Elective In Artificial Intelligence - HRI and Reasoning: Project Report

Luca Polenta 1794787
Michela Proietti 1739846
Sofia Santilli 1813509

July 15th, 2022

Contents

1 Abstract **3**

2 Introduction **3**

3 Related Works **4**

4 Proposed Approach **5**

5 Implementation **6**

5.1 Adopted tools 6

5.2 Pepper Interactions 7

5.3 Websocket Communication 11

5.3.1 Server 11

5.3.2 Client 13

5.4 Reasoning: AIPlan4EU Unified Planning framework 14

5.5 Website for User Interaction 16

6 Adaptation to the Physical Robot **20**

7 Results **21**

8 Conclusion **22**

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. Several frameworks were exploited, including pepper_tools, AIPlan4EU, three.js and additional web development tools in order to carry out the robot's relational model, its intelligence to solve the game and a visual communication interface for the user in order to interact with the game.

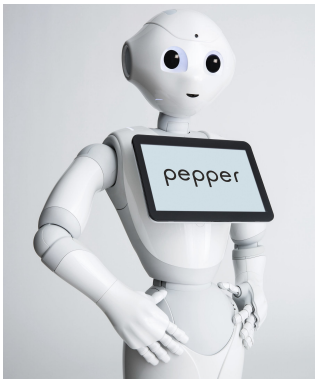
2 Introduction

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 the daily lives of every person, 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. There are several studies and applications in which intelligent robots are being applied to the medical field for assistance purposes [1] or at school to support teachers in their task of educating new generations [2]. Despite the excitement produced by such innovative uses of AI, the lastly mentioned paper highlights the fact that it is not possible to fully replace humans by AI agents in many crucial tasks, and therefore an integration an interaction between the two parties is necessary to achieve the end goal in the most optimal way. The project that is discussed in this paper aims at showing how a human and a robot can interact in a natural way to achieve a predefined purpose. Specifically, a playful robot has been developed with the purpose of entertaining the human by playing an interactive game in which the human and the robot make moves alternately to achieve the victory. In addition, a series of verbal interactions were included to make the robot warn the user about possible mistakes or communicate him when it is his turn to play.

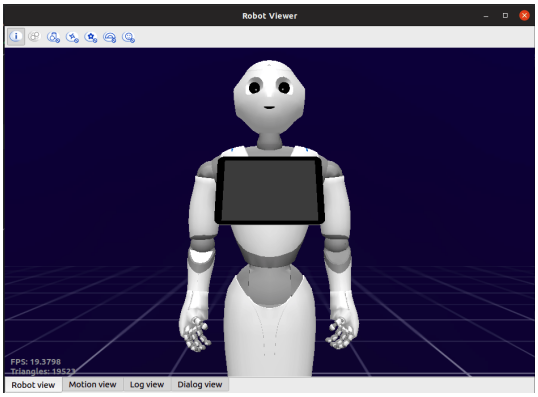
This project has been mainly developed through Python and web development tools. Web development tools, including frameworks such as three.js [3] [4] and jQuery, allowed the creation of a graphical environment in which the user can visualize the progress of the game and interact to make the next move for solving the problem. These frameworks were 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. Additionally, NAOqi [5] 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.

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, the QiSDK [6], 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 can be seen at figure 1b. Furthermore, a Python framework called AIPlan4EU [7] has been exploited to execute the planning in order to make the robot able to correctly choose the best next move to make.

Finally, universal websockets [8] [9] have been used to ensure communication between the planner in Python, the robot software, and the website managed by JavaScript.



(a) Pepper Robot



(b) Virtual Pepper in Android SDK

3 Related Works

The use of intelligent robots for playing games is not a novelty. Multiple studies have already demonstrated their real and effective usefulness both in the case of assistive robots for particular pathologies, such as autism [10] [11], and in more common cases, such as in pet robots [12]. Concerning assistive robots for people affected by autism, the first paper we have mentioned proposes the employment of small humanoid robots to help in performing therapy for people with autism. Differently, in the second paper the use of modular robots has been proposed in order to record the interaction of the autistic user and, by associating it with his or her medical record, it is possible to develop further diagnosis and insights in order to improve the therapies to be administered. However, robots and artificial intelligence are not only exploited in applications that are strictly medical. For instance, there have been studies aimed at tracing the potential links between the development of empathy and social and emotional learning when using a robot dog as a learning tool in the education of preschoolers in kindergarten. In fact, it has been shown that human interaction with animals lead to greater externalization of feelings and bonds that are usually more difficult to demonstrate and study. The use of social robots, such as toy dogs, offer designers new opportunities to rethink areas of change affecting how children relate to their peers (nurturing), learn (empathy), and play (social interaction).
Despite their proven usefulness, these applications present both hardware and software limitations. Concerning software components, studies [13] have been done to make robots intelligent through cloud computing. Through this approach, each robot is only required to have the physical features that are needed to execute the addressed tasks while its intelligence is provided by much more powerful computers that are located in farms scattered around the world. This would allow for high scalability and upgradeability of each robot. Two early cases 2 of this large-scale expansion in a domestic and popular setting are given by Bubble Robot 2a, produced by Clementoni, and Astro, created by Amazon 2b. The former is an interactive robot that is not really intelligent, but nevertheless marks a great change in society and in the use that is made of robots in it. In fact, this 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. Amazon Astro, on the other hand, is a more concrete robot assistant capable of following the human and interacting with him constantly, partly thanks to its high integration with Alexa [14]. This virtual assistant entails high scalability and intelligence that allow it to perform highly difficult tasks and play games. Currently, its only limitation is purely of physical nature, since it has no limbs or arms to make it capable of performing complex interactions with the environment. Nevertheless, consiering Amazon's high interest in the field, its future evolution cannot be ruled out.

