Elective In Artificial Intelligence - HRI and RA
Hanoi: A Social Robot for People's Entertainment

Luca Polenta 1794787
Michela Proietti 1739846
Sofia Santilli 1813509

July 15th, 2022

SAPIENZA
UNIVERSITÀ DI ROMA

All authors have contributed to the project in an equal way.

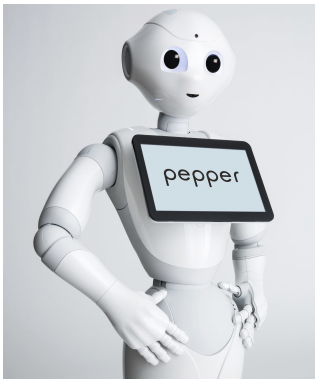
Contents

1 Abstract

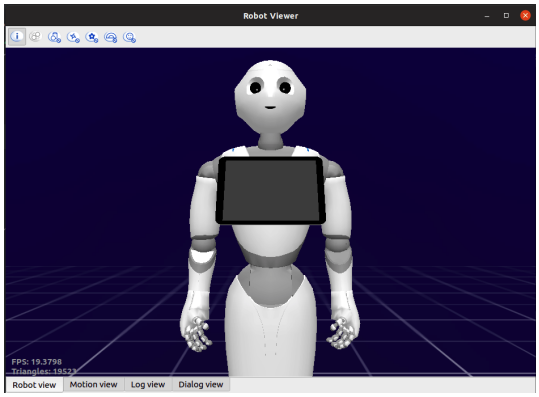
The purpose of the project is to develop a social and interactive robot for people's entertainment. It allows to play a cooperative version of the Tower of Hanoi game, in which the user and the robot execute a move alternately. The robot is also able to advise the person on the most appropriate level for him, by collecting some data about the user through an initial phase of questions. Communication between the two agents has been implemented in different ways. Additionally, a planning problem has been solved in order for the robot to be able to choose the best action to achieve victory.

2 Introduction

Nowadays, the constant progress in the field of Artificial Intelligence (AI) is leading to a significant increase in the use of AI agents in a growing variety of applications. This inevitable expansion obviously also leads to a greater integration of the latter into every person's daily life, but this requires that such tools must interact and respond in a coherent and intelligent manner to a range of stimuli produced by people and their surroundings.
The idea behind this project is the development of a robot that cooperates with a human to solve the Tower of Hanoi game. The latter consists in a simple environment with three rods. Initially, there is a certain number of disks of decreasing size placed at the leftmost rod, and the aim of the game is to move all of them to the rightmost one. In doing this, it is forbidden to place a bigger disk on a smaller one. Concerning human-robot interaction (HRI), our purpose is to make the robot interact with the user through different channels and make the interaction as natural as possible. Relatively to the reasoning part, we want to make the user and the robot execute a move alternately, and therefore the robot needs to be able to plan the best next action to make to win in the fastest way possible.
To make the user and the robot play together, we have created a web application from scratch using JavaScript (JS) and HTML, with other libraries, such as ThreeJS [1] [2] and jQuery. These tools have allowed to create a graphical environment in which the user can visualize the progress of the game and interact to make the next move for solving the problem. These frameworks have been used because they can easily, modernly and efficiently develop a graphical website with which to model 3D elements in space, and thus define an appropriate game environment for the current project. The reasoning part has been implemented using a Python framework called AIPlan4EU [3], which allows to define different types of objects, fluents, actions, and the initial and final states that need to be specified in a planning problem definition. For the HRI part, instead, NAOqi [4] has been used as it provides APIs to directly connect the Pepper robot 1a to the server that sends it the directives to execute, such as when and what to say, how to move and so on. In this way, we provided a second mean of interaction, since the robot can also interact with the user through voice and gesture. The functioning of these features has been tested through a virtual Pepper robot. This is possible through the use of the Pepper SDK plugin for Android Studio. It provides a set of graphical tools and a Java library, QiSDK [5], in order to virtualize and visualize a Pepper robot and almost all of its functionalities. This software has been developed by the company SoftBank Robotics and its interface is reported in figure 1b. Finally, in order to make all these components communicate with each other, we have exploited universal websockets [6] [7]. A great attention has been also focused on the naturalness of the interaction and on the success of the user's experience while cooperating with Pepper, trying to provide an adaptation of the robot to the specific user. This has a crucial importance, not only in the case of entertaining robots, but mostly in cases in which robotic agents are used in other domains where the robots need to interface with vulnerable people, like in healthcare or at schools. To make an additional example, even in manufacturing where robots and humans need to work together, it is important to make the robot adapt to the user it is working with to make the interaction as effective and efficient as possible.



(a) Pepper Robot



(b) Virtual Pepper in Android SDK

3 Related Works

The use of intelligent robots for playing games is not a novelty, and multiple studies have already demonstrated their great usefulness in several domains, such as healthcare and education, showing that they are not just entertaining. Examples of such uses are represented by assistive robots for people affected by autism [8] [9], pet robots for the education of preschoolers in kindergarten [10], the Bubble Robot 2a by Clementoni, and Astro [11] by Amazon, see Figure 2b. In the following two sections we will focus on previous studies that concern respectively the HRI and reasoning components.

(a) Bubble Robot by Clementoni

(b) Astro by Amazon

Figure 2: Cases of robots applied to everyday use

3.1 Human-Robot Interaction

The increase in the use of social robots in diverse applications has underlined the need for more complex interactions. To this aim, [12] introduces some suggestions for enabling the development of the next generation of socially aware computing, that includes the basic ideas behind social intelligence. By this term we refer to the ability to understand and manage social signals coming from a person we are communicating with, and it is an aspect of human intelligence that has been argued to be indispensable and perhaps the most important for success in life. The first thing that must be highlighted is that humans communicate producing many different social signals, that are all fundamental for the overall interaction. Consequently, the robot needs to interface with the user in different ways. For instance, the Bubble Robot is able to interact with

children both through a cell phone application and through real interaction in which it shows the children several simple geometric drawings that they have to reproduce. Similarly, Astro is able to communicate both verbally and through a tablet. However, in both cases, the absence of arms and limbs precludes more complex interactions. On the other hand, in order to allow a natural interaction, the robot has to recognize signals coming from the human counterpart. This is crucial for example in [9], in which modular robots are used to record the responses of the autistic user that, in association with his or her medical record, are exploited to develop further diagnosis and insights to improve the therapies to be administered. Another factor that has a great relevance, as explained in [12], is interpersonal distance. In fact, it defines the type of relationship between people and it is divided into 4 zones: the intimate zone, the casual-personal zone, the socio-consultative zone and the public zone. While building a social robot, it is necessary to respect such division and never make the robot enter the intimate zone. Although there have been several important advances in machine analysis of social and behavioural cues like blinks, smiles, crossed arms, laughter, and similar, the design and development of automated systems for social signal processing (SSP) are rather difficult. To improve naturalness, [13] presents a way of performing user's adaptation thus improving the robot's autonomy. The authors propose to build users' profiles to make the robot able to act proactively.

3.2 Reasoning Agents

Most of the previously cited applications involve a reasoning component which allows to plan the execution of the considered tasks. Reasoning can be performed in different ways and at different levels of complexity, going from simple if-else statements to using specific languages, such as PDDL, to formally define more complex problems. For example, Astro [11], mainly thanks to its high integration with Alexa, is able to perform highly difficult tasks and play games.
Planning is a decision-making technology that allows to determine how and when to act in order to achieve a desired objective. It has a fundamental importance for many application areas that need fast, automated and optimal decisions, such as agile manufacturing, agrifood or logistics. The AIPlan4EU project (`https://www.aiplan4eu-project.eu/`) aims at developing a uniform, user-centered framework to access the existing planning technology and by devising concrete guidelines for who is willing to use it. Among the use-cases presented within this project, there is warehouse logistics. In this context, they consider Magazino, an innovative company producing robots for warehouse intra-logistics. In Magazino, each robot is given a prioritized list of jobs to perform, and each of them is associated with a hand-written plan, encoded using a behavior tree formalism. However, this solution comes with some drawbacks, since handwriting of the behavior trees requires expert knowledge and quite some brain power to be carried out without mistakes. Planning techniques can be used to guide the user to design the complex plans required to accomplish the tasks the robot is given, and to follow their execution taking into account the automatic recovery from unforeseen circumstances, e.g., after a failure has occurred.

4 Proposed Solution

As previously explained, the aim of this project is to develop a robot that plays with the user in a collaborative version of the Tower of Hanoi game. The proposed solution, illustrated in Figure 3, is basically made up by three main components:

- A web application, developed using HTML, JS, and several other libraries, namely ThreeJS and jQuery. This component represents the graphical part of the game. It allows the user to always know the current state of the game and to move a disk when it is his turn to play;
- A planning component which leverages the AIPlan4EU framework to define the game's domain and

find a sequence of actions to reach the solution. The planner is called inside the server that sends data to the web application, thus allowing the robot to make its moves;

- A final component that exploits NAOqi APIs to make the robot move and talk.

The communication between these three components is guaranteed through the use of two websockets. In particular, the websockets are used to connect the web application and the Pepper-related frameworks to their respective servers. The planner, which simply consists of python code, is called within the web application's server with a simple function's call.

