Robot Applications

Lars Engel, Vikash, Ahsan Yousuf

Faculty of Computer Science and Electrical Engineering
Fachhochschule Kiel: University of Applied Sciences
Kiel, Germany

July 9, 2015

Contents

1. INTRODUCTION

Just a short introduction for motivation of project (why should we let the robot play this game?)

2. BACKGROUND

2.1 Explanation and Description of Robot

2.2 Explanation and Description of Camera

2.3 Explanation and Description of Ninemens Morris

2.4 Explanation and Description of the AI

2.5 Setup

3. IMPLEMENTATION

3.1 Work on the artificial intelligence

To simplify the use of the AI, some code was written directly into the classes of the AI. The most important class of the AI, which is responsible for organizing the game flow, is the *GameController*. It runs in an own thread and is started by the *MainController*, which is also part of the AI. (See Figure 1)
To be able to use the self written classes inside of the GameController, it was necessary to forward the classes, that where instantiated inside the robot thread, through the MainController to the GameController.

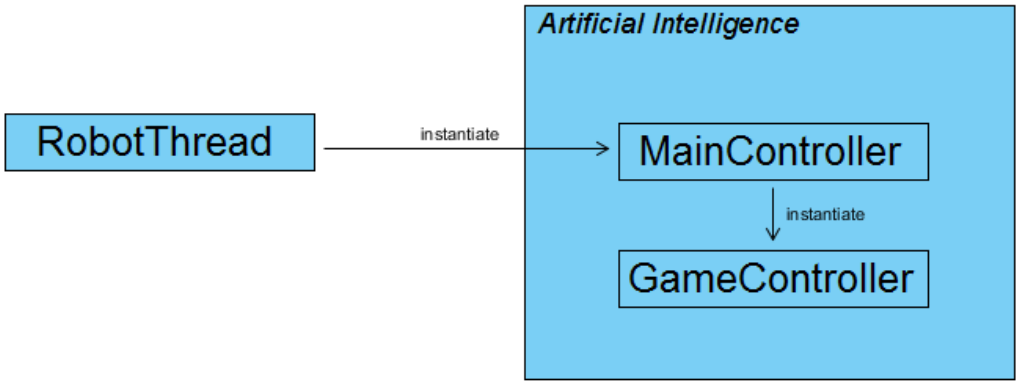

Figure 1. The MainController thread is started by the robot thread. The MainController then starts the GameController thread.

3.2 Description of own classes

3.2.1 RobotInteractions

The RobotInteractions class shown in Figure 2 provides methods so that the robot can interact with the game. It contains for example the *close()* and *open()* methods to open and close the gripper of the robot. The method *movePiece(AbstractFame origin, AbstractFrame destination)* can be used to move a game token from one position to another. It uses methods of the RobotMovements class to perform its movements.

