COMP3431 Robotic Software Architecture

Assignment 2 Report

UNSW
AUSTRALIA

Dancing Nao Group

Adam Dobrzycki (z3470061)

Lancelot Chen (z5021315)

Yukai Miao (z3486098)

November 10, 2015

Abstract

This report describes the process of programming a Nao to autonomously play a dancing game. It details the design, implementation, experimental results and future improvements. The project was successful and the Nao was able to complete a game.

Contents

1 Introduction

The aim of this project was to build software to allow a Nao[3] robot to play a dancing game in front of a screen.
The Nao is an autonomous, programmable humanoid robot developed by Aldebaran Robotics, with APIs for multiple programming languages.
The dancing game targeted for this project is Dance Dance Revolution, or its various clones. In it, a player makes movements according to markers appearing on the screen. Usually these markers are arrows in in four different directions (up, down, left and right), and scroll from the bottom of the screen to the top. The player is required to use their feet to press the corresponding direction on a game pad when the arrow reaches a certain position at the top of the screen, indicated by an outline or "slot".
For this project, a free open-source clone of Dance Dance Revolution was used, called Step Mania. This game further provided the ability to make custom game maps and modify existing ones, giving precise control over the speed and type of movements presented.
A further aim was to mimic the human playing process by using the Nao's built in vision and motion systems.

2 General Design

To achieve this task, architecture design shown in Figure 1 was used.

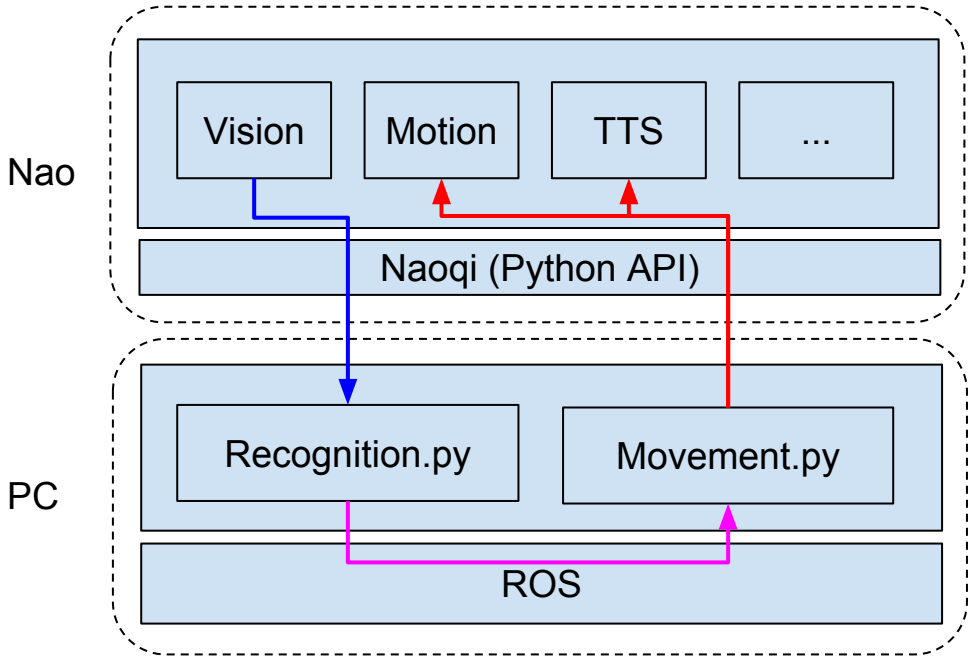

Figure 1: System Architecture

In general, the program is run on an external computer, and communicates with the Nao using the Naoqi Python API through a local network. The program contains two modules - the Recognition module and Movement module. The Recognition module is responsible for retrieving camera images from the Nao's camera's, recognising relevant signals in the images, and making movement decisions. The Movement module is then responsible for sending commands to Nao's Motion and TTS module to respond to the decisions made by the Recognition module. To transmit the movement decisions, both modules are simply wrapped as ROS[1] nodes, with the Movement module providing a ROS service which is invoked by the Recognition module when a decision needs to be transmitted.

3 Implementation

3.1 Recognition Module

3.1.1 Prerequisites

1. HD Camera. To achieve good precision, a HD camera is required. The Nao has two built-in cameras in its head, which provide a maximum $1280 * 960$ resolution at 30 frames per second[4]. One of these - the top one - is looking straight forward, while the other one is looking down, so the top camera is used to capture video frames of the screen for further processing.