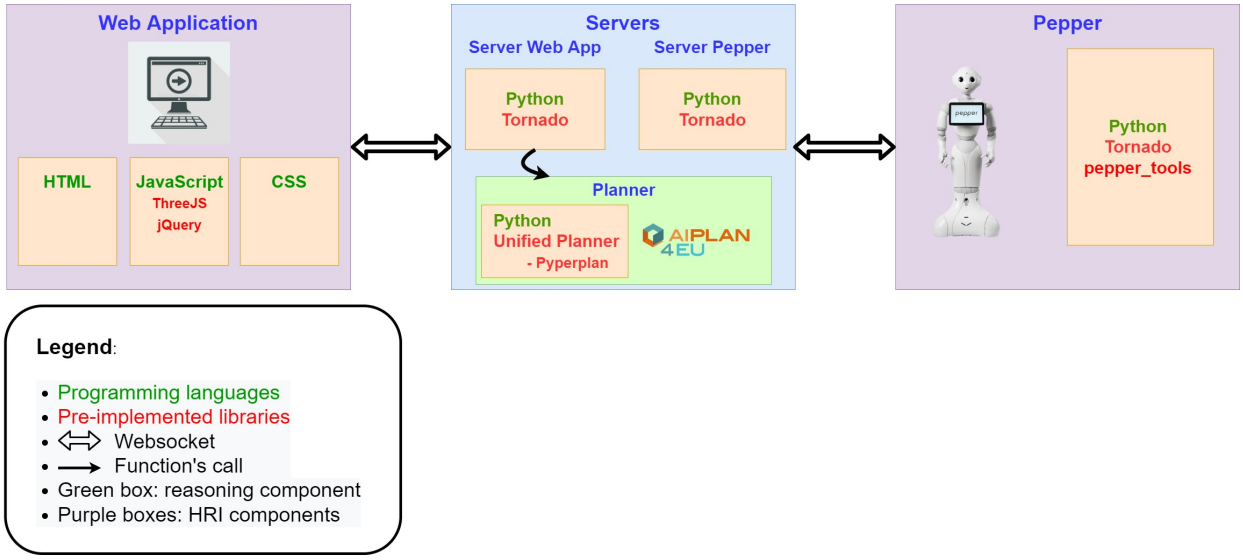


Figure 3: Main components developed within the project. We highlight the used languages, the pre-implemented libraries that have been exploited, and how the different components are connected.

Concerning the HRI part, the interaction between the robot and the user can be divided in three different phases:

- An initial phase, in which the user is presented with a questionnaire that is used to assess his experience and to build an extremely simplified user's profile to suggest the most appropriate difficulty level to play. The communication takes place verbally;
- A playing phase, in which the user interacts with the tablet by taking turns with the robot in an attempt to move all the disks to the rightmost rod and achieve victory. The robot also informs the user verbally of when it is his turn to play or if he is trying to make a move that is against the rules of the game;
- A final phase, in which the robot asks the user to rate the game and the difficulty encountered while playing using the tablet. Based on the answer to this last question, the robot updates the user's profile and if the user chooses to play again, the difficulty of the game is adapted accordingly.

In order to start the interaction, we take into account the considerations made by [12] concerning the interpersonal space. In particular, we start an interaction with the user just if he gets closer than a threshold, thus manifesting his desire to play. Instead, the robot does not get closer, thus avoiding it entering the

person's intimate zone. As in [13], the user's profile is instead used to adapt the robot to the user and to make it able to be proactive by making suggestions to improve the user's experience. In fact, if the game is too hard for the user, he might get discouraged and might not want to play again.

The initial choice of the difficulty level represents a very simple form of reasoning involving some if-else statements on the user's answers. The actual reasoning is performed in order to solve the game, computing the best sequence of moves that allows to reach the goal. From the final plan, we only take the first action, since after that it will be the person's turn to make a move, and therefore based on the outcome of it, the robot will have to come up with a new plan.

5 Implementation

This section provides a detailed explanation of how each component of the project has been implemented, from the tools and frameworks that have been used to the reasons motivating our choices.

5.1 Experimental setup

This project has been developed in a Linux environment, in particular using Docker, which allows creating a container where it is possible to work independently from the rest of the system. In this way, any change made on or by the code toward Pepper's environment or framework are not permanently applied, therefore it is not affected by development problems. This capability is provided by an additional abstraction through OS-level virtualization by executing the following commands: i) run a docker image with all libraries considered:

```
./run.bash
```

and run such image to enter the architecture of your robot:

```
docker exec -it pepperhri tmux a
```

With these commands it is possible to create various environments in which to start a NAOqi server for virtual or physical connection to a Pepper robot and other environments for the execution of additional functionalities. For example, one of such environments has been used to activate a server that could connect together Pepper, the graphical website for user interaction, and the planner for providing intelligent responses to the game by the robot. Similarly, a separate environment is needed for using the NAOqi server, to allow the communication with the robot, which is located at the top of the hierarchy of the architecture which it is a part of. This hierarchy is shown in figure 4.

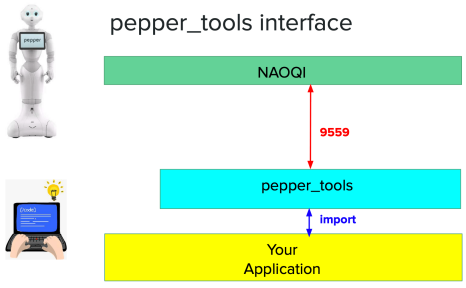


Figure 4: NAOqi architecture.

5.2 Human-Pepper Interactions

5.2.1 *Pepper_ tools* for verbal and gestural communication

In order to exploit all Pepper's functionalities, a python program has been developed. First, Pepper is accessed via *pepper_ tools* utilities. These allow to easily call different pre-implemented functions, shown in Figure 5, to make Pepper perform different tasks, such as making it say something, make it move or use its sensors for different purposes.

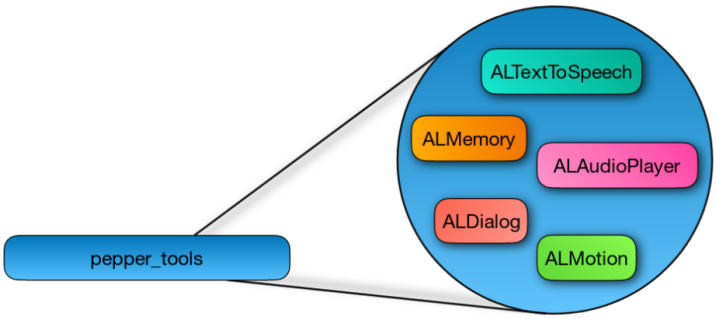


Figure 5: Modules in *pepper_ tools*.

Once a connection is established with the physical robot or with the robot simulator, we use *pepper_ tools* functions to read information provided by the sonar sensor, to check whether there is a person in front of it, closer than two meters.

```
# Sonar Activation
pepper_cmd.robot.startSensorMonitor()

stop_flag = True
try:
while stop_flag:
  p = pepper_cmd.robot.sensorvalue()
  if(p[1]!=None and p[1]<3):
    print("I have located the user")
    stop_flag=False
  else:
    if p[1]==None or p[1]=="None":
        print("I don't locate any users near me")
    else:
      print("I have located the user, but it is not close enough")
    #I stop 3 seconds before checking for another person
    time.sleep(3)
except KeyboardInterrupt:
    pass

# Sonar Deactivation
pepper_cmd.robot.stopSensorMonitor()
```

If the sonar detects a person, Pepper starts the interaction. In this phase, the interaction takes place verbally and it has been implemented manually. In fact, it is necessary to make the verbal greetings asynchronous with respect to the movement of the waving arm. This is possible by calling a function that returns the session environment variable, from which to call the modules that allow us to define the asynchronous vocal interaction during the initial greeting.

```
    Our_tts_service = pepper_cmd.robot.session_service("ALTextToSpeech")
    Our_tts_service.setLanguage("English")
    Our_tts_service.setVolume(1.0)
    Our_tts_service.setParameter("speed", 1.0)

    Our_tts_service.say("Hello! Do you want to play with me?"+" "*4, _async=True)
    doHello()
```

As we can see in line 2, the ALTextToSpeech module, which is among the most popular one in *pepper_tools*, is used. It consists of a unit that makes the robot speak. Specifically, it sends the sentence to be said to a speech synthesis module that customizes the sentence with Pepper's characteristic voice by following different guidelines, such as speed or language. The result of this synthesis is finally sent to the speakers integrated in the robot. Similarly, also the waving gesture has been manually implemented, using the ALMotion module, as we are interested in applying those movements with customized timelines.

```
def doHello():
  ourSession = pepper_cmd.robot.session_service("ALMotion")

  jointNames = ["RShoulderPitch", "RShoulderRoll", "RElbowRoll", "RWristYaw", "RHand", "
    HipRoll", "HeadPitch"]
  jointValues = [-0.141, -0.46, 0.892, -0.8, 0.98, -0.07, -0.07]
  times  = [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0]
  isAbsolute = True
  ourSession.angleInterpolation(jointNames, jointValues, times, isAbsolute)

  for i in range(2):
    jointNames = ["RElbowYaw", "HipRoll", "HeadPitch"]
    jointValues = [1.7, -0.07, -0.07]
    times  = [0.8, 0.8, 0.8]
    isAbsolute = True
    ourSession.angleInterpolation(jointNames, jointValues, times, isAbsolute)

    jointNames = ["RElbowYaw", "HipRoll", "HeadPitch"]
    jointValues = [1.3, -0.07, -0.07]
    times  = [0.8, 0.8, 0.8]
    isAbsolute = True
    ourSession.angleInterpolation(jointNames, jointValues, times, isAbsolute)

  return
```

