains the robot and its object (used to rotate).</td>
</tr>
<tr>
<td><strong>Dictionary&lt; string, Bounds&gt; bounds</strong></td>
</tr>
<tr>
<td>A dictionary with all the boxes that contain the robot's components and its object (used to translate).</td>
</tr>
<tr>
<td><strong>Dictionary&lt; string, Transform&gt; transforms</strong></td>
</tr>
<tr>
<td>A dictionary with the transforms of the robot's components and its object. It is necessary to know the current position and rotation of those elements, so that the boxes and the sphere, which are in local space, can be translated and rotated to fit the robot in world space.</td>
</tr>
</tbody>
</table>

Table A.2: Public variables in the Navigate class from the Station namespace

The second step taken to build the navigation system was to write three functions that make use of the boxes and the sphere created before: CanOccupyPoint, CanMoveToPoint and CanRotateInPoint. All of these functions receive a layer mask, used ignore colliders assigned to a given set of layers.

![Image](image1.png)

(a) Function returns false

(b) Function returns true

Figure A.11: Testing the CanOccupyPoint function

The CanOccupyPoint function checks if the robot can occupy a given point, with a given rotation. It uses the OverlapBox function from the Unity Physics library, one time for each box. If one of those boxes overlaps with something, the function returns false. This function was tested with a table as an obstacle and the results can be seen in Figure A.11, with the boxes that overlapped in red and the ones that did not in green.

![Image](image2.png)

(a) Function returns false

(b) Function returns true

Figure A.12: Testing the CanMoveToPoint function

The CanMoveToPoint function checks if the robot can translate from a point to another, in a straight line, with a fixed rotation. It uses the BoxCast function from the Unity Physics library, one time for each box. If one of those boxes collides with something, the function returns false. This function was tested with a table as an obstacle and the results can be seen in Figure A.12, with the boxes that
collided in red and the ones that did not in green. It should be noted that, though neither the start nor the end positions in Figure A.12a overlap with the table, the function still returns false, because the stretched boxes that connect both positions do.

(a) Function returns false  
(b) Function returns true

Figure A.13: Testing the CanRotateInPoint function

The CanRotateInPoint function checks if the robot can rotate to any orientation in a given point. It uses the OverlapSphere function from the Unity Physics library, only one time, with the sphere defined before. If that sphere overlaps with something, the function returns false. This function was tested with a table as an obstacle and the results can be seen in Figure A.13, with the sphere that overlapped in red and the one that did not in green.

The third step taken to build the navigation system was to create search functions that find the path between two points in the station. There are three different types of search: one for the modules, one for the paths and another one for the grid. All of these searches use the uniform cost search algorithm [11], outlined in Figure A.14. The explored list is created empty and the frontier list is created with only the initial node. The initial node assigned with the initial state of the problem (either a position or a name, depending on the type of search, received as input) and zero cost. When choosing a node, the frontier is searched for the node that has the lowest cost. The chosen node is a goal node if its state equals the final position or name received as input. If the node is not a goal node, it is expanded to create new nodes and moved from the frontier to the explored list. For each new node created, it is necessary to check if there is already a node with the same state in the explored list or in the frontier. If there is a node with the same state in the explored list, this new node is discarded. If there is a node with the same state in the frontier and the cost of the new node is lower than the cost of the old node, the old node is replaced with the new node; if the cost is greater, the new node is also discarded. Each node is defined by the state, the cost and the parent node. The content of the nodes and the actions taken to expand a node vary with the type of search.

The modules search is used to find the path between any two modules of the station. It receives the name of the initial and the final module, and returns a list with the names of the modules that form the path between the start and end modules.

To expand a node, all modules directly connected to the current one are considered. The state of each new node is the name of the new module. The cost equals the one of the parent node - the current
node being expanded - plus one unit, which represents the additional module crossed. The distance is not used because this search does not receive any positions, only the name of the modules those positions belong to.

This search never fails for the current station, because all modules are connected to each other. However, the adequate verification mechanisms were put in place, so that the algorithm is prepared if a posterior version of the station is created with isolated islands of modules.