(a) Bubble Robot from Clementoni

(b) Astro from Amazon

Figure 2: Cases of robots applied to everyday use

The increase in the use of robots in such diverse applications have underlined the need for more complex interactions. Naturaleness is also an aspect that needs to be improved to make humans more willing to interact with robots in everyday life tasks. To this aim, [15] 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 of 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. For instance, one of the first and most important social signal that manifests itself in a social interaction is interpersonal space. In fact, the interpersonal space defines the type of relationship between people and 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 it 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 naturaleness, [16] 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 of acting proactively.
These insights have been used in our project to make the interactions between the user and the robot as natural and effective as possible.

4 Proposed Approach

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 is basically divided into three parts:

- An initial interaction 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;
- A playing phase, in which the robot and the user alternately have to make a move to solve the game and achieve victory;
- A final interaction in which the robot asks the user to rate the game and to say whether the game was to hard or too easy. 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 will be adapted accordingly.

In order to start the interaction, we take the considerations made by [15] concerning the interpersonal space into account. In particular, we start an interaction with the user just if he gets closer than a threshold, thus manifesting his desire to play. As in [16], 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. More details about our implementation are given in the next section.

5 Implementation

This section provides a detailed explanation of the whole implementation of the project, from the tools and frameworks that have been used to the reasons motivating our choices.

5.1 Adopted tools

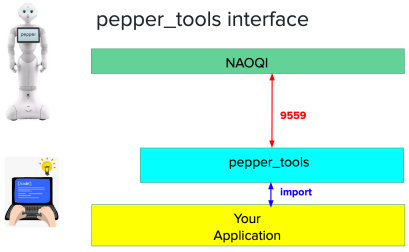
This project has been developed in a Linux environment, in particular using Docker, which allows creating a container where it is possible to work independently from the rest of the system. In this way, any change made on or by the code toward Pepper's environment or framework are not permanently applied, 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
```

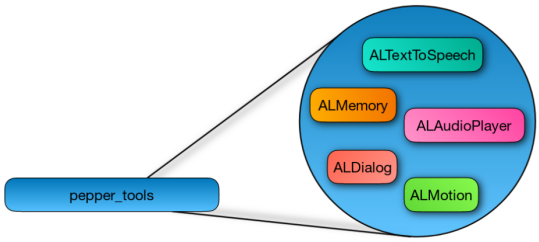
and run such image to enter the architecture of your robot:

```
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 3a. Interactions between the human and the robot have been implemented using *pepper_tools*. These allow to easily call different pre-implemented functions, shown in Figure 3b, to make Pepper perform different tasks, such as making it say something, make it move or use its sensors for different purposes.



(a) NAOqi Architecture



(b) Details of the pepper_tools

Among the most popular modules, ALTextToSpeech is 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. Motor animations can also be associated with the voice playback so as to increase user interest, involvement and interaction. ALAudioPlayer provides playback services for several audio file formats. ALMemory is a centralized memory used to store all key information related to the robot's hardware configuration. ALDialog is a module for equipping the robot with conversational capabilities using a list of appropriately written and categorized "rules." In the end, the ALMotion module provides methods that facilitate the robot's movements. There are also many other modules to access data collected by Pepper's sensor or to implement several of its functionalities. Only some of these routines have been used in this project. All the code that interacts with the docker was developed in Python.
The graphic part of the game, instead, consists of a web application developed in HTML/JS. Consequently, its use is not limited to Linux environments, but it can be run also on other operating systems, such as Windows and macOS, through a browser. Finally, the interaction between various languages, softwares and frameworks is provided by the use of universal websockets for passing key information in order for each component to function.

5.2 Pepper Interactions

When Pepper is started for the first time, it listens for the Sonar sensors in order to detect if someone is in front of it, in particular closer than two meters. This is done by assuming that a person that do not get close to the robot is not willing to start an interaction. When this happens, Pepper turns on its eyes' leds and starts the interaction by welcoming the user, both through the vocal systhesis of a sentence and by a movement of the right arm that rises and mimics a greeting. The sentence is executed asynchronously with respect to the movement, so they occur simultaneously as shown in Figure 4:

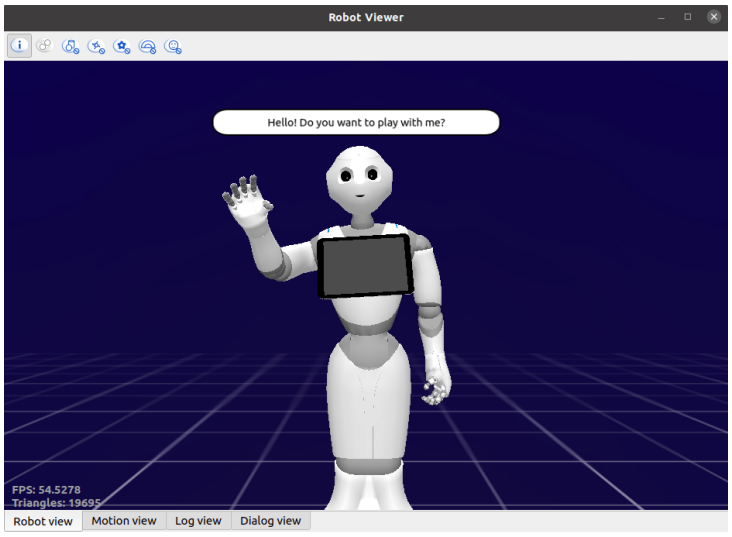


Figure 4: Initial greetings of Pepper

When the user starts the quiz, Pepper explains that the questions it is about to ask have the aim of allowing it to collect data in order to suggest the most appropriate game level for the user. 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". Once the robot has preprocessed these information by following a series of pre-established rules, by counting the number of positive and negative answers it is able to choose the most appropriate level in order to achieve the best user's experience and it will communicate its suggestion by voice. The user is then free to follow the advice or to choose the level independently.
Once this first phase of interaction is concluded and the user chooses 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. This connection will be explained in details in the next section. 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:

- 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;
- When the game ends, Pepper notifies the user of the victory (through tablet and voice) and does a little celebration dance.

Finally, the robot asks the user to rate his experience and, depending on the difficulty he found in playing that level, the user's profile is updated. If the user wants to keep playing, he is redirected to the game page corresponding to the appropriate level based on the new profile.

In order to have Pepper perform all these functions, a python program has been developed. First, Pepper is accessed via *pepper_tools* utilities. Once a connection is established with the physical robot or with the robot simulator, we interact with Pepper in two ways: either through pre-defined functions or by structuring our own interaction functions from scratch. In the first case, pre-defined functions are defined within the *pepper_tools*' pepper_cmd.py file and are used to perform some basic actions such as some voice interactions:

```
            if(received[0]=="RuleViolation"):
                print("%s!!RuleViolation!!%s" %(RED,RESET))
                pepper_cmd.robot.say('You violated the game rules. Try again.')
            elif(received[0]=="EmptyRod"):
                print("%s!!EmptyRod!!%s" %(RED,RESET))
                pepper_cmd.robot.say('You chose an empty rod. Try again.')
            elif(received[0]=="ActionDone"):
                print("%s!!ActionDone!!%s" %(GREEN,RESET))
                pepper_cmd.robot.say('Now it\'s your turn.')
```

Or they are used to turn on, turn off, and read the Sonar sensor at Pepper startup when the user needs to be located:

```
    # 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()
```

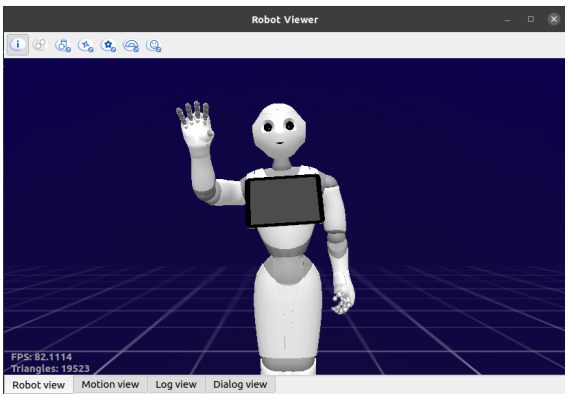
Instead, the second way in order to interact with Pepper is by structuring some interactions manually. This was necessary because sometimes the pre-defined functions did not allow much customization of the interaction, whereas in the current project it is necessary to make some vocal interaction asynchronous or to manually manage some timing in the Pepper robot's movements. To carry out these interactions, it was first necessary to call a function that returned the session environment variable, from which to call the modules that allowed us to define some custom interactions, such as the first 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()
```

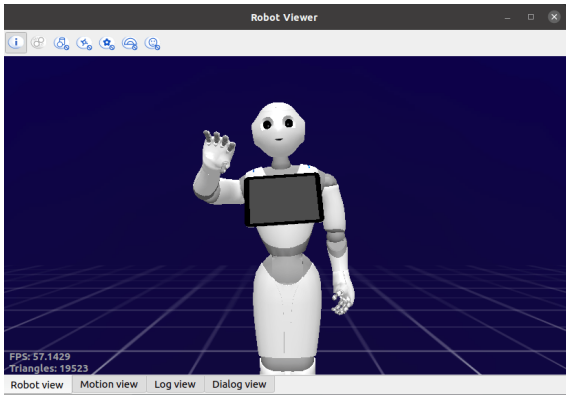
or it became necessary when the movement is defined and implemented, as we are only interested in applying it to certain components from time to time and we are interested in implementing those movements with customized timelines:

```
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
```

Finally, the additional movements that have been implemented are two: movements during the game that mimic to the user the direction in which Pepper has decided to move the object, and the final victory movement. In the case of the movement during play, if Pepper moves a disk to a position further to the right, it will raise her right arm 5 and wave her hand from right to left so that the user sees the action mirrored from left to right. This can be seen in the following figures:



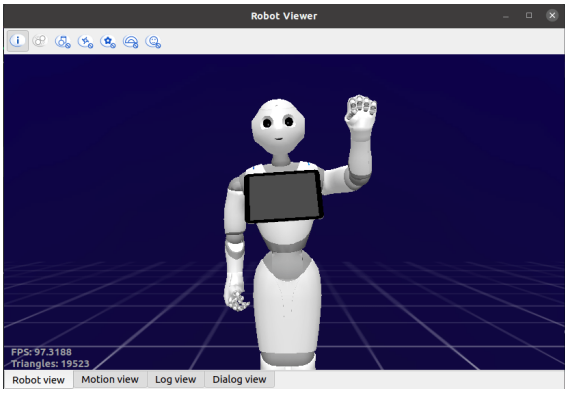
(a) First Step



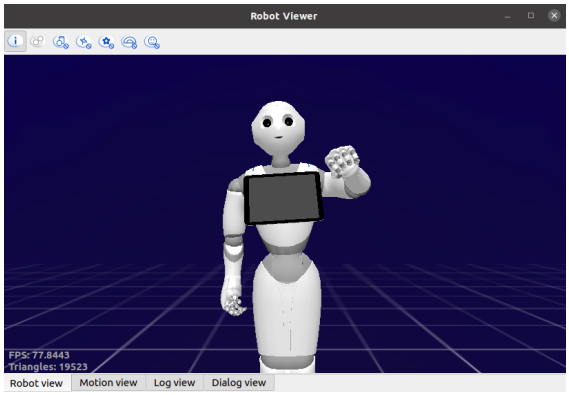
(b) Second Step

Figure 5: Pepper shows movement with his right hand

Instead, in the case where Pepper moves a disk to a position further to the left than the starting position, 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 6:



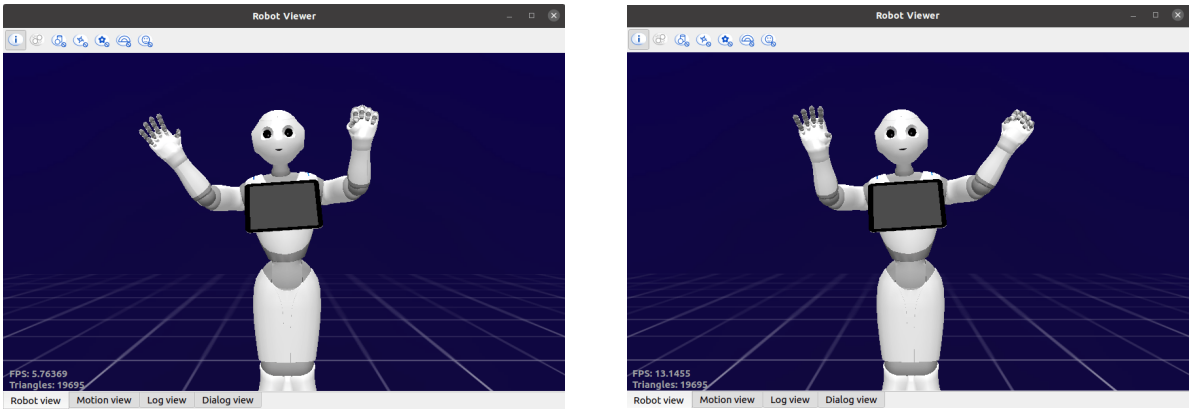
(a) First Step



(b) Second Step

Figure 6: Pepper shows movement with his left hand

Instead, the final animation consists in Pepper raising both arms and waving them for a couple of seconds in a sign of victory, as shown in Figure 7:



(a) First Step (b) Second Step

Figure 7: Pepper does the victory dance

5.3 Websocket Communication

The communication between different components within this project is carried out through the use of websockets [8] [9]. 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. In detail, two websockets were developed: one to connect the Javascript structure of the interactive Web site with the robot-related python frameworks and a second one to connect the AIPlan4EU planner with the code linked to the Pepper robot, which runs in a separate docker where the AIPlan4EU libraries cannot be called.

5.3.1 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 for 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 Web site, 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 explained above. 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 will be shown in section 5.5. The code by which the connection is generated is the following:

```
    # Run web server for HTML
    application = tornado.web.Application([(r'/websocketserver', MyWebSocketServer),])
    http_server = tornado.httpserver.HTTPServer(application)
    http_server.listen(server_port)
    print("%sWebsocket server for HTML listening on port %d%s" %(GREEN,server_port,RESET))

    try:
        tornado.ioloop.IOLoop.instance().start()
    except KeyboardInterrupt:
        print(" -- Keyboard interrupt --")


```

```
13      if (not websocket_server is None):
14          websocket_server.close()
15      print("Web server for HTML quit.")
```

To easily implement websockets, Tornado [17] was used. It is a web framework and asynchronous network library in Python which facilitates the scripting of a client-server connection. 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. In it 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.

5.3.2 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 precious 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!!!");
    }
```

```
50
51 }
52
53 export function wsrobot_quit() {
54     websocket.close();
55     websocket = null;
56 }
57
58 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 can be seen that IP passing from one HTML page to another is provided by the use of sessionStorages that are valid until the browser is restarted.

5.4 Reasoning: 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:

```
1  problem name = towersHanoi
2
3  types = [Item]
4
5  fluents = [
6    bool is_disk[disk_i=Item]
7    bool clear[disk_i=Item]
8    bool on[disk_i=Item, disk_pos=Item]
9    bool smaller[disk_i=Item, disk_j=Item]
10 ]
11
12 actions = [
13   action move(Item disk, Item l_from, Item l_to) {
14     preconditions = [
15       is_disk(disk)
16       smaller(disk, l_to)
17       on(disk, l_from)
18       clear(disk)
19       clear(l_to)
20     ]
21     effects = [
22       clear(l_from) := true
23       on(disk, l_to) := true
24       on(disk, l_from) := false
25       clear(l_to) := false
26     ]
27     simulated effect = None
28   }
29 ]
30
31 objects = [
```

[1]This library has been developed for the AIPlan4EU H2020 project[7], funded by the European Commission under grant agreement number 101016442.