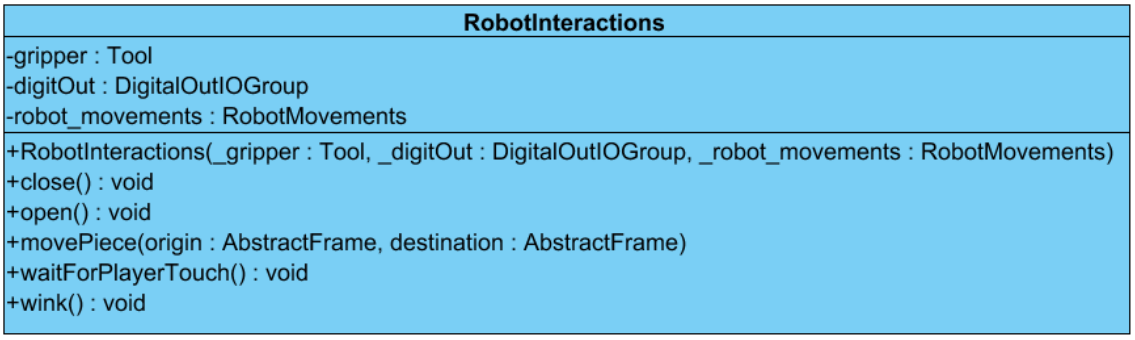

Figure 2. Class diagram of the RobotInteractions class

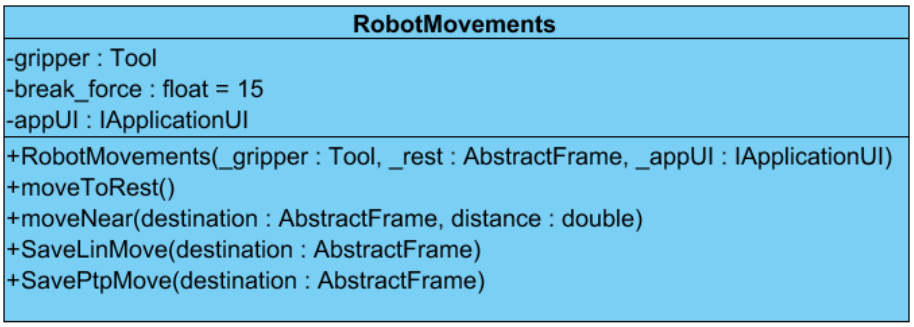

Figure 3. Class diagram of the RobotMovements class

3.2.2 RobotMovements

The RobotsMovements class shown in Figure 3 provides special movement methods like the *moveToRest()* method which moves the robot to its rest position. It also provides the methods *saveLinMove(AbstractFrame destination)* and *savePtpMove(AbstractFrame destination)*. These are methods which where created to enable safety for the human player. Since the robot is interacting in the same space where the human is also interacting, it is important to ensure that the human player will not be harmed by the robot.

As shown in Figure 4 the *savePtpMove(AbstractFrame destination)* method calls the *move()*

```
/**
 * PTP Move method, which stops when a specific force is reached
 *
 * @param destination
 */
public void savePtpMove(AbstractFrame destination) {
    ForceCondition testForceCondition = ForceCondition.createSpatialForceCondition(
            gripper.getDefaultMotionFrame(), break_force);
    IMotionContainer movement = gripper.getDefaultMotionFrame()
            .move(ptp(destination)
                    .breakWhen(testForceCondition)
                    .setJointVelocityRel(0.5));
    IFiredConditionInfo firedCondInfo = movement.getFiredBreakConditionInfo();
    if (firedCondInfo != null) {
        ThreadUtil.milliSleep(1000);
        appUI.displayModalDialog(ApplicationDialogType.INFORMATION, "App Stopped...","Continue");
        savePtpMove(destination);
    }
}
```

Figure 4. Code Listing of the RobotMovements class showing the SavePtpMove() method

method of the *IMotionContainer* class from the KUKA libraries with a *breakWhen()* condition attached to that. This means that the movement will stop, when the specified ForceCondition will be fired. In case the ForceCondition was fired the movement will be stopped, the robot will wait for a second and a dialog will be shown on the KUKA Smartpad. The user then has to click on this dialog so that the movement can be repeated.

3.2.3 Bordpoints

The BoardPoints class shown in Figure 5 provides all the information about important coordinates. At the start of the application this class calculates the coordinates for all 24 board points

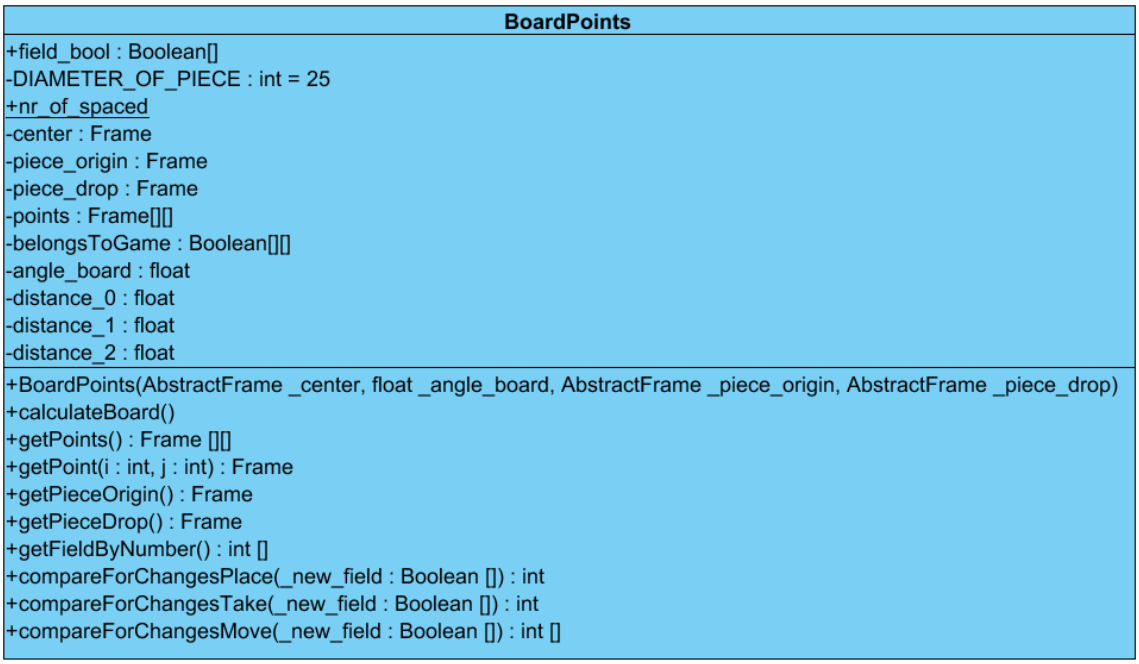

Figure 5. Class diagram of the BoardPoints class

and stores them in a multidimensional array.