<table>
<thead>
<tr>
<th>SearchModulesNode</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>string state</td>
<td>The name of any module in the station.</td>
</tr>
<tr>
<td>float cost</td>
<td>The number of modules crossed from the initial module to this module.</td>
</tr>
<tr>
<td>SearchModulesNode parent</td>
<td>The parent node of this node.</td>
</tr>
</tbody>
</table>

Table A.3: Public variables in the SearchModulesNode class from the Station namespace

<table>
<thead>
<tr>
<th>SearchModules</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bool isValid</td>
<td>True if the search was successful.</td>
</tr>
<tr>
<td>List&lt;string&gt; answer</td>
<td>A list with all the modules the robot needs to visit to reach the final module.</td>
</tr>
</tbody>
</table>

Table A.4: Public variables in the SearchModules class from the Station namespace

The paths search is used to find the path between any two reference points of the station. This is the search that puts to practice the first navigation approach mentioned before. It uses the network of reference points and lines explained in the station manager section. The paths search receives the name of the initial and final reference points, and returns a list with the positions of all reference points that form the path between the start and end reference points.

To expand a node, all reference points directly connected to the current one are considered. Each of these new points is added to the frontier only if the robot is able to move from the chosen point to the new point, in a straight line, without hitting any objects.

Some searches establish a second condition for adding a new point to the frontier. If the initial and final points belong to the same module, only reference points that also belong to that module are considered. The ones that do not can be discarded, because the shortest answer will never include points from other modules, due to the layout of the station. This measure helps to increase performance because it decreases the number of points that the search needs to visit.

The state of each new node is the name of the new reference point. The cost equals the one of the parent node - the current node being expanded - plus the distance from the current reference point to the new one.

This search can fail, even though all reference points are connected to each other. The failure is always caused by the presence of an obstacle in the middle of the paths. That obstacle is never a piece of scenery, because they were all positioned in a way that avoids obstructing the aisles of the modules. However, the user can place the player where he wants, including in a location that blocks a path. If the player is positioned in a way that does not allow the robot to occupy the final point, a failure is immediately returned, with performing any actual search. If the player is blocking a line, creating two isolated islands of reference points, and the initial and final points do not belong to the same island, a failure is also returned, after visiting all reference points while executing the search.
<table>
<thead>
<tr>
<th>SearchPathsNode</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>string state</td>
<td>The name of any reference point in the station.</td>
</tr>
<tr>
<td>float cost</td>
<td>The distance between the initial reference point and this reference point, following the path marked by the hierarchy of nodes that originated this one.</td>
</tr>
<tr>
<td>SearchPathsNode parent</td>
<td>The parent node of this node.</td>
</tr>
</tbody>
</table>

Table A.5: Public variables in the SearchPathsNode class from the Station namespace

<table>
<thead>
<tr>
<th>SearchPaths</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bool isValid</td>
<td>True if the search was successful.</td>
</tr>
<tr>
<td>List&lt;Vector3&gt; answer</td>
<td>A list with all the reference points the robot needs to visit to reach the final reference point.</td>
</tr>
</tbody>
</table>

Table A.6: Public variables in the SearchPaths class from the Station namespace

The grid search is used to find the path between any two points inside the station. This is the search that puts to practice the second navigation approach mentioned before. It uses an auxiliary grid of equally spaced points to help the robot find a path around the obstacles. The grid search receives as input the position of the initial point and the position of the final point; and returns a list of points that form the path that can take the robot from the initial point to the final point, without hitting anything.

To expand a node, six new points are created, each of them along one of the world’s semi-axes and all at the same distance from the chosen point. This distance is called the spacing of the grid and is set to 20 cm. A greater spacing makes the grid more sparse and decreases the computation time, which means that the answer is found more quickly. However, if the spacing is increased too much, the answer might not be found at all, even if there is one, specially if the obstacles are close to each other and the openings between them are narrow. This requires a denser grid, with a smaller spacing, so that the points that the robot can occupy are included in the search. Another advantage of having a smaller spacing is the increased probability of finding a shorter path. After several tests, the value mentioned previously was found to offer a good balance between these factors. Besides those six new nodes, an additional node is also tested. This seventh node has the position of the final point. Each of these seven points is added to the frontier only if the robot is able to move from the chosen point to the new point, in a straight line, without hitting any objects.