2. High Speed Wireless or Cable Network. To transmit the raw HD images with at least 5 frames per second, a high speed network with at least 150Mb/s available bandwidth is needed. A dedicated wireless or cable network provides this capability. For the final testing and demonstration, a direct cable link between the Nao and computer was used.

3. OpenCV. Due to the large amounts of complex but routine image processing tasks, OpenCV is heavily used in this module to quickly build up the pipeline and to save more time for testing.

3.1.2 Processing Flow

The general processing flow in this module is illustrated in Figure 2. The detailed image processing steps for arrow detection, which is the highlighted part in Figure 2, are shown in Figure 3.
In Figure 3, the upper side steps are all routine image processing steps and can be easily done with OpenCV APIs. However the other steps are designed specifically for the arrow detection task.
Firstly, the "classifier" is used to classify the detected shapes that are similar to the template arrow into static arrow "slots" and moving "arrows". This is simply done by comparing the

[1]ROS[1] is a set of libraries for helping building components for robots.

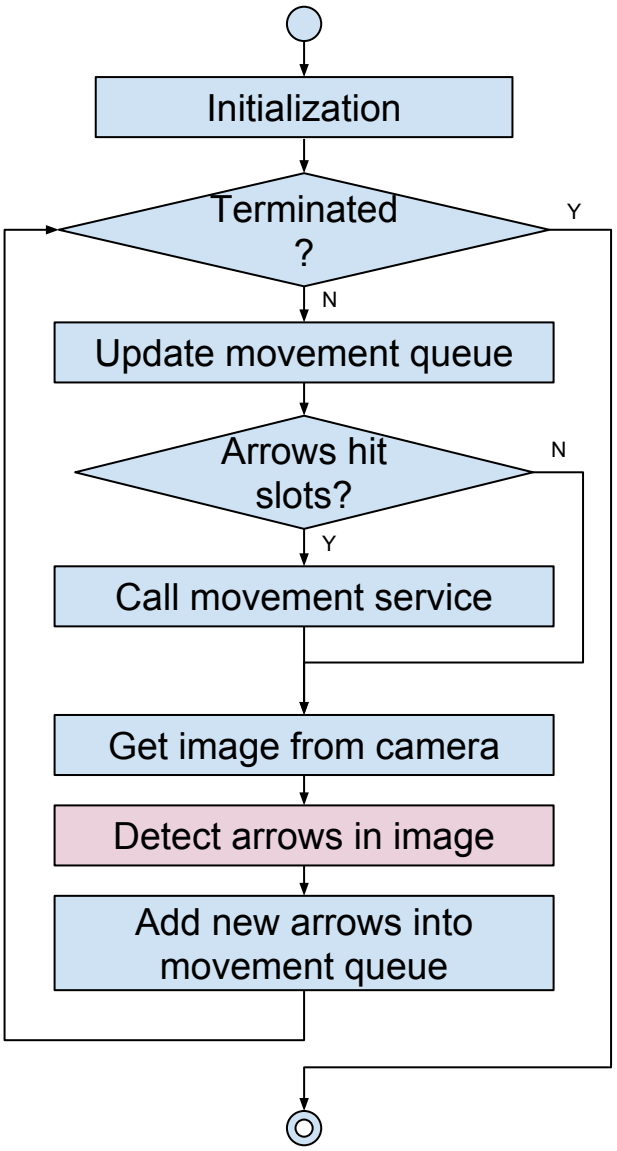

Figure 2: General processing flow in Recognition module

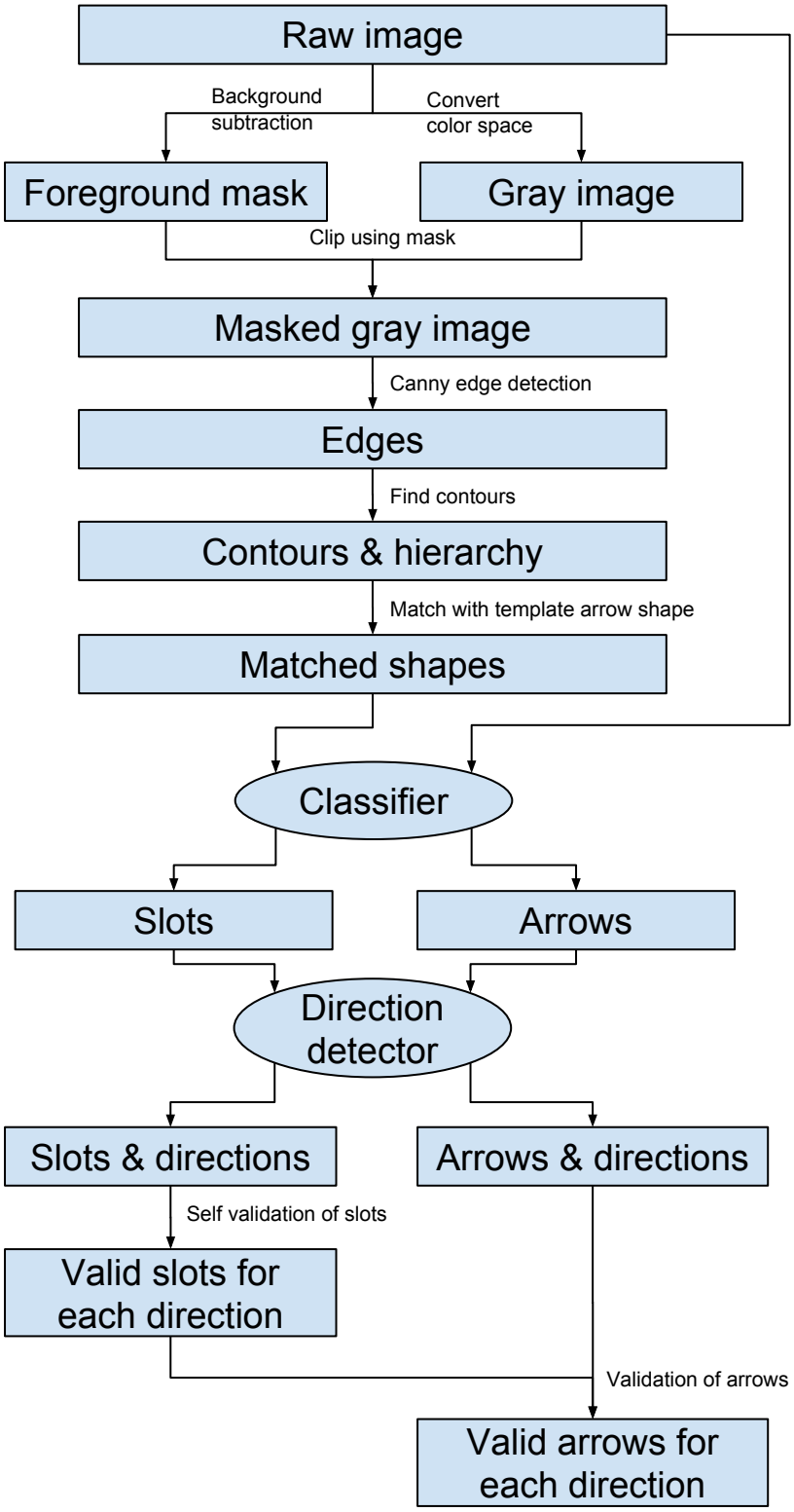

Figure 3: Image processing steps for arrow detection

saturation of the average colour in the shape to a threshold value; if it is above the threshold value, then the shape is a "arrow", otherwise the shape is a "slot".
Secondly, the "direction detector" determines which direction the arrow or slot is pointing to[2]. This is implemented based on the positions of convex hull defects in the arrow shapes, as shown in Figure 4, where red dashed lines are convex hulls, green areas are defects, and purple points are the furthest points of each defect. Although this method works in theory, care must be taken in real world applications, as the shapes may contain lots of vertices, and most of which may be noisy or redundant. As a solution to this, contour approximation provided by OpenCV is very helpful and effective.
Following this, due to the noise from the source image and possible loss of precision, the above steps could contain false negative or false positive results, so self-verification is done on the candidate slots based on their directions and relative positions. Only valid ones are kept, and based on these positions and angles for the "tracks" of each direction are decided on. Finally we also use these "tracks" to verify the candidate arrows, and only valid arrows will be taken into the next processing steps[3].