Figure 6 shows the calculation for the board point 0, 0 which is the point in the lower left

```
points[0][0] = new Frame();
points[0][0] = center.copy();
points[0][0].setX(center.getX() + (-distance_3  * Math.cos(angle_board) + (-distance_3 * Math.sin(angle_board))));
points[0][0].setY(center.getY() + ( distance_3  * Math.sin(angle_board) + (-distance_3 * Math.cos(angle_board))));
```

Figure 6. Code Listing of the Boardpoints class showing the board points calculation

corner. The calculation uses the center point of the board and the angle of rotation of the board to calculate the coordinates for each game point. The plan was to get the center point and the angle of the board from the camera, but unfortunately the detection of the created jobs was not precise enough to get the correct angle of the board. So in this project the board needed to stay at a fixed angle. But with a correct working method to get the current angle from e.g. the camera this class is already prepared to compute the points based on the rotation of the board.

3.2.4 Logger

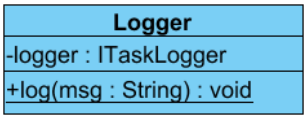

Figure 7. Class diagram of the Logger class

The logger class shown in Figure 7 is instantiated as a static object in the main thread. This way logging information for debug messages or game messages could be used anywhere in the code.

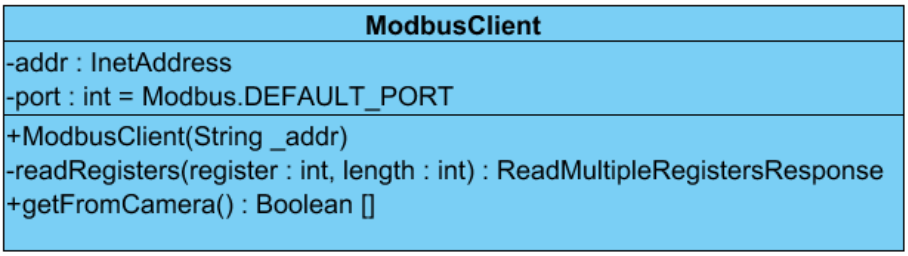

Figure 8. Class diagram of the ModbusClient class

3.2.5 ModbusClient

The ModbusClient class shown in Figure 8 is responsible for the communication between the camera and the robot. As described in section 3.3.2 the communication is realized with Modbus/TCP. The ModbusClient class is instantiated inside the GameController of the artificial intelligence. The GameController calls the *getFromCamera()* method which uses the *readRegisters(int register, int length)* method to read the specified registers from the Modbus/TCP server provided by the camera.

3.3 Visual analysis

The visual analysis of the game was realized with an Cognex In-Sight 7000 Integrated Vision System.

3.3.1 In-Sight Explorer

With the help of the Cognex In-Sight Explorer it was possible to create jobs to analyze the image captured by the camera.

It was the intention to create a job that can analyze the positions of game tokens on the board considering the rotation of the board. To accomplish this the job should detect the corners of the game board. These corners then could be connected. The intersection between the two lines that cross the game board will be the center of the game board (See Figure 9 part one). Comparing the angle of one of the lines with a reference line with an angle of 0 degrees provides the information about the rotation of the game board.

Unfortunately the detection of the board corners did not work as precise as needed. This lead to the problem that the rotation of the board could not be analyzed correctly. A wrong angle of the board would lead to wrong calculations for the board points. Since it is important, that the location of the board points are precise, so that the robot moves and takes pieces from the correct coordinates, the decision was made to use a fixed rotation for the game board. This is a limitation that could be resolved in a following project.

The detection of the tokens is realized with a tool of the In-Sight Explorer that detects circles at specific locations. It will output a message "available", if a circle was found at this location and another message "not available", if no circle was found at this location. The tool is not able to determine the color of the found circles. But since nine men's morris is a game, that is played in turns, we can get the information about the color of the tokens within the robot application. On the downside, this limits the ability of cheat or misplaced game token detection.