Some searches establish a second condition for adding a new point to the frontier. If the initial and final point belong to the same module, the new point also has to inside the module bounds. A point that is outside the bounds can either be outside the station or inside another module. If the point is outside the station, it is best to discard the point immediately, instead of wasting time going through the collision checking process, which would surely return a collision with the walls. If the point is inside another module, it is also safe to discard it because the final point belongs to the same module as the initial point and the shortest path will never involve a point from another module. Another problem would come up if that path does not exist, that is, if it is impossible to reach the final point. If the grids that belong to other modules were not excluded, the search would try to reach the final point by going through other modules. It would have to explore all grid points for all modules before concluding that no answer exists. Since it is known beforehand that exploring other modules is a waste of time, because the station is not circularly connected, it is best to prevent the program from trying to do so. It should be noted that the search would still find the same answer even if this verification were not put in place,
but it would take more time to find it.

The state of each new node is the position of the new point. The cost equals the one of the parent node - the current node being expanded - plus the distance from the current point to the new one.

This search immediately returns a failure if the robot cannot occupy the final point, without performing any actual search; or if it finds that the initial and final points are isolated from each other by obstacles, after visiting all grid points while executing the search. When it returns a success, the answer is a list of points that form the path the robot should follow to go from the start point to the end point. This list is not the complete list of points obtained with the uniform cost search, but a simplified version of that list. After obtaining the complete list, all positions that are between two points that the robot can reach in a straight line, without colliding with an object, are removed, and only the remaining ones are included in the simplified list. The grid search can also return an answer without any intermediate grid points, when the robot can move directly from the start point to the end point.

Figures A.15, A.16 and A.17 display the results obtained with the grid search in four different situations. The straight line that connects the start point to the end point is represented in yellow. The complete path is represented in red and the simplified path appears in blue.

(a)  
(b)  

Figure A.15: Two examples of a failed grid search

The two situations shown in Figure A.15 are examples of a failed grid search. In Figure A.15a, the search was not successful because the robot cannot occupy the final point, since it is to close to the bag. In Figure A.15b, the final point is inside an enclosure with six walls formed by bags. There is enough space inside the bags for the robot to occupy that point, but the robot is not able to reach it because the start and end points are completely isolated from each other by the obstacles.

(a)  
(b)  

92
The situation shown in Figure A.16 is an example of a successful grid search. One of the walls of the enclosure formed by the bags was eliminated, allowing the robot to reach its inside.
The situation shown in Figure A.17 is also an example of a successful grid search. The bags are organized in three walls, that occupy the entire width and height of the module. There is only one missing bag in each wall, and the robot finds that empty space and uses it to get across the wall.

### Table A.7: Public variables in the SearchGridNode class from the Station namespace

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector3 state</td>
<td>The position in world space of any point inside the station.</td>
</tr>
<tr>
<td>float cost</td>
<td>The distance between the initial point and this point, following the path marked by the hierarchy of nodes that originated this one.</td>
</tr>
<tr>
<td>SearchGridNode parent</td>
<td>The parent node of this node.</td>
</tr>
</tbody>
</table>

### Table A.8: Public variables in the SearchGrid class from the Station namespace

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool isValid</td>
<td>True if the search was successful.</td>
</tr>
<tr>
<td>List&lt;Vector3&gt; answer</td>
<td>A list with all the points the robot needs to visit to reach the final point.</td>
</tr>
</tbody>
</table>

The fourth and final step taken to build the navigation system was to create a function that combines the three types of search to find the path between any two points inside the station. This function receives the initial and final points, and returns a list with all the points the robot needs to visit when going from the start point to the end point.

If the points belong to the same module, the function uses only one grid search. If the points belong to different modules, the function uses a modules search and several paths and grid searches. The first is the modules search, to find the list of modules that the robot has to visit. The second is a grid search, from the initial point to the frontier point with the first intermediate module. For each intermediate module, a paths search is performed; only if that fails is a grid search executed. Finally, a grid search is made for the last module, from the frontier point to the final point.

For example, if the initial point is in Columbus and the final point is in US Lab, the first step is to make a modules search, with the initial module set to Columbus and the final module set to US Lab. This search returns the following modules: Columbus, Node 2, US Lab. The second step is to make a grid search in Columbus, from the initial point to the position of the reference point Frontier Columbus.
Node 2. The third step is to make a paths search in Node 2, from the reference point Frontier Node 2 Columbus to the reference point Frontier Node 2 US Lab. If this search returns a failure, a grid search is performed instead, with the positions of those same reference points. It should be noted that the reference point Frontier Columbus Node 2 has the same world space position as the point Frontier Node 2 Columbus. The fourth step is to make a grid search in US Lab, from the position of the reference point Frontier US Lab Node 2 to the final point. The results from the successive grid and paths search need to be concatenated to obtain the final, complete answer. If any of the grid searches are unsuccessful, the process is interrupted and the function also returns a failure.