After this, Pepper invites the user to answer some questions to suggest him the most appropriate level to play. Even in this case, some custom functions have been implemented in order to make the robot wait for an answer after making a question. As we can see in the following code, we use the already provided *say* and *asr* functions, which respectively allow us to make the robot ask the question and perform autonomous speech recognition on the user's answer.

```
def handleWaitingAnswer(question,counting_answers):
  pepper_cmd.robot.say(question+" "*8)
  stop_flag2 = False
  while not stop_flag2 and not stopQuestionaire:
    vocabulary = ["yes", "no"]
    timeout = 10 # seconds after function returns
    answer = pepper_cmd.robot.asr(vocabulary, timeout)
    if answer=="yes":
        counting_answers+=1
        stop_flag2=True
                elif answer=="no":
        stop_flag2=True
                else:
        print("answer: "+answer)
        pepper_cmd.robot.say("I didn't quite understand. Answer me only 'yes' or 'no'. "+
    question+" "*8)
```

```
        return counting_answers
```

As we will better explain in the following, the robot suggests a level to the user by performing some simple reasoning over the received answers. After the questionnaire, the user can select the level he wants and start playing. In order to collaboratively solve the game, the user and the robot need to execute a move alternately. While doing so, the robot informs the user when it is its turn to play and mimics to the user the direction in which Pepper has decided to move the object. This movement is implemented through a custom function similar to *doHello()*. While playing, Pepper also warns the user when he tries to make an invalid move, such as picking from an empty rod or trying to move a disk on a smaller one. This is done by receiving messages from the web application through a websocket, as will be better explained in one of the following sections. The warning is given to the user by using the aforementioned *asay*, whose name stands for animated say, meaning that the robot gesticulates while speaking.

```
            elif(received[0]=="RuleViolation"):
                print("%s!!RuleViolation!!%s" %(RED,RESET))
                pepper_cmd.robot.asay('You violated the game rules. Try again.'+' '*8)
            elif(received[0]=="EmptyRod"):
                print("%s!!EmptyRod!!%s" %(RED,RESET))
                pepper_cmd.robot.asay('You chose an empty rod. Try again.'+' '*8)
            elif(received[0]=="ActionDone"):
                print("%s!!ActionDone!!%s" %(GREEN,RESET))
                frasi = ['Now it is your turn.', 'You go', 'Go ahead', 'Make your move']
                r = random.randint(0, 3)
                pepper_cmd.robot.asay(frasi[r]+' '*8)
```

Finally, when the robot and the user manage to win by moving all the disks to the rod on the right, the robot informs the user while making a little victory dance, that has been implemented similarly to the *doHello* function, leveraging the ALMotion module. The robot then invites the user to rate his experience and the difficulty he encountered using the tablet, as we will better explain later, and he can then play again or leave.

5.2.2 Website for interaction with the tablet

The website has been developed from scratch using HTML, JavaScript and CSS. In particular, we have exploited ThreeJS [1] [2], a JavaScript library for 3D Web graphics based on WebGL. More specifically, the latter is a low-level implementation that allows OpenGL to run on the browser, while ThreeJS is a high-level implementation of OpenGL that hides the pragmatic and time-consuming coding of gl programs, vertex shaders, fragment shaders, buffers, projections and rendering. ThreeJS allows generating and manage complex 3D objects in a very simple way on browsers. The graphical environment is created through a hierarchical pyramidal model where each object is connected to the others through a parent-child relation. By accessing it by ID or name it is possible to change some of its properties such as position, rotation, appearance and so on. The hierarchical model of the game page is shown in Figure 6:

```
Scene [Scene]
  ├─Hemisphere Light [AmbientLight]
  ├─Directional Light [DirectionalLight]
  └─Game [Group]
    ├─rod1 [Mesh]
    ├─rod2 [Mesh]
    ├─rod3 [Mesh]
    ├─floor [Mesh]
    ├─disk1 [Mesh]
    ├─disk2 [Mesh]
    └─disk3 [Mesh]
```

Figure 6: Scene structure for the easy level with only three disks.

We can see that there is an ambient light and a directional light that allow to properly illuminate the scene and a group named *Game*, which contains all the main components of the game, namely three rods, their basement (floor) and a variable number of disks depending on the chosen level.
The game is mainly managed through buttons. When a button is clicked, a Javascript code will handle the consequence (opening of a new page or start of a new action). The graphics of the different web pages that have been implemented are shown in the *Results* section. The first page that is presented to the user is an *Home page* in which we needed to insert a text area to write the IP address to be used in order to connect to Pepper. Actually, this has not been needed while using the physical robot, since the IP addresses have all been properly defined within the code, without the need to manually insert them. Additionally, there is another button that allows the user to decide when to start the quiz to asses the most appropriate level for him.
Once this button is clicked, the user is redirected to the quiz page, which basically consists of a *Waiting page* that remains there until the user has answered all questions made by Pepper. Moreover, this page contains a button that allows the user to skip the quiz and go directly to the next page. The latter consists in the *Game Menu page*, where the user can select the level to play through three different buttons. Furthermore, there is a tutorial button that allows to check the game rules. In particular, by leveraging the *onclick* function we change the visibility of the game's explanation so that, being initially invisible, it appears when the user clicks on the button and it disappears if the user clicks again.
After the user has chosen the level to play, we move to the actual *Game page*. Here, we have manually built the game's environment using different geometries and materials from ThreeJS. In particular, each element in the scene is a ThreeJS *Mesh*, for which we need to specify a geometry and a material. The floor consists of a simple *BoxGeometry* with very small height and a width and depth that adapt to the browser's size. The rods are built using the *CylinderGeometry*, and they all have an adaptable height and very small base radius. The floor and the rods share the same material, which is a ThreeJS light-brown colored *MeshPhongMaterial*. Finally, the disks have *TorusGeometry* with different radii and *MeshPhongMaterial* of different colors. The position of each object can be defined using the *position* and *rotation* attributes of the corresponding mesh, and this is also used to handle the disks' animation. While the user plays, we keep three arrays (*leftStaff*, *centerStaff* and *rightStaff*) containing the disks (numbered from 1 to N=maximum number of disks in the current level) that are stacked on each rod. In this way, the height at which a disk needs to be placed when it is moved to a new rod r can be automatically computed by multiplying the number of disks already present on r for the height of a disk. By construction, it is also very simple to check whether any game's rule gets violated (further details about how violations are handled will be hgihlighted later).
In order to play, the user will exploit buttons, always handled through the JavaScript *OnClick()* event. In fact in the game page, in addition to buttons to go back to the main menu, or to restart or exit the game, we also introduce a button "Pick" below each rod: clicking on one of that buttons means selecting the upper disk of the corresponding rod.
Once a user has made his choice about which disk to move, the action selection buttons change, as shown in Figure 8: the one from which the item to be moved has been selected becomes "Undo" to allow the user to withdraw his choice, while the others become "Drop" to let him select the destination of the move. After the user has selected the destination, if the selected move is not against the game's rules, the disk is moved there and all the buttons become "Pick" buttons again. The actual animation consists of many successive calls to a function that accesses the hierarchical model of the scene and repetitively modifies the position of the object to be moved. First, the disk is risen from the rod it is located at. Then, it is moved in the direction it needs to go, and finally, once it has arrived on the chosen rod, it is placed at the correct height, depending on how many disks are already present. Subsequently, once the planner associated with Pepper has processed the solution and passed the next move to the JavaScript associated with the website with the corresponding server, this animation function is called again to animate the move proposed by Pepper. During this entire process, the rod selection buttons "Pick" are disabled to prevent the user from performing further actions and they are reactivated only when the animation is complete. Additionally, to provide a user-friendly game interface, a banner has been also inserted at the top of the game window, which informs the user about what is happening.

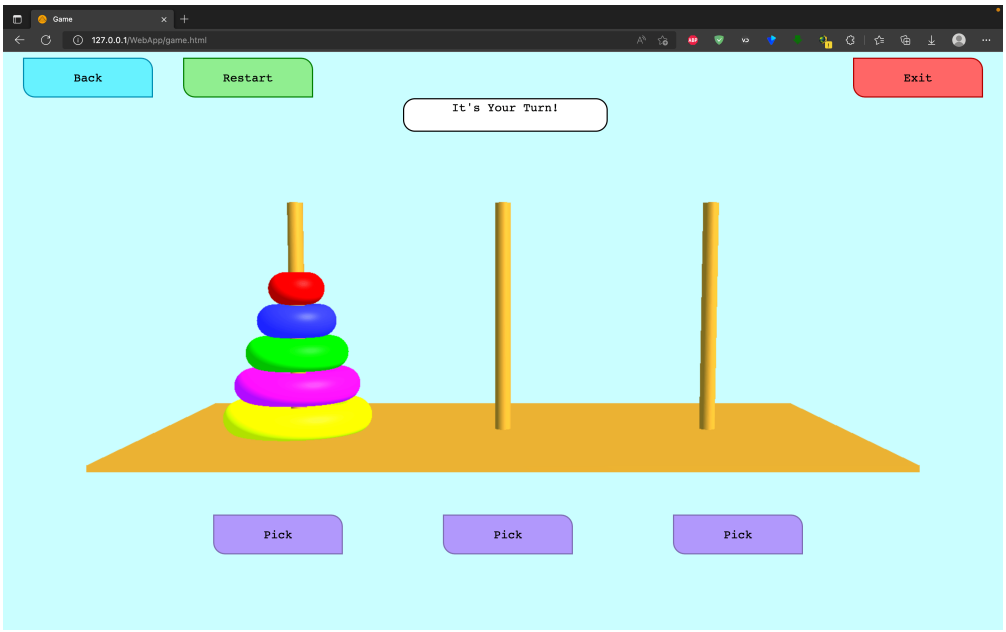


Figure 7: Game page. First Situation in the Game. (Medium level)