```
  Item: [loc_1, loc_2, loc_3, disk_1, disk_2, disk_3]
]

initial fluents default = [
  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 that 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 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.

5.5 Website for User Interaction

The website was developed using HTML, JavaScript and css. In particular Three.js [3] [4] has been exploited. It is a JavaScript library for 3D Web graphics based on WebGL. 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. Therefore, through it, one can generate and manage complex 3D objects in a very simple way on browsers. Specifically, the graphical environment is created through a hierarchical pyramidal model where each object is connected to the others through a parent-child relation, and through 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 8:

```
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 8: Scene Structure

On the other hand, regarding the graphic and functioning of the site, the tablet initially shows the Home page in Figure 9. Here, beyond changing the connection IP of the server if needed, the only option the user has is to press the "Start quiz" button.

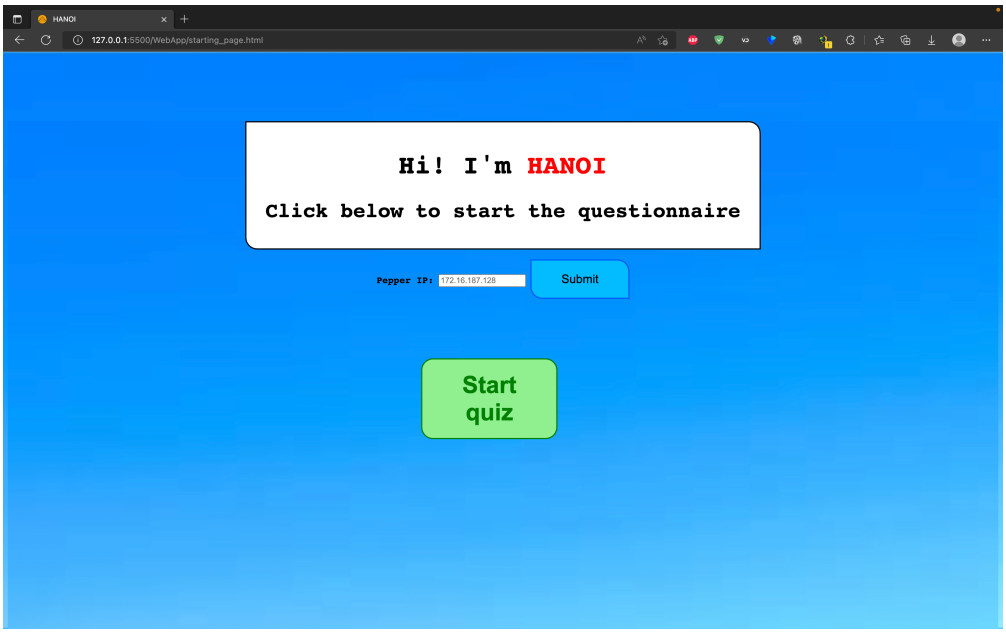


Figure 9: Home page

When the user is ready to start answering the questions, he is redirected to the waiting page in Figure 10 that will remain open on the tablet until the quiz is not finished or until the user decides to skip the questionnaire and go directly to play the game.

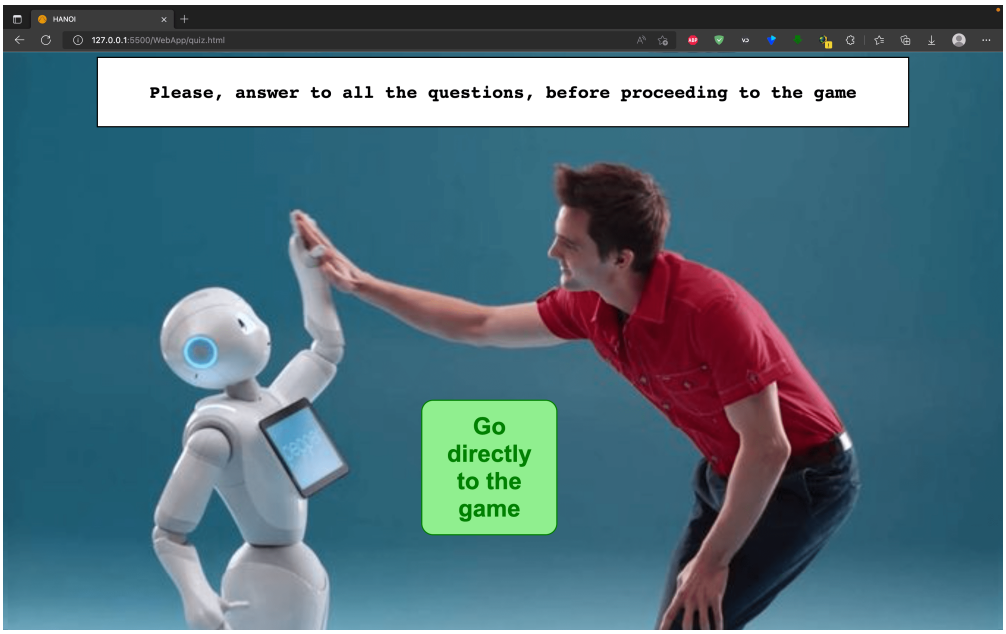


Figure 10: Waiting page during the initial questions' phase

In both cases, the user is redirected to the Game menu where he has multiple choices, shown in Figure 11.

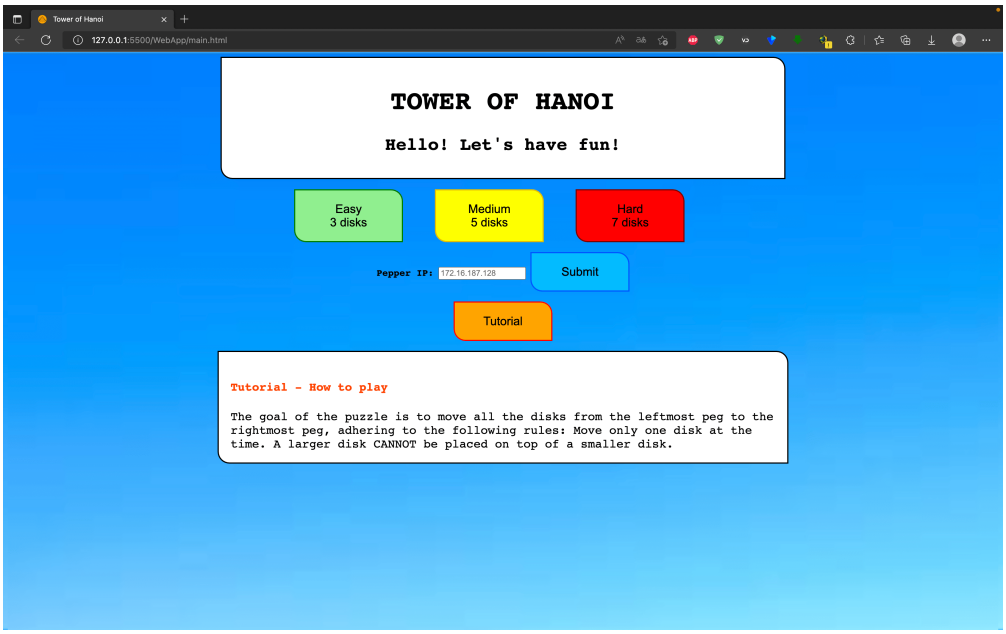


Figure 11: Game Menu

In particular, the user can select the difficulty level, change the connection IP of the server if needed, or read a tutorial on the game rules. Once a level is selected, the Game page opens. The game proposed is the Towers of Hanoi. The goal is to move all the disks from the leftmost rod to the rightmost one, adhering to the following rules: move only one disk at the time and