It would be possible to use only grid searches to cross an intermediate module. If the paths search fails, the time spent on that module is the sum of the paths search time with the grid search time. However, if the paths search succeeds, the overall time is only the one that the paths took to perform, which is less than the time required by a grid search. Since the paths search is usually successful, this method is more likely to decrease the computation time than to increase it. It would also be possible to use only one grid search from the initial point to the final point, but dividing the search for an answer and focusing it on individual modules was also a measure taken to improve performance.

A.5 The robot cancellation system

Before outlining the robot cancellation system, it is necessary to establish which circumstances cause the cancellation of an instruction. Most instructions take several frames to complete, which means that the state of the game when the instruction was computed may change while the instruction is being obeyed. This is troublesome when those changes directly affect the instruction. It is considered that an instruction has been affected when a new instruction computed using the same command or order is different from the original one. The station is never an agent of change, except when the user commands a restart. That does not pose a problem, because the robot is also restarted. But the player can move while the robot is following an instruction, which may cause that instruction to be outdated or even invalid. Some situations are more critical than others. When the robot is following an instruction related to the player, either to grab an object or to drop an object, the slightest body or arm movement will change the position of the hand. When the robot is following an instruction unrelated to the player, either a docking station instruction or a grab or drop instruction targeting the station or a colleague astronaut, most player movements will not bother the robot. There is a problem only if the player blocks the path the robot had decided to follow. Two different approaches were considered in order to solve these issues.

The first possibility is more simple than the second one. It prioritizes the avoidance of cancellation checking over the realism of the simulation. During a player related instruction, the player controls are frozen, which means that the user is unable to move the character, and he only regains that control after the instruction is finished. This guarantees that the hand of the player stays in the same place while the robot is working. During an unrelated instruction, the player is free to move anywhere in the station. If he blocks the path of the robot, the robot will bump against him and push him aside, because the robot is a kinematic rigidbody, while the player is non kinematic. This approach does not cancel any instructions, but that simplification comes at a cost, by causing many less than ideal situations. It is not
reasonable to expect that, in a real-life scenario, the astronaut would not move his hand, even if slightly, after asking the robot to grab or drop an object. It would be even more problematic if the robot were in a different module when called by the astronaut. The robot would take some time to reach the person, who would have to stay perfectly still while waiting for the robot, instead of dedicating himself to other tasks until the robot arrived. It should also not be acceptable to have the robot ram into a person, even if the astronaut purposefully intercepted its path.

The second possibility is more complex than the second one. It prioritizes the realism of the simulation over the need for cancelling instructions. When circumstances relevant to the current instruction change, this approach always cancels the instruction and reissues a new one, with the same options as the previous one but taking into account the current game state. This new instruction follows the path of a normal instruction. If it is valid, the manager will try to complete the instruction, and it will be successful unless the circumstances change again. If it is invalid, the manager will give up on the command or order that prompted the instruction and return to its idle state. There is no limit on the amount of cancellations that can occur for a single command or order.

The first possibility was discarded, in favor of the second one. The difficulties of implementing this approach can be condensed into three major steps, described in the following paragraphs. All robot instructions were divided into two groups, the first with player related instructions and the second with the remaining ones. Those three steps were followed twice, one time for each group.

The first step taken to build the cancellation system was to establish which variables are relevant when deciding if an instruction should be cancelled. During a player related instruction, almost all user input should cause a cancellation. If the user uses the keyboard to move the body or the arm, the hand will end up in a different place from the one it was before. However, he should be able to rotate the eyes without causing a cancellation. As for the player commands, the ones that are relevant depend on the robot instruction.