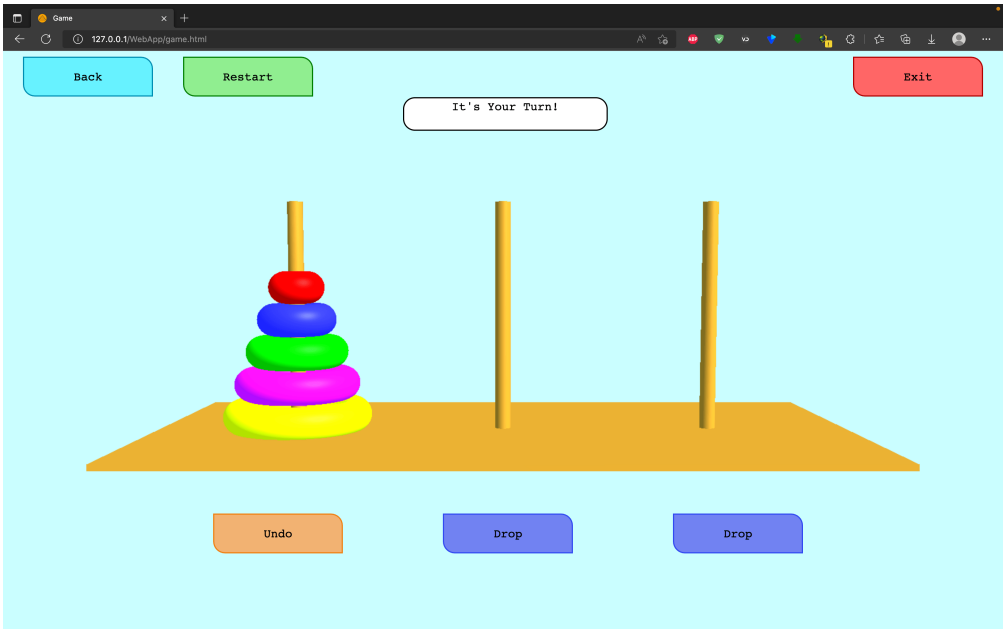


Figure 8: Game page. Second Situation in the Game. The user made his choice. Now he can drop the disk on another rod or undo the selection of the disk. (Medium level)

Finally, once the game is won, the user is automatically redirected, after some seconds, to a final *Rating page*, where he can judge his experience with Pepper and the difficulty he met while playing. He can answer by selecting from 1 to 5 stars or squares respectively. In order to implement the 5 stars rating system, we have defined a *div* with class name *stars* and we have added a form tag with input type *radio*. Each star has its own label that is used inside the CSS file.

```
        <div class="stars" align="center">
            <form action="">
                <input class="star star-5" id="star-5" type="radio" name="star"/>
                <label class="star star-5" for="star-5"></label>
                <input class="star star-4" id="star-4" type="radio" name="star"/>
                <label class="star star-4" for="star-4"></label>
                <input class="star star-3" id="star-3" type="radio" name="star"/>
                <label class="star star-3" for="star-3"></label>
                <input class="star star-2" id="star-2" type="radio" name="star"/>
                <label class="star star-2" for="star-2"></label>
                <input class="star star-1" id="star-1" type="radio" name="star"/>
                <label class="star star-1" for="star-1"></label>
            </form>
        </div>
```

To implement the actual star, we need to add the star icon, and this has been done using font-face.

```
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/font-awesome/4.2.0/css/font-
    awesome.min.css"> <!--Star icon-->
```

When the user selects the star corresponding to his rating, the stars up to that one become lit, and this transition is implemented in the CSS file as follows:

```
  div.stars {
    width: 18%;
    margin: auto;
    position: relative;
    direction: rtl;
    border-radius: 0px 20px 0px 20px;
  }

  input.star { display: none;}

  label.star {
    float: right;
    padding: 10px;
    font-size: 36px;
    color: #444;
    transition: all .2s;
  }

  input.star:checked ~ label.star:before {
    content: '\f005';
    color: #FD4;
    transition: all .25s;
  }

  input.star-5:checked ~ label.star:before {
    color: #FE7;
    text-shadow: 0 0 20px #952;
  }

  input.star-1:checked ~ label.star:before { color: #F62; }

  label.star:hover { transform: rotate(-15deg) scale(1.3); }

  label.star:before {
```

```
35      content: '\f006';
36      font-family: FontAwesome;
37    }
```

The squares for difficulty ratings have been implemented exactly in the same way, by changing the *content* attributes in lines 20 and 35 of the CSS file, thus substituting star icons with rectangles. Even in this final HTML page there are two buttons: the first one is used to play again with the difficulty level selected by Pepper based on the user's responses, while the second one is used to end the interaction with the robot.

5.3 Reasoning Agents

5.3.1 Planning for a solution of the game: AIPlan4EU Unified Planning framework

Pepper's reasoning capabilities are provided and implemented through the AIPlan4EU Unified Planning framework [1]. This framework exploits the Unified Planning library, making it easy to formulate planning problems in a planner-independent way and to solve it by invoking different automated planners (*pyperplan*, *tamer*, *enhsp*, *fast-downward*) and heuristics.
The resulting formulation of the problem "towersHanoi" in the case of the "Easy level", with only three disks, is the following:

```
problem name = towersHanoi

types = [Item]

fluents = [
  bool is_disk[disk_i=Item]
  bool clear[disk_i=Item]
  bool on[disk_i=Item, disk_pos=Item]
  bool smaller[disk_i=Item, disk_j=Item]
]

actions = [
  action move(Item disk, Item l_from, Item l_to) {
    preconditions = [
      is_disk(disk)
      smaller(disk, l_to)
      on(disk, l_from)
      clear(disk)
      clear(l_to)
    ]
    effects = [
      clear(l_from) := true
      on(disk, l_to) := true
      on(disk, l_from) := false
      clear(l_to) := false
    ]
    simulated effect = None
  }
]

objects = [
  Item: [loc_1, loc_2, loc_3, disk_1, disk_2, disk_3]
]

initial fluents default = [
```

[1]This library has been developed for the AIPlan4EU H2020 project[3], funded by the European Commission under grant agreement number 101016442.

```
  bool is_disk[disk_i=Item] := false
  bool clear[disk_i=Item] := false
  bool on[disk_i=Item, disk_pos=Item] := false
  bool smaller[disk_i=Item, disk_j=Item] := false
]

initial values = [
  is_disk(disk_1) := true
  smaller(disk_1, disk_2) := true
  smaller(disk_1, disk_3) := true
  smaller(disk_1, loc_1) := true
  smaller(disk_1, loc_2) := true
  smaller(disk_1, loc_3) := true
  is_disk(disk_2) := true
  smaller(disk_2, disk_3) := true
  smaller(disk_2, loc_1) := true
  smaller(disk_2, loc_2) := true
  smaller(disk_2, loc_3) := true
  is_disk(disk_3) := true
  smaller(disk_3, loc_1) := true
  smaller(disk_3, loc_2) := true
  smaller(disk_3, loc_3) := true
  on(disk_3, loc_1) := true
  on(disk_2, loc_1) := true
  on(disk_2, disk_3) := true
  on(disk_1, loc_2) := true
  clear(disk_2) := true
  clear(disk_1) := true
  clear(loc_3) := true
]

goals = [
  on(disk_1, disk_2)
  on(disk_2, disk_3)
  on(disk_3, loc_3)
]
```

In the game of the Towers of Hanoi we had to deal with two types of objects: locations and disks. Locations are always three (rod 1, rod 2 and rod 3). The number of disks has been parameterized depending on the level the user wants to play: three disks for the Easy level, five for the Medium and seven for the Hard one. Also fluents, actions, initial and final states need to be specified in a planning problem definition. Fluents are conditions that can change over time. They can be applied to objects or not. Here we have:

- *is_ disk(x)*, which returns true if an item x is a disk, false if it is a location;
- *clear(x)*, which returns true if item x has no objects above itself. If true, it means that it is possible to move it on a different place, if x is a disk, or to place another disk on it, both if x is a disk or not;
- *on(x, y)*, which returns true if item x is on item y;
- *smaller(x, y)*, which checks if item x is smaller than item y;

In the last two fluents x can be only a disk, while y can be both a location or a disk.
The unique action needed is the *move(i, a, b)*, that allows to move *disk_ i* from a to b. It is specified through preconditions and effects, defined with fluents.
A simplification was adopted by considering the locations as disks, considering both in the *Item* class. Therefore we set that all the disks are smaller than each location *loc_ i* and that the location is clean only if has no disks placed on it. The only thing that differentiate a location from a disk is that locations obviously cannot be moved from a to b through the *move* action.

The goal state requires all the disks employed in the game to be on the third rod (the one on the right). The initial state can differ, depending on the game. In fact, since our robot is cooperative, game moves will be performed alternately by the user and the robot. In the formulation presented here the user already made his first move, by placing the smaller disk (*disk_ 1*) on the second rod (*location 2*) and this information was sent to the problem formulator in order to plan starting from the current goal situation. In this project, *pyper-plan* is the planner employed to return the optimal plan towards the goal state. The problem defined above is passed to this solver, that is called by the program as follows:

```
for planner_name in ['pyperplan']:
    with OneshotPlanner(name=planner_name) as planner:
      result = planner.solve(problem)
          if result.status == PlanGenerationResultStatus.SOLVED_SATISFICING:
              ...Do Something...
        else:
              noPlan = "No plan found"
              ...Do Something...
```

Once the solver returns a sequence of actions towards the goal, only the first one is sent to the web browser and applied in the game. Subsequently, the planner will wait for the human to take the next action and recompute a new optimal plan from the situation that arises.