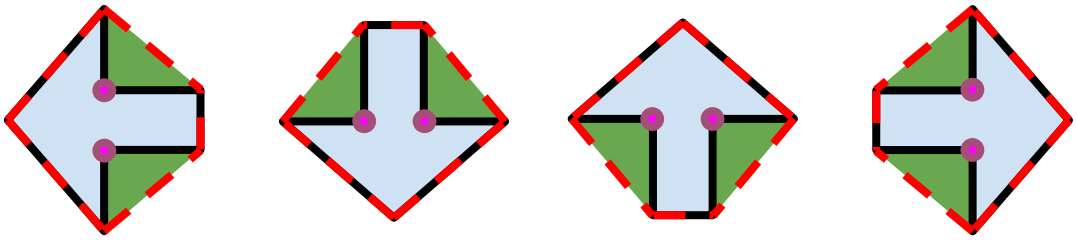

Figure 4: Determining directions of arrows using convex hull defects

3.2 Movement Module

The movement node was implemented as a server. When the service call is made, the movement node passes a python script with the corresponding joint movements to the NAOqi bridge.
The directions included left, right, forward, and back. The forward and backward directions each had two variations: forward-right (move forward on the right foot) and forward-left (move forward on the left foot), and similarly back-right and back-left for backwards.
Whenever the Nao moves forward or backward it alternates between right and left. This is done to keep the Nao facing the screen. For example, every time the Nao does a forward

[2]The template shape matching in OpenCV is rotation-irrelevant and scale-irrelevant, so it is useful for detecting arrows in arbitrary sizes and directions, but also means that we need to detect the directions by ourselves.

[3]In the implementation, we didn't save the "tracks" explicitly - it is here just for making the concept clearer. In fact, we just need to save the position of the slot for the first direction, which can be observed or estimated, and the angle of the row of slots.

right it finishes turned slightly right from where it started. To counteract this a forward-left is done the next time the Nao moves forward. Back also alternated in the same way.
The movements were recorded using Aldeberan's Choregraphe suite[2]. These were recorded in the Nao's animation mode by positioning and then capturing the poses. Each move was stored in a timeline box, consisting of 3 major frames with transition frames in between, as shown in Figure 5. Frame 1 is the standing frame, frame 15 is the final pose frame, and frame 30 is the original standing frame. The frames per second was set to 30, such that each frame lasts 1/30th of a second and the entire move takes one second. The transitioning frames were automatically determined by Choregraphe. Each movement was then converted into a python file using Choregraphe's native export to python function.

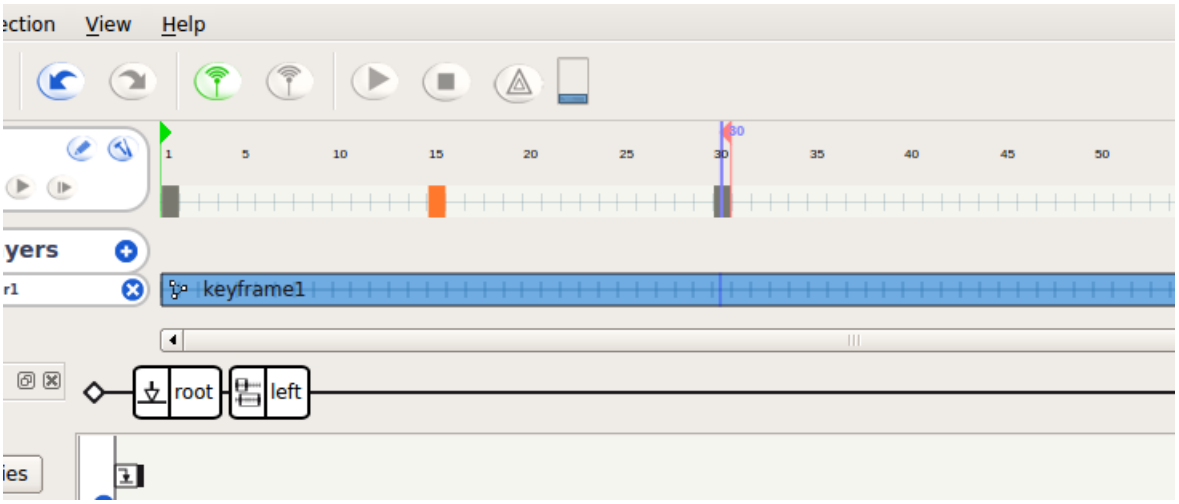

Figure 5: Implementing a move as a timeline in Choregraphe. The gray and highlighted boxes are key frames. The scale represents individual frames.