a larger disk cannot be placed on top of a smaller disk. The user sees on the tablet a screen as in figure 12 where he can sees 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. Once a complete user made his choice, the action selection buttons change as shown in the figure 13: the one from which the item to be moved was 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. Once the user has made his choice and the functions defined within the JavaScript verify that this choice is feasible, we check whether his move is feasible with respect to the game's rules. Based on this, it alerts Pepper to say something or to process the next action. At the same time, a graphical function is called repeatedly 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 placed at. Then it is moved in the direction it needs to go, and finally, once it has arrived on the right rod, it is placed at the right height. Subsequently, once the planner associated with Pepper has processed the solution and passed the next move to the JavaScript associated with the site, this animation function is called again to animate the move proposed by Pepper. During this entire process, the rod selection buttons are disabled to prevent the user from performing further actions and they are reactivated only when the animation is complete. At the end of the animation associated with the action selected by Pepper, the user is vocally informed that it is his turn to make a move. Similarly, if the user makes some mistakes, Pepper alerts his by vocal interaction. Additionally, to also provide a user-friendly game interface, a banner has been also inserted at the top of the game window, which inform the user about what is happening, as can be seen in Figures 12 and 13.

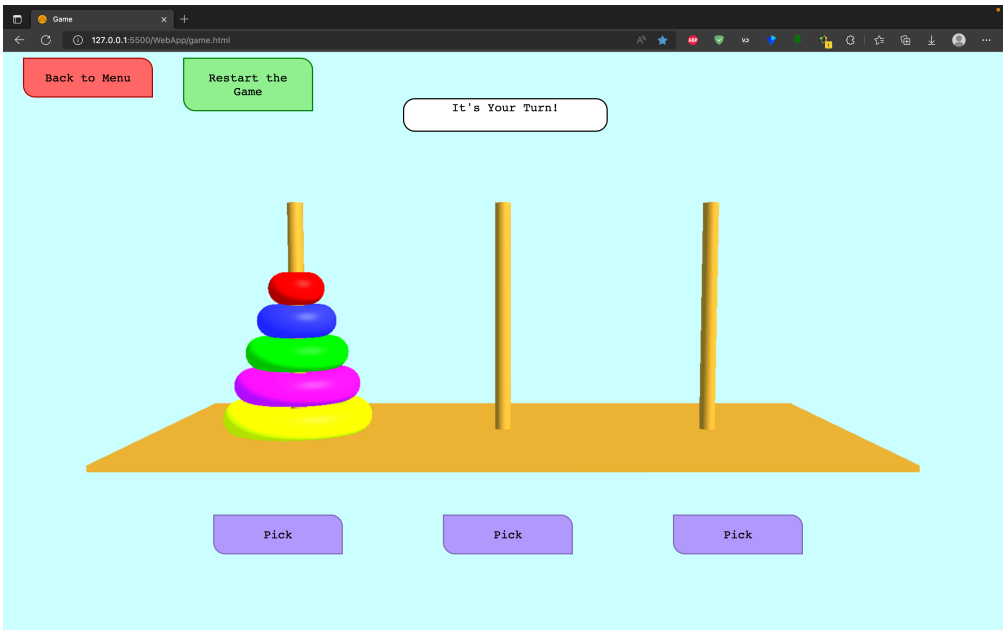


Figure 12: First Situation in the Game. (Medium level)

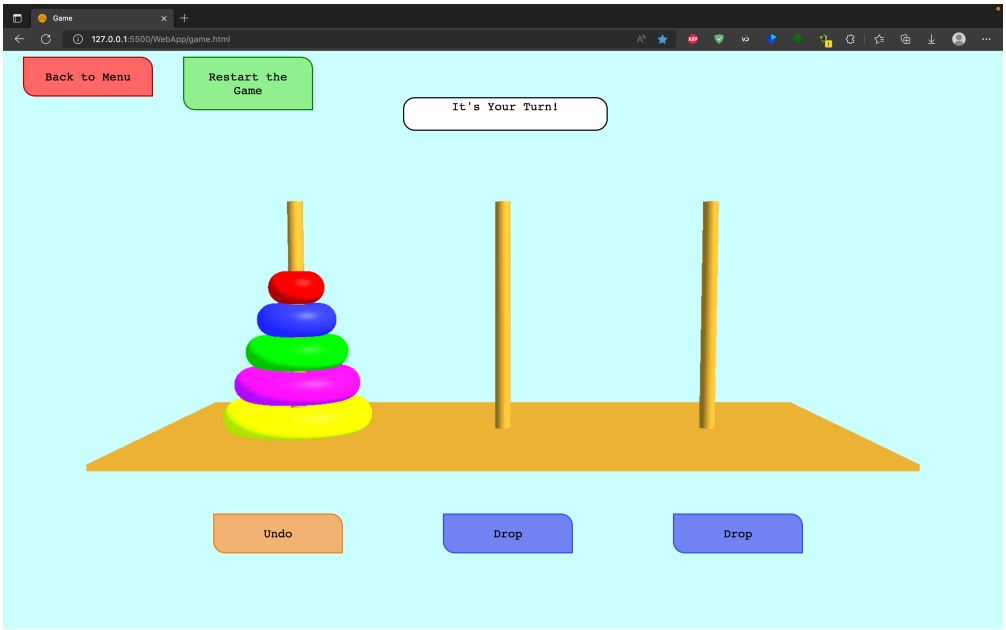


Figure 13: Second Situation in the Game. The user made his choice. Now he can drop the disk on another rod or undo the selection of the disk. (Medium level)