5.3.2 Simple reasoning for user profile adaptation of the interaction

After the questions, Pepper has collected the information about the user and performs a simple reasoning in order to suggest the most appropriate level for him. This is based on the total number of affirmative answers (*counting_ answers* in the code). If the user answers "Yes" only one time then Pepper will suggest the Easy level; if he does so 2 or 3 times, then the medium level is advised; if 4, the Harder level is suggested.

```
def questionLevel():
    counting_answers=0
    counting_answers=handleWaitingAnswer('Are you less than 10 years old?',counting_answers)
    counting_answers=handleWaitingAnswer('Are you already familiar with the towers of Hanoi?
    ',counting_answers)
    counting_answers=handleWaitingAnswer('Have you played before?',counting_answers)
    counting_answers=handleWaitingAnswer('Do you like solving problems on recursion?',
    counting_answers)

    if stopQuestionaire==False:
  if (counting_answers==2 or counting_answers==3):
      pepper_cmd.robot.asay("I suggest the medium level for you"+" "*8)
  elif (counting_answers==4):
      pepper_cmd.robot.asay("I suggest the hard level for you"+" "*8)
  else: # valueStopping=="stop" or counting_answers==1 or error
      pepper_cmd.robot.asay("I suggest the easy level for you"+" "*8)
    pepper_cmd.robot.asay("If you want, you can take a look at the tutorial. Let's start
    playing!"+" "*8)
    pepper_cmd.robot.normalPosture()
    return
```

A similar reasoning is carried out also in order to choose the level at which to redirect the user after the ratings. It is exclusively based on how many squares the user selects in voting the difficulty he encountered: if the user chooses 1 or 2 squares over five, he is redirected to an harder level; with 3 stars, he is redirected to the same level he just played; with 4 or 5 squares the site redirects him to an easier level. Actually this reasoning is not performed passing through the robot, but directly within the website:

```
  // Setting the level to play
  let difficulty = sessionStorage.getItem('difficulty');
  if (difficulty != 0) {
```

```
    if (difficulty > 3) {
      levelToPlay=1;
      leftStaff=[3, 2, 1];
    } else if (difficulty == 3) {
      levelToPlay=2;
      leftStaff=[5, 4, 3, 2, 1];
    } else {
      levelToPlay=3;
      leftStaff=[7, 6, 5, 4, 3, 2, 1];
    }
  }
```

5.4 Websocket Communication

The communication between different components within this project is carried out through the use of websockets [6] [7]. They are a computer communication protocol that provides full-duplex communication channels over a single TCP connection. The main advantages of this technology are its ease of use and high interoperability as it allows communication between different programming languages. More specifically, two websockets have been developed to connect the web application and the Pepper client to the respective servers.

5.4.1 Server

The server components of the two connections are handled through the same python file. Within it, two connections are defined on two different ports where port 9020 is reserved for the connection to the site and port 9030 is used for the connection to Pepper (the ports clearly change while working with the physical robot). Once the user interacts with the website, the server receives several instructions through the first connection and depending on what is requested, it will produce a plan and/or have the Pepper robot interact with the user in the different ways. Thus, both the Pepper robot and the website are clients of this server. Furthermore, since the server's IP address may also vary depending on the connection to which the operating system is hooked, it prints its public IP address on the terminal and it is customizable on the user website, as shown in section 5.2.2. The code by which the connection is generated is the following:

```
1   # Run web server for HTML
2   application = tornado.web.Application([(r'/websocketserver', MyWebSocketServer),])
3   http_server = tornado.httpserver.HTTPServer(application)
4   http_server.listen(server_port)
5   print("%sWebsocket server for HTML listening on port %d%s" %(GREEN,server_port,RESET))
6
7   try:
8       tornado.ioloop.IOLoop.instance().start()
9   except KeyboardInterrupt:
10      print(" -- Keyboard interrupt --")
11
12
13  if (not websocket_server is None):
14      websocket_server.close()
15  print("Web server for HTML quit.")
```

To easily implement websockets, Tornado [14] was used. It is a web framework and asynchronous network library in Python which facilitates the scripting of client-server connections. Furthermore, in line 2 it is possible to notice that the definition of a Tornado Application is associated with a custom class, which in the case of this connection is the "MyWebSocketServer" class. Inside this class, the behavior of the websocket

is established, both in terms of the standard steps to open, close or manage a connection, and in terms of what to do when a message is received. A portion of the code of this class is reported below:

```
1  class MyWebSocketServer(tornado.websocket.WebSocketHandler):
2
3      def open(self):
4          global websocket_server, run
5          websocket_server = self
6          print('New connection with the website\n')
7
8      def on_message(self, data):
9          global code, status
10         received = data.split("_");
11         if(received[0]=="RuleViolation"):
12             print("%s!!RuleViolation!!%s" %(RED,RESET))
13             websocket_server_2.write_message(received[0])
14         elif(received[0]=="OK"):
15             # ONLY FOR DEBUG: print(received)
16             left = []
17             center = []
18             right = []
19             if received[1]!='':
20                 left = received[1].split(",");
21             if received[2]!='':
22                 center = received[2].split(",");
23             if received[3]!='':
24                 right = received[3].split(",");
25             num_disk = len(left)+len(center)+len(right);
26             moveToDo = HanoiTowersPlanner.initProblem(num_disk, left, center, right);
27             self.write_message(moveToDo)
28             print("%sPlan Sent%s" %(GREEN,RESET))
29             print()
30         elif ...
31
32     def on_close(self):
33         print('Connection closed\n')
34
35     def on_ping(self, data):
36         print('ping received: %s' %(data))
37
38     def on_pong(self, data):
39         print('pong received: %s' %(data))
40
41     def check_origin(self, origin):
42         #print("-- Request from %s" %(origin))
43         return True
```

In this snippet of code, it is possible to see how the received strings are handled: they are initially structured to be then easily split into lists, and depending on the content, a different task is performed. Also in line 13 it is possible to see how one websocket can call the other to send information on a second connection, or call itself to send back a response as in line 27.

5.4.2 Client

The clients that are needed to provide all the necessary information to the different components of the project are three. First of all, Pepper is a client, since it receives information from the web application about the actions performed by the user (valid move, rules' violation, reached victory). Secondly, the web application is a client itself, since it receives the move performed by Pepper. Finally, another client is used to inform Pepper to start the quiz after the user has pressed the button, that will then send back the command to

change window and go to the game page. All these clients are very similar, since the same steps are executed in the same order and all the information they get is received through the server described in the previous section. Specifically, it is necessary to establish a connection to a specific IP and port and then always define a class in which functions are structured to handle the connection. An example of a client is the following Javascript code from the interactive website:

```
import * as GAME from './game.js'

// log display function
function append(text) {
    console.log(text);
}

// websocket global variable
var websocket = null;

export var connessioneStabilita = -1;

export function wsrobot_connected() {
    var connected = false;
    if (websocket!=null)
        console.log("websocket.readyState: "+websocket.readyState)
    if (websocket!=null && websocket.readyState==1) {
        connected = true;
    }
    console.log("connected: "+connected)
    return connected;
}

export function wsrobot_init(port) {
    var ip = sessionStorage.getItem("ip_pepper") //"172.16.187.128" "127.0.0.1" "127.0.1.1"
    var url = "ws://"+ip+":"+port+"/websocketserver";
    console.log(url);
    websocket = new WebSocket(url);

    websocket.onmessage = function(event){
        append("message received: "+event.data);
        GAME.moveForPlanner(event.data);
    }

    websocket.onopen = function(){
        connessioneStabilita=1;
        append("connection received");
    }

    websocket.onclose = function(){
        append("connection closed");

    }

    websocket.onerror = function(){
        window.alert("Connection problems! Returning automatically to the main menu")
        location.href='./main.html'
        append("!!!connection error!!!");
    }

}

export function wsrobot_quit() {
    websocket.close();
    websocket = null;
}

export function wsrobot_send(data) {
```

```
59 if (websocket!=null)
60     websocket.send(data);
61 }
```

As can be seen, the functions defined are very similar to those of the server in their names and operation. In line 25, it is possible to notice that the passage of IP addresses from one HTML page to another is provided by the use of sessionStorages, that are valid until the browser is restarted.

6 Results

In this section the final result of our project is shown, where all the distinct parts we presented above merge in a social and reasoning agent, whose purpose is to entertain the user.
When Pepper is started for the first time, it listens for the Sonar sensors and if it detects someone in front of it and closer than two meters, it starts the interaction. In this way, if a person does not get close, thus meaning she does not want to talk with the robot, Pepper respects her personal spaces, without invading her intimate zone. When Pepper detects someone, it turns on its eyes' leds and starts the interaction by welcoming the user, simultaneously by voice and by moving its arm, as shown in Figure 9:

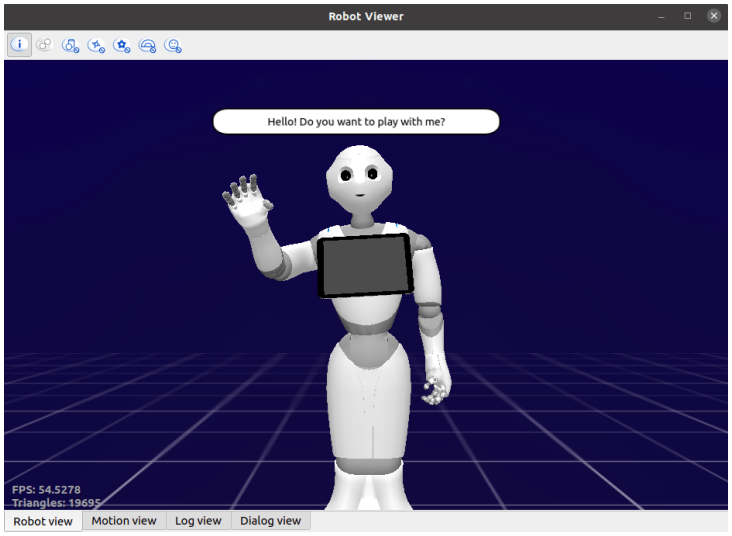


Figure 9: Initial greetings of Pepper

Pepper also tells him that it would like to ask some questions, explaining that it wants to collect data in order to suggest the most appropriate game level for the user. Immediately after, Pepper displays the *Home page* shown in Figure 10 on its tablet.