If the robot is grabbing the object, the critical commands are: lower the arm and drop the item. If the player lowers the arm, the robot should not only cancel the instruction but also abandon it entirely, because the arm is now inactive. If the player drops the item back to the station, the instruction should also be abandoned, because it is not prepared to grab an item from the station, even if it considered the new item position. The remaining commands are not relevant, for different reasons: raise the arm is invalid because the arm is already raised; show the target or hide the target are invalid because the player is holding an object; grab an item is invalid because the player already has one; accept the item is invalid because the robot is not trying to deliver an item; release the item is valid and necessary, but only when the right conditions are verified, and it should also be noted that it does not change the position of the hand.

If the robot is dropping the object, the critical commands are: lower the arm and show the target. Both of these commands are valid even when the robot is trying to deliver an item to the player, but their execution should cause the robot to abandon the current instruction. The remaining commands are not relevant: raise the arm is invalid because the arm is already raised; hide the target and grab the item are invalid because the target is not visible; drop the item is invalid because the player is not holding
any object; release the item is invalid because the robot is not trying to pick up an object; and accept
the item is valid and necessary, but only when the adequate conditions are verified.

It is known that some commands will be considered invalid by the player main manager because of
the current instruction running in the robot main manager. If the instruction was considered valid, that
offers some information on the current state of the simulation, which allows the classification of some
commands as non-problematic. For example, if the robot is in the process of delivering an object, it is
known that the player target is hidden, because the robot would not try to drop an object on top of the
target. If the player grabbed an object while the robot was preparing to drop another one on his palm, a
conflict would occur. But this conflict never happens, because the player first has to activate the target,
and only then is he able to grab an object. As soon as the player shows the target, the robot instruction
is abandoned. Using this logic, the grab item player command does not have to be taken into account by
the cancellation system.

The second step taken to build the cancellation system was to establish how to detect a change in
the relevant variables, using an effective method with a low processing cost, since this verification will
have to be done in every frame.

The variables discussed above are numerous and hard to evaluate. The robot could check all keys
that move the player, to see if any had been pressed in the current frame. However, even if no keys were
pressed, the player could still have been moved, as an effect of a collision controlled by the physics engine.
The commands pose an even greater difficulty, since the robot would need to be granted access to the
player animation status. This would violate the principle of keeping the managers separate, as the robot
manager would require extensive knowledge of the inner workings of the player manager. The solution
was to find a single variable that is external to the player manager and reflects any changes in one of the
mentioned variables.

If the robot is grabbing the object, this variable is the transform of the item that the player is holding.
Any of the problematic situations described above will cause the item to change its original position or
rotation. This is the variable that actually matters, because the robot decides its approach point based
on the current item transform. The grab item robot instruction has several steps. A cancellation can only
occur up until the point the robot parents the item. After the robot secures the item, it is not important
if the player moves his hand.

If the robot is dropping the object, the variable is the transform of the touch point that steers
the robot towards the correct place to put down an item in the player’s hand. All of the problematic
situations described above alter the position or the rotation of the touch point. A cancellation can only
occur up until the point the player parents the item. After that, it does not matter if the player moves
his hand, while the robot retreats.

The third step taken to build the cancellation system was to decide which manager is better suited
to enforce a cancellation. For a player related instruction, the choice fell on the robot main manager.
This is the manager that has the ability to immediately cancel its own instruction and any instructions
that might be running in the sub-managers. A change in circumstances can occur during any type of
subinstruction, which means that the cancellation cannot be done from one of the sub-managers. For
example, if the player moves during a wait subinstruction, only the main manager has the ability to get out of the waiting cycle. If the player moves during an arm subinstruction, the main manager needs to stop the arm from moving immediately, so that it can give an order to the body instead, to position it on the new approach point. If the player moves during a body subinstruction, the instruction running in the body manager will be cancelled externally by the main manager, and eventually the body manager may receive a new order, issued by the new instruction in the main manager.

These three steps were followed in order to build the first part of the cancellation system. This part of the system is able to handle unexpected movements during a player related instruction. If the player moves after asking the robot to grab or drop an item on his hand, the robot will adjust its path as many times needed, in order to mimic the player’s movements. If the user lowers the arm, the robot will abandon the request.

Figure A.18 shows the stages involved in a main manager cancellation. In Figure A.18a, the astronaut asks the robot to grab the object he is holding and the robot finds a path that takes it towards the approach point. However, while the robot is still following that path, the astronaut moves away from the robot, as shown in Figures A.18b and A.18c. This forces the robot to find a new path, starting at its current position and ending in the new approach point, as displayed in Figure A.18d.