The banner also writes "VICTORY" in the case the cooperative work of the two agents is successful in moving all the disks on the rightmost rod. At this point, the user is redirected to the 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.

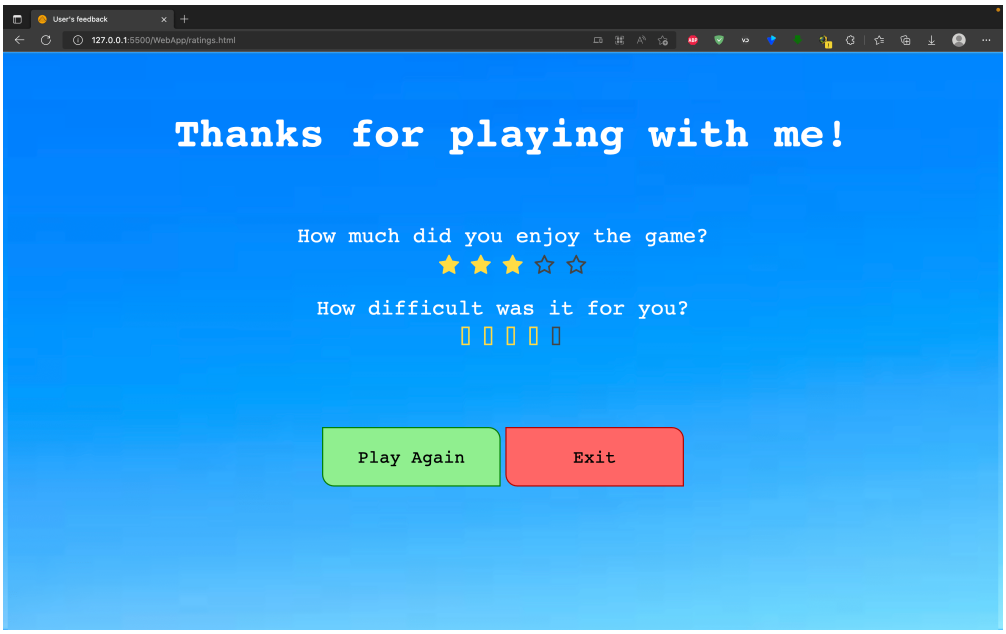
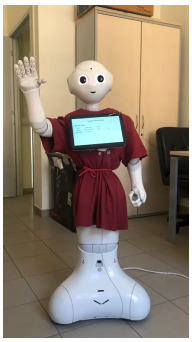


Figure 14: Rating page

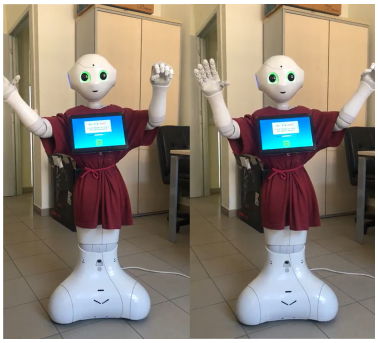
From this page, the user can also stop interacting with Pepper, pressing directly the "Exit" button, that brings to the Home page, where a new user can be initialized. Instead, if he wants to keep playing, he will press "Play again". In the case he has indicated the difficulty he met, the site will redirect 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 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 15, we can see the gestures we had tested on the simulator performed by the physical robot. Additionally, in figure 15f 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 15c, with the arms abandoned along its body, in order to not overheat its actuators.

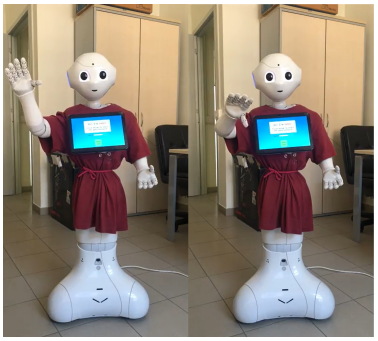


(a) Greetings

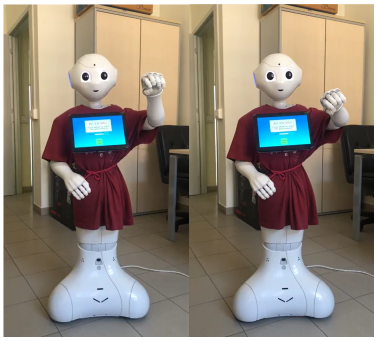


(b) Victory dance

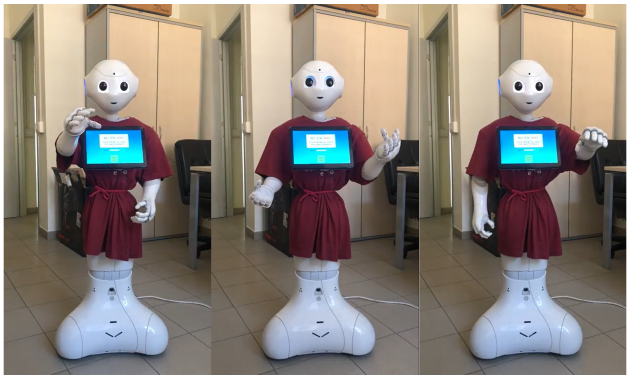
(c) Normal posture



(d) Moving a disk from right to left



(e) Moving a disk from left to right



(f) Speaking gestures

Figure 15: Physical Pepper robot's gestures.

7 Results

The results obtained were also tested on a physical Pepper, and the demonstration video is publicly available at the following link: DA AGGIUNGEREEEE.
In addition, it should be pointed out that, given the generation gap between the new standard web technologies and those available in the obsolete browser built into the Pepper's operating system, the tablet interaction part was done in a second computer instead of directly on the Pepper's tablet. In any case, however, new generations of robot Pepper that have more up-to-date software will be able to have the graphical interaction part directly on the robot itself.

8 Conclusion