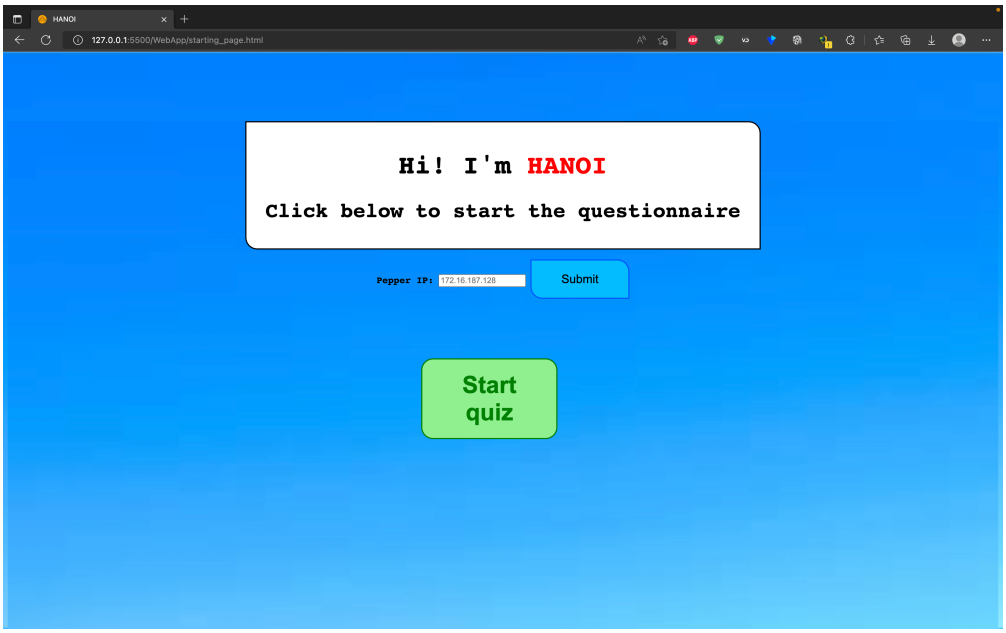


Figure 10: Home page

When the user starts the quiz, he is redirected to the *Waiting* page in Figure 11 that will remain open on the tablet until the quiz is finished or until the user decides to skip the questionnaire and go directly to play the game.

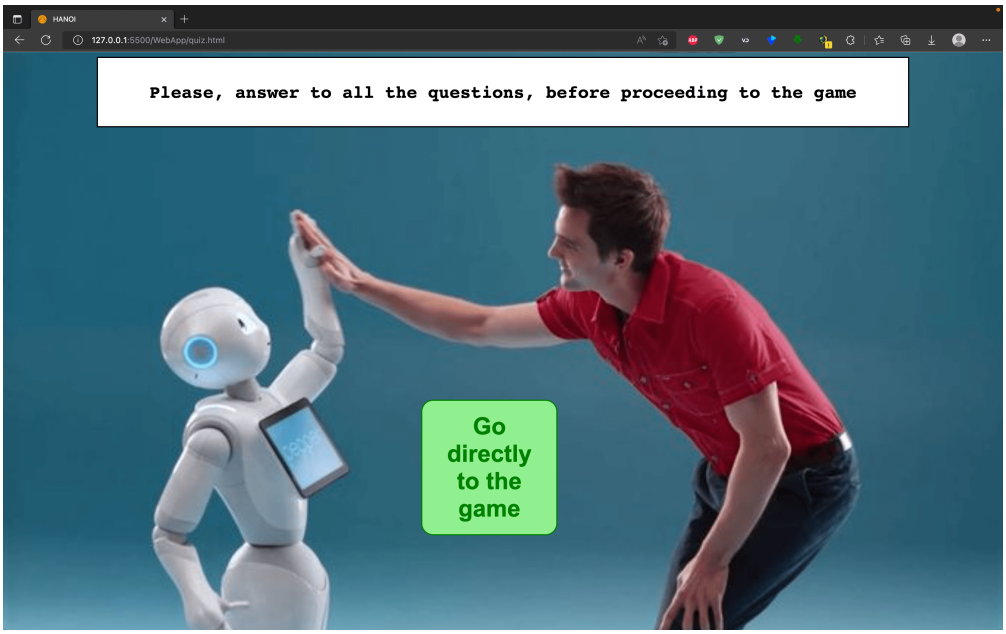


Figure 11: Waiting page during the initial questions' phase

The questions are about the user's age, his predisposition for logic games and his knowledge of the Tower of Hanoi game. Pepper will ask these questions by voice and also the user is expected to answer verbally with a simple "Yes" or "No". If a different answer is given, Pepper will tell the user that it did not understand and to simply answer "Yes" or "No". So the robot pre-processes the information, by simply counting the number of affirmative answers and, based on this number, chooses the most appropriate level in order to achieve the best user's experience and it will communicate its suggestion by voice. The user, redirected to the *Game Menu page* in Figure 12, is then free to follow the advice or to choose the level independently.

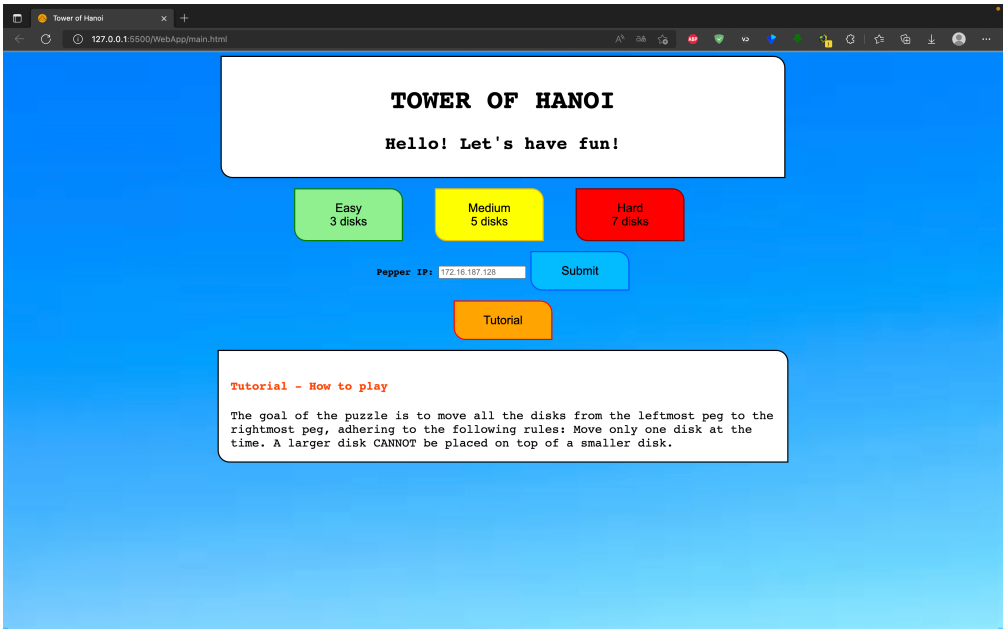


Figure 12: Game Menu

Once this first phase of interaction is concluded and the user has chosen the level to play, the robot connects to the server that puts it in communication with both the graphical user website and the planner to process the game plan. The user sees on the tablet a screen as in figure 13 where he can see the current state of the game and has buttons available to restart the game, go back to the menu, or select a rod from which to move an item in the game. For the whole time the cooperative game proceeds, the Pepper client is connected to the server and waits for several stimuli in order to make Pepper interact with the user. In fact if the user violates a game rule or tries to move a piece from an empty rod, Pepper alerts the user of the mistake through a vocal interaction; when Pepper ends his turn, he alerts the user that it is his time to make a move. The client-server connection is needed also in order to send to the server the update of the game situation after the user made his move, so that it is sent to the planner as initial condition of the planning problem. When the planner has computed the plan towards the goal, it sends the first action of the plan to the server, thatin turn sends it to the web page. At this point, the latter can replicate the move graphically, through an animation. Also, during the robot's turn, we have implemented a movement that mimics to the user the direction in which Pepper has decided to move the object. If Pepper moves a disk to a position further to the right, it will raise its right arm and wave its hand from right to left so that the user sees the action mirrored from left to right. This can be seen in the following Figure 14.