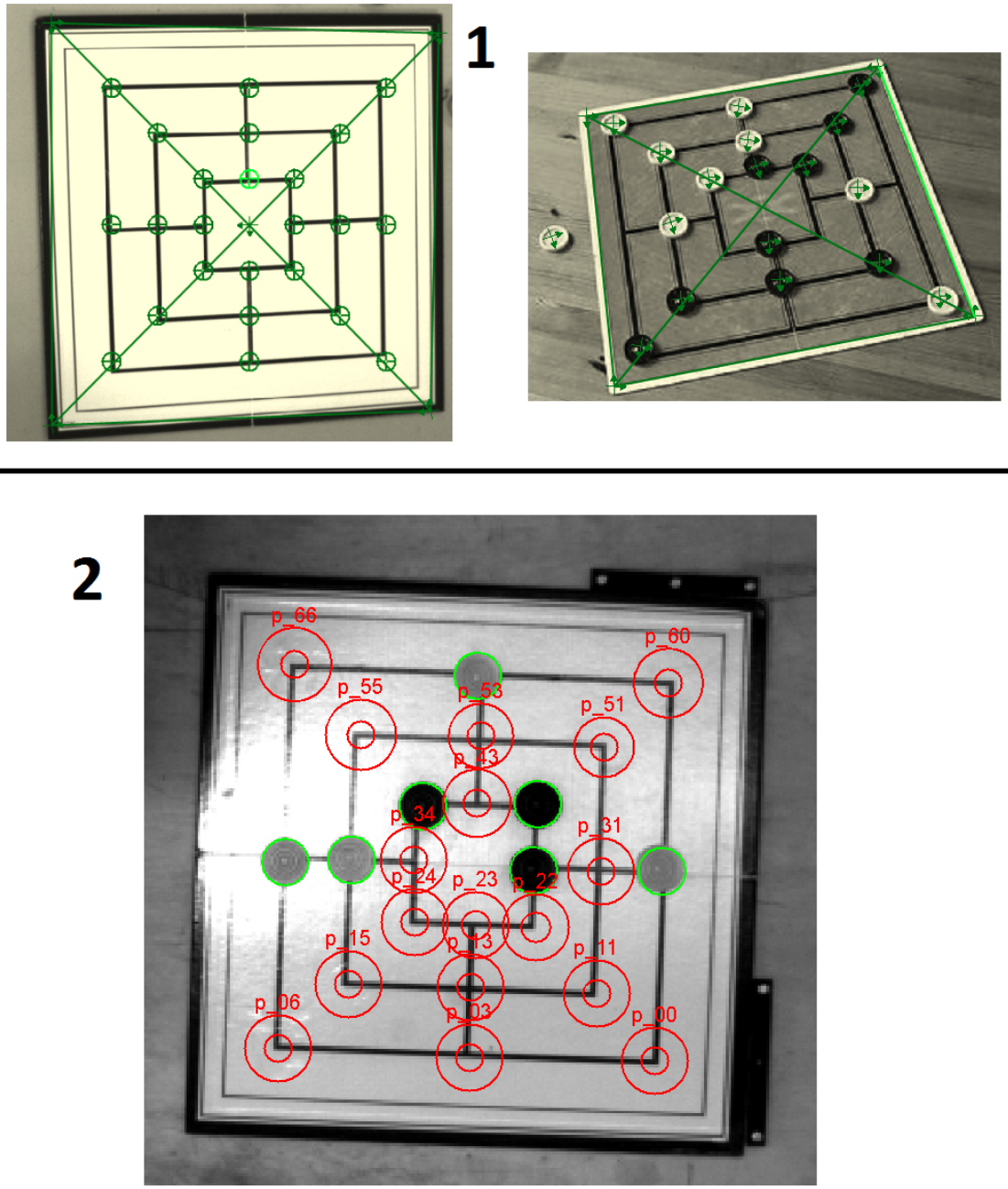

Figure 9. Different approaches for visual analysis

3.3.2 Communication between Robot and Camera

The communication as shown in Figure 10 uses the Modbus/TCP protocol. The camera acts as an Modbus/TCP server and the robot as Modbus/TCP client. The *ModbusClient* class of the robot application (see Figure 8) sends a *ReadMultipleRegistersRequest()* to the IP and port of the camera. The camera then sends the output of the visual analysis as TCP packages back to the robot application. The response of the camera is stored as an array of 24 boolean values inside the robot application where *true* means a token was found and *false* means no token was

found at this location. This array can now be compared to the previously stored array *field_bool* in the *BoardPoints* class (see Figure 5).