Due to the difficulty of getting the Nao to balance on one leg, the movements instead involve the Nao sliding the relevant foot forwards or backwards. This has the unintended effect of making the other foot move in the opposite direction, and causes the Nao to rotate away from its starting position as it makes more moves. In order to deal with this excessive rotation, multiple moves were recorded until one with minimal movement was found, and the aforementioned method of alternating between legs was used. To distinguish the forward and backward moves from each other, the Nao's hands are used to point in the direction of movement, and the Nao also says out loud which move it is making. This also makes the Nao more interactive as it dances.

4 Results

Many tests were performed on our implementation, and the results are briefly described here.

4.1 Results from Local Test

To begin with, the code was run using a local video file without the Nao, and it was found that the algorithm was very accurate and efficient. Using a local file, it can produce correct movement commands with the right timing without too much CPU load. Figure 6, Figure 7 and Figure 8 show some screenshots of the running application[4].

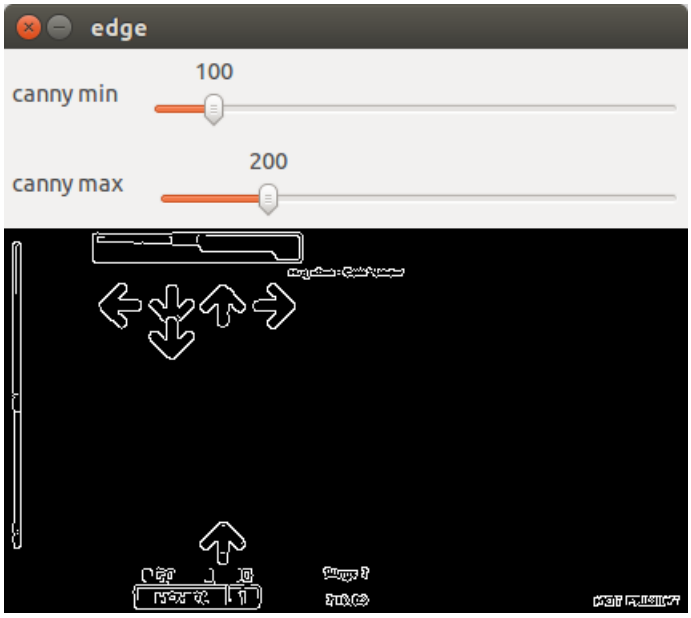

Figure 6: Canny edge detection

4.2 Results in Real Scenario

In the final demonstration, we also ran the application in a real scenario, with a Nao is connected to a computer, and a TV showing the game screen. We used a 70 inch 4K TV to display the images, with the intention that the large screen and high resolution would make it easier to identify relevant shapes. Initially, the results were surprisingly bad, and the Nao could barely recognize arrows and slots. We then realised that the illumination conditions in the laboratory heavily influenced the algorithms. We tried to decrease this influence by turning off the lights and placing a board near the TV to block out ambient light. Doing this produced an acceptable result, with an accuracy above 90 percent, although the response time for the Nao was very long compared to the speed of the arrows. The delay may be due to the transmission and processing time for Naoqi, and cannot be easily fixed with the current architecture design. In order to deal with this, the game difficulty was set to novice, and then the map was modified by removing arrows until the Nao could keep up.

[4]The source code contains lots of debugging lines commented out, which can be uncommented to show more debugging windows with intermediary results.

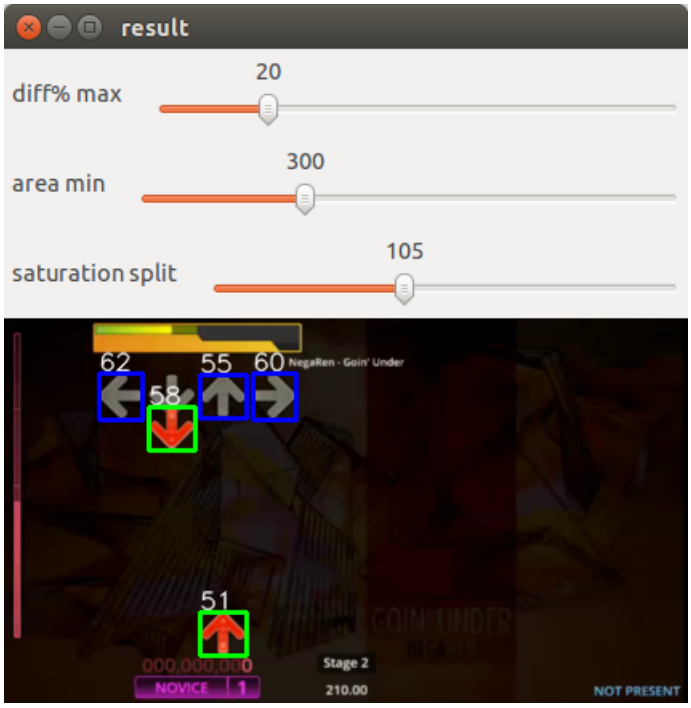

Figure 7: Slots and arrows marked with blue or green respectively in the original image