![Figure A.18: Example of a main manager cancellation](image)

All the steps described previously were repeated for an instruction unrelated to the player. In respect to the first and second steps, the instruction should be updated only if the player blocks the path
of the robot. It is not practical or useful to test the keyboard input, because, even if there is player movement, the pressed keys offer no information on whether the movement put the player on a collision course with the robot. It was decided to use the navigation functions described in Appendix A.4 to perform the required cancellation checks.

When the robot is translating, first it uses an auxiliary function not mentioned in the appendix, which is a simplified version of the CanMoveToPoint function. Instead of casting the set of boxes that the original function uses, this secondary version casts only one box, that bounds the robot and the item, if it has one. If the auxiliary function returns a collision with the player, the CanMoveToPoint function is called next, to check if the player is really in the robot’s way, or only close enough to signal a collision when using the simplified box. If the auxiliary function does not return a collision, the CanMoveToPoint function is not executed. This improves the performance, because the original function requires more computation time than the auxiliary one. When the robot is rotating, it uses the CanRotateInPoint function. The original version already employs only one sphere, so no auxiliary function was created. These three functions use a layer mask to check for collisions exclusively with the player. It is safe to ignore all other colliders because the path computed by the instruction already took them into account, and the environment does not change during an instruction.

In respect to the third step, it was decided to base this part of the cancellation system in the body manager, instead of the main manager. Only this sub-manager knows the details of the robot translation path necessary to evaluate the need for a cancellation. An instruction unrelated to the player only has three different kinds of subinSTRUCTION, the ones used to give orders to the sub-managers. It does not have any wait subinSTRUCTIONS that the main manager alone can break. It is also important to note that the relevant changes in circumstances always occur during a body subinSTRUCTION. It does not matter if the player moves during an arm subinSTRUCTION, for example, because the path that will be used by the body subinSTRUCTIONS that come after has not yet been calculated. That path is only computed when the main manager enters the body subinSTRUCTION and sends an order to the body manager.

These three steps were followed in order to build the second part of the cancellation system. This part of the system is able to handle unexpected movements during an instruction unrelated to the player. If the user blocks the path, the robot will make a deviation to avoid running into the player. It is still possible to cause a collision between the two, because the speed of the player is greater than the speed of the robot. However, the robot will always try, within the limits of its maneuverability capacities, to prevent any contact with the player.

It should be noted that the player retains some degree of freedom to position himself in the path without causing a cancellation. The CanMoveToPoint function uses the current position of the robot as the initial point and the next position on the path - the one targeted in the current body manager subinSTRUCTION - as the final point. This means that the player can follow the robot while it moves around the station, without triggering a cancellation. Even though the player is blocking the path included in the current instruction, he is only overlapping the portion that was already covered by the robot. It also means that the player can move in front of the robot without causing a cancellation, as long as he keeps enough distance between the two. For example, if the player moves into the path of the robot, but pulls
back when the robot approaches, the manager does not waste time computing an alternative path that is ultimately unnecessary.

There are some circumstances that will make the robot abandon the command entirely, instead of pursuing the reissued instruction. This happens if the player positions himself right on the spot the robot is trying to reach, or if he blocks the rotation of the robot with his arm, to offer some examples.

Figure A.19 shows the stages involved in a body manager cancellation. In Figure A.19a, the astronaut on the left asks the robot to grab the object he is holding and the robot finds a path that takes it towards the approach point. However, while the robot is still following that path, the astronaut on the right moves into the path of the robot, as shown in Figures A.19b and A.19c. This forces the robot to find a new path, starting at its current position and ending in the same approach point, as displayed in Figure A.19d.

As a final note, there is one robot command that does not fit strictly within the division established. Usually, only one of the parts of the cancellation system is active at the same time, but the instruction grab item from station requires both parts to be working simultaneously. The second part is needed when the player blocks the path between the robot and the item. The first part is needed when the user asks the robot to grab an item from the station, but then commands the player to grab that same item before the robot can reach it. If this happens, the robot should abandon its instruction. The player is allowed to do this because the item still belongs to the station layer, up until the moment the robot secures it. The instruction that makes the robot grab an item from a colleague is not troubled by this problem, because the item belongs to the colleagues layer and the player is not able to snatch an object from the hand of another astronaut.