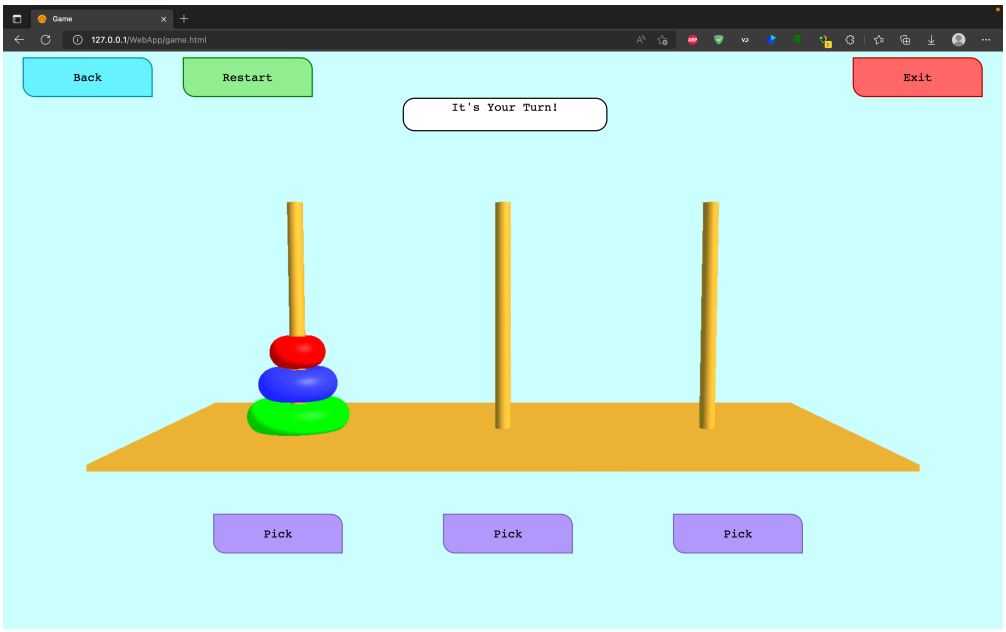
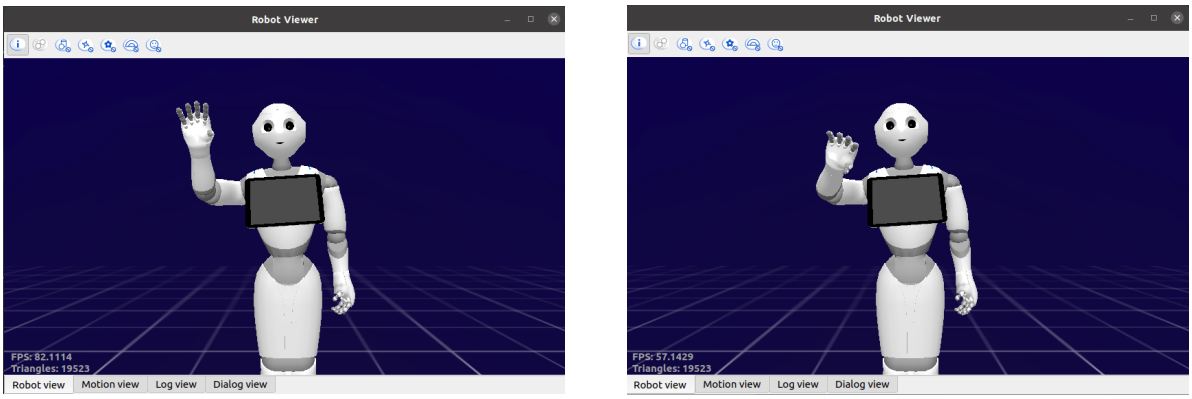


Figure 13: Opening Game page. (Easy level)

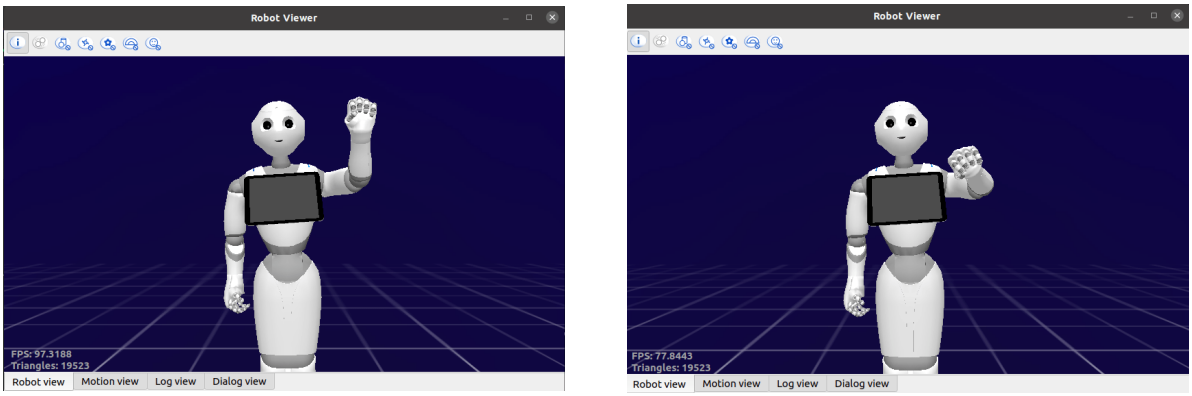


(a) First Step

(b) Second Step

Figure 14: Pepper shows movement with his right hand

Instead, in the case where Pepper moves a disk towards left, it will make a similar action to the previous one, but with its left arm and in a specular way. This is shown in Figure 15.



(a) First Step (b) Second Step

Figure 15: Pepper shows movement with his left hand

When the cooperative actions of the two agents manage in moving all the disks on the rightmost rod, the game is ended and the banner at the top of the web page writes "VICTORY", as in Figure 16. Pepper

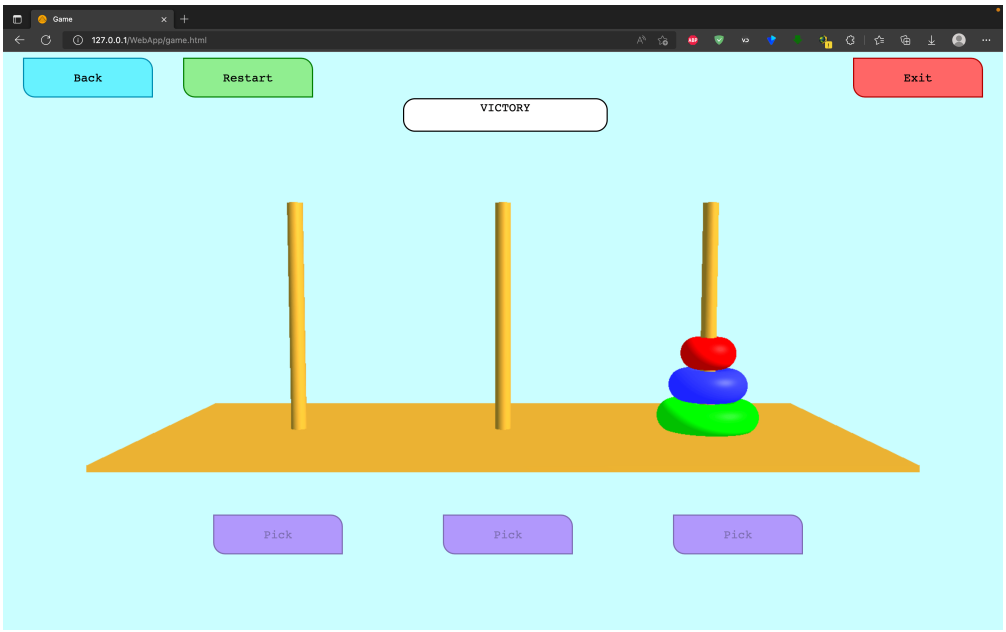
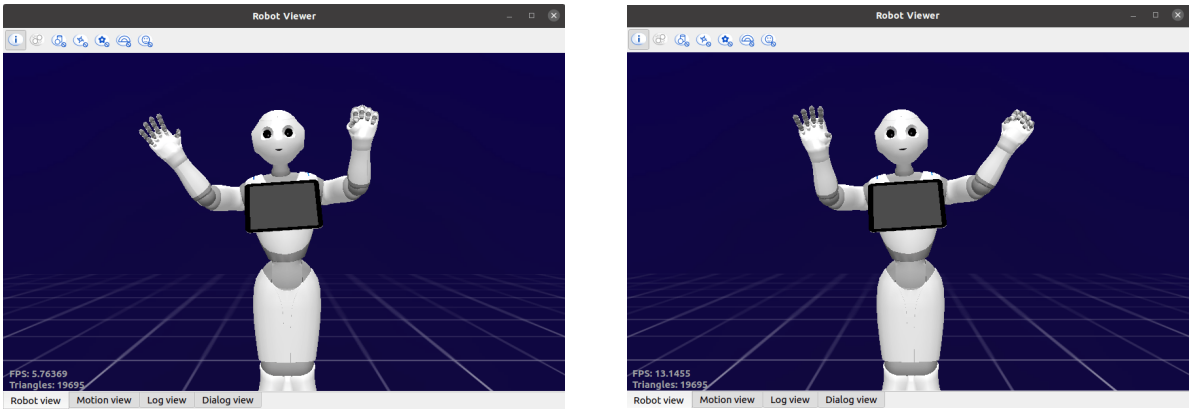


Figure 16: Game page. Victory situation (Easy level)

notifies the user of the victory (through tablet and voice) and does a little celebration dance. This animation consists in Pepper raising both arms and waving them for a couple of seconds in sign of victory, as shown in Figure 17.



(a) First Step (b) Second Step

Figure 17: Pepper does the victory dance

Finally, the robot asks the user to rate his experience and redirects him to the *Rating page* in Figure 18, where he can judge his experience with Pepper and the difficulty he met while playing. He can answer by selecting from 1 to 5 stars or squares respectively.