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 to help people in many fields, such as health, entertainment, education, and many others, even enabling machines to perform complex or dangerous tasks exclusively. With our project we wanted to demonstrate and implement a human-robot interaction that would allow a Pepper robot, named Hanoi, to interact with people of any age in order to entertain them. Indeed, given the amount of verbal, physical, and playful interactions, we believe we have successfully achieved our goal. This was made possible also thanks to the excellent software provided by Professor Luca Iocchi and Professor Fabio Patrizi of the DIAG department at the University "La Sapienza" of Rome, their colleagues and collaborators who provided the AIPlan4EU planner, and Softbank Robotics who provided Pepper robot and related software tools.

References

[1] Kyong Il Kang, Sanford Freedman, Maja J Mataric, Mark J Cunningham, and Becky Lopez. A hands-off physical therapy assistance robot for cardiac patients. In *9th International Conference on Rehabilitation Robotics, 2005. ICORR 2005.*, pages 337–340. IEEE, 2005.

[2] Anant Krishna Parab. Artificial intelligence in education: teacher and teacher assistant improve learning process. *International Journal for Research in Applied Science & Engineering Technology*, 8(11):608–612, 2020.

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

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

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

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

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

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

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

[10] 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.

[11] 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.

[12] 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.

[13] Javier Salmerón-Garcı, Pablo Inigo-Blasco, Fernando Dı, Daniel Cagigas-Muniz, et al. A tradeoff analysis of a cloud-based robot navigation assistant using stereo image processing. *IEEE Transactions on Automation Science and Engineering*, 12(2):444–454, 2015.

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

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

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

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