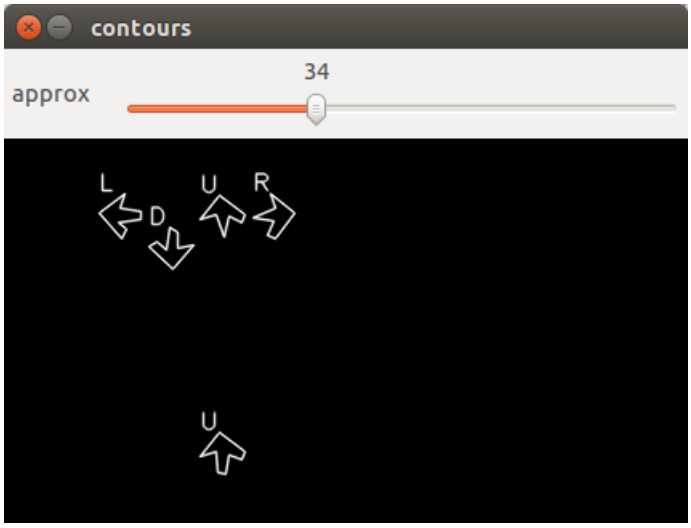

Figure 8: Contours approximation and direction detection

There were also issues with the Nao falling over on the table we had placed it on, due to the design of the movements and the lack of friction between the table and feet. This problem was remedied by placing a piece of cardboard underneath the Nao.
In the final run, the Nao played well; it did not fall over and only missed three arrows in the simple song.

5 Future Work

Future work would focus on three objectives. The first would be to improve the processing speed of the system as a whole. Due to the large amount of time required to process arrows, the game had to be modified to display arrows at a much lower frequency than the default settings. The time taken to recognise the arrows is very short, so the delay exists between the Nao and the PC. In order to decrease the response time, it may be necessary to change the system architecture, such as by running the movement scripts locally on the Nao.
The second focus would be on improving the robustness of the image recognition. The current version is highly affected by lighting conditions, from glare on the screen and ambient lighting. Locating accurate thresholds and improving error margins would be necessary in order to allow the Nao to detect arrows under a wider range of lighting.
The third objective would be to further work on the movements. The current movements do not require the Nao to lift its feet off the ground, so that ensuring the Nao is balanced is relatively easy. However, as seen, this has the negative effect of moving the Nao away from its starting position every time a move is made. Improving the movements would involve adjusting the movements to minimise excessive movement, and if there is sufficient time, redesigning the moves to include the Nao lifting its feet off the ground before placing them in the required position. Not only would this reduce the rotation problem, the Nao would more closely mimic a human playing. The downside is that this requires complex balancing, and may negatively affect the speed of the movements.

6 Conclusion

The overall aim of implementing code for a Nao to play a dancing game was successfully met. The code was able to detect and determine the direction of arrows in the game and make corresponding movements, both in a local file and with a live screen. However, the response time was very slow and the Nao was not able to make movements precisely at the timing required. Furthermore, the Nao was not able to lift its feet off the ground, and so did not completely emulate a human playing the game. The directions of further work to address these shortcomings have been proposed.

References

[1] Open Source Robotics Foundation. *ROS.org*. 2015. URL: http://www.ros.org/.

[2] Aldebaran Robotics. *Choregraphe overview*. 2015. URL: http://doc.aldebaran.com/1-14/software/choregraphe/choregraphe_overview.html.

[3] Aldebaran Robotics. *NAO robot: intelligent and friendly companion*. 2015. URL: https://www.aldebaran.com/en/humanoid-robot/nao-robot.

[4] Aldebaran Robotics. *NAO - Video camera*. 2015. URL: http://doc.aldebaran.com/2-1/family/robots/video_robot.html.