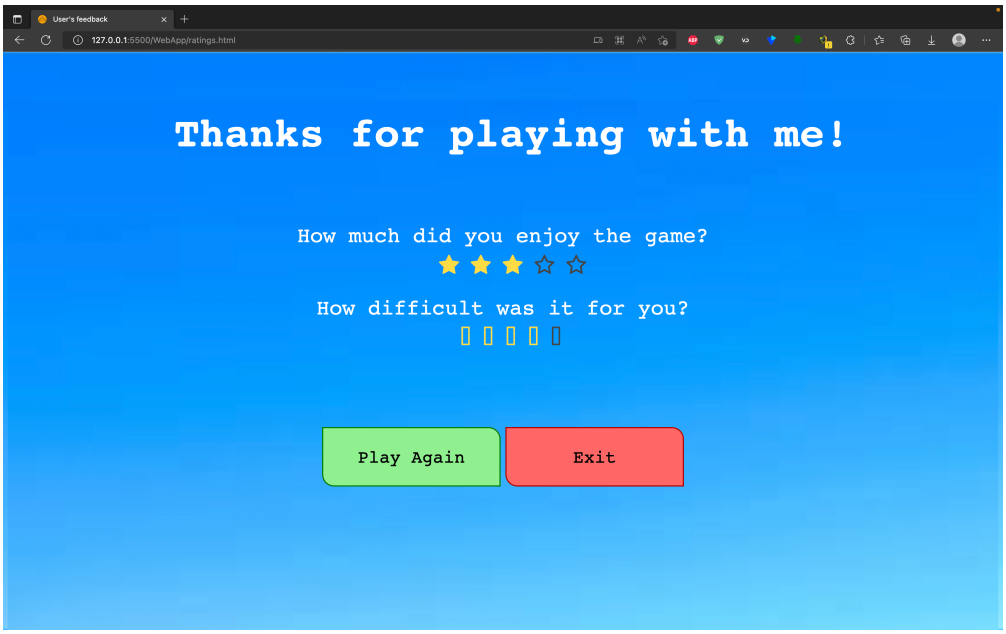


Figure 18: Rating page

From this page, the user can also stop interacting with Pepper, pressing directly the "Exit" button, that brings to the Home page, where a new user can be initialized. Instead, if he wants to keep playing, he will press "Play again". In the case he has indicated the difficulty he met, the site will redirect him to the Game page with a level that depends on the scores indicated by the user. On the contrary, if no user's opinion was given, the site will redirect the user to the same level of the previous game.

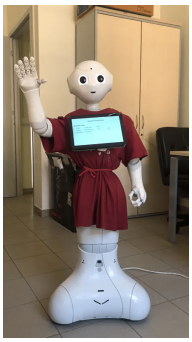
6.1 Adaptation to the Physical Robot

The developed software has been tested also using the physical Pepper robot. The adaptation simply consisted in changing the IP addresses to make the different components communicate effectively. Unfortunately, due to conflicts between Pepper's tablet browser, which is quite old, and our JavaScript code, it was not possible to make the web application run correctly on Pepper's tablet. In particular, the handling of the buttons has been done in a way that is not supported in the tablet's browser, and therefore it is impossible to use the buttons to change windows. However, by interacting with the website from an external computer it has been possible to better appreciate the interaction with the robot, by listening to it talking and watching it move. In Figure 19, we can see the gestures we had tested on the simulator performed by the physical robot. Additionally, in figure 19f we show some of the gestures that Pepper performs while talking. These make the interaction much more natural. Every time a gesture is performed, Pepper is then brought back to the "normal" posture, as in figure 19c, with the arms abandoned along its body, in order to not overheat its actuators.

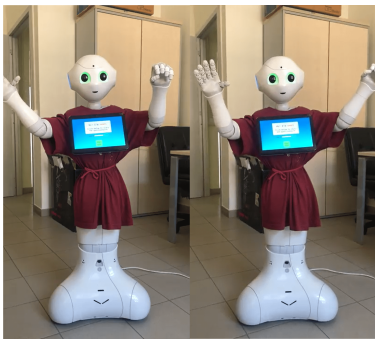
The results obtained using the physical Pepper robot are shown in a demonstrative video that is publicly available at the following link: `https://www.youtube.com/watch?v=XQVYMuwKDL8`.

7 Conclusions

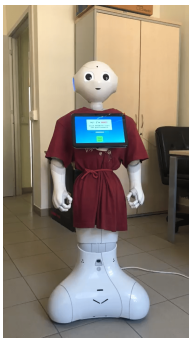
Technologies related to the world of artificial intelligence and robotics have been evolving exponentially in recent years, and the trend is not likely to decline. Such efforts are providing effective ways for helping people in several fields, such as health, entertainment, education, and many others, even enabling machines to perform complex or dangerous tasks by themselves. Our project consists in making a Pepper robot, named Hanoi, able to perform the reasoning needed to solve the Tower of Hanoi game and to do this by taking turns with a human user. We have provided different modalities to make the user interact with the robot, specifically verbally and through its tablet. Moreover, given the importance of interpersonal distances in social interactions, we make sure that the robot starts an interaction only when a person gets close to it, thus manifesting his desire to play. At the same time, the robot stays still and does not get closer itself, otherwise it would risk invading the user's intimate zone. Finally, we make the robot collect information about the user to build a simplified user's profile that allows to successfully interact with people of different ages. This profile gets also updated by making the user rate the difficulty of the level he has previously played, so that he does not get bored if the game is too easy or discouraged if it is too hard. Although the user is advised by the robot concerning the level that is most suited to him, he is always free to make his own choices. In any moment, he can also end the interaction by clicking a button. When this happens, the robot becomes ready to play with a new user. All these accouterments have allowed us to achieve a natural and successful interaction between the robot and the user.
The implementation of all these functions made us learn how to create and program an autonomous and efficient reasoning agent. It also prompted us to think critically about what were the best functional and natural level interactions to be implemented between the human and the robot, without, however, creating physical harms to people or the environment. The programming of the reasoning agent was facilitated by the library used, which allowed us to more easily and immediately define the entire problem domain. The most difficult part was definitely the programming of Pepper's motor interactions since it has multiple joints of different types and finding the most correct combination of each movement, both technically and in terms of safety, required numerous attempts.
We find the achieved results very satisfying, but they could still be improved. For example, concerning the HRI part, it would be possible to add a facial recognition system, so as to allow greater accuracy in recognizing people and to make sure with greater confidence that they are looking in the direction of the robot and are not just passing by. If the robot was able to recognize a user who already played, Pepper



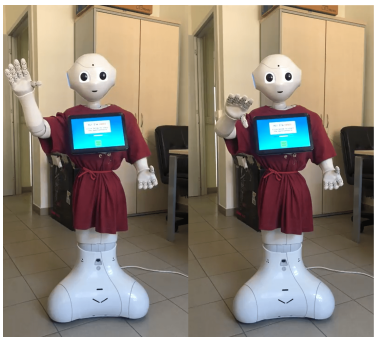
(a) Greetings



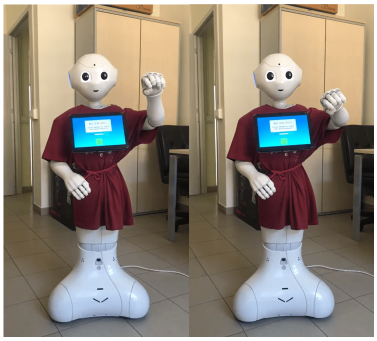
(b) Victory dance



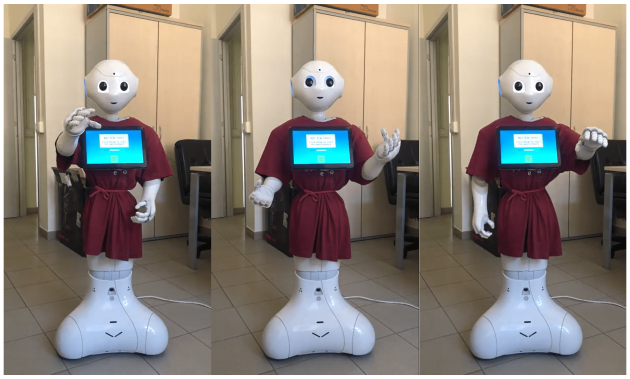
(c) Normal posture



(d) Moving a disk from right to left



(e) Moving a disk from left to right



(f) Speaking gestures

Figure 19: Physical Pepper robot's gestures.

could also save the progress in playing keeping a database so that if a user returns to play, he does not have to answer the questionnaire again and can automatically start from levels that are more complex and suited to his abilities. Furthermore, as for the Reasoning Agents part (RA), a possible expansion could be to add new games. In fact, it is possible to define different planning domains and let the planner find the solution to the planning problem of more complex games. This would greatly increase Pepper's intrinsic intelligence and could make people who look for harder challenges develop more interest in this entertainment robot.

References

[1] Three.js. `https://threejs.org`. Accessed: 2022-07-15.

[2] Three.js fundamentals. `https://threejsfundamentals.org/`. Accessed: 2022-07-15.

[3] Aiplan4eu. `https://www.aiplan4eu-project.eu`. Accessed: 2022-07-15.

[4] Naoqi. `http://doc.aldebaran.com/2-1/index.html`. Accessed: 2022-07-15.

[5] Qisdk. `https://qisdk.softbankrobotics.com/sdk/doc/pepper-sdk/index.html`. Accessed: 2022-07-15.

[6] Websocket in python. `https://pypi.org/project/websocket-client/`. Accessed: 2022-07-15.

[7] Websocket in javascript. `https://it.javascript.info/websocket`. Accessed: 2022-07-15.

[8] Kerstin Dautenhahn and Aude Billard. Games children with autism can play with robota, a humanoid robotic doll. In *Universal access and assistive technology*, pages 179–190. Springer, 2002.

[9] Henrik Hautop Lund. Modular playware as a playful diagnosis tool for autistic children. In *2009 IEEE International Conference on Rehabilitation Robotics*, pages 899–904. IEEE, 2009.

[10] PIRITA Ihamäki and K Heljakka. Social and emotional learning with a robot dog: technology, empathy and playful learning in kindergarten. In *9th annual arts, humanities, social sciences and education conference*, pages 6–8, 2020.

[11] Amazon astro presentation. `https://www.theverge.com/2021/9/28/22697244/amazon-astro-home-robot-hands-on-features-price`. Accessed: 2022-07-15.

[12] Alessandro Vinciarelli, Maja Pantic, and Herve Bourlard. Social signal processing: Survey of an emerging domain. *Image and Vision Computing*, 27:1743–1759, 11 2009.

[13] Martin Mason and Manuel Lopes. Robot self-initiative and personalization by learning through repeated interactions. pages 433–440, 03 2011.

[14] Tornado. `https://www.tornadoweb.org/en/stable/`. Accessed: 2022-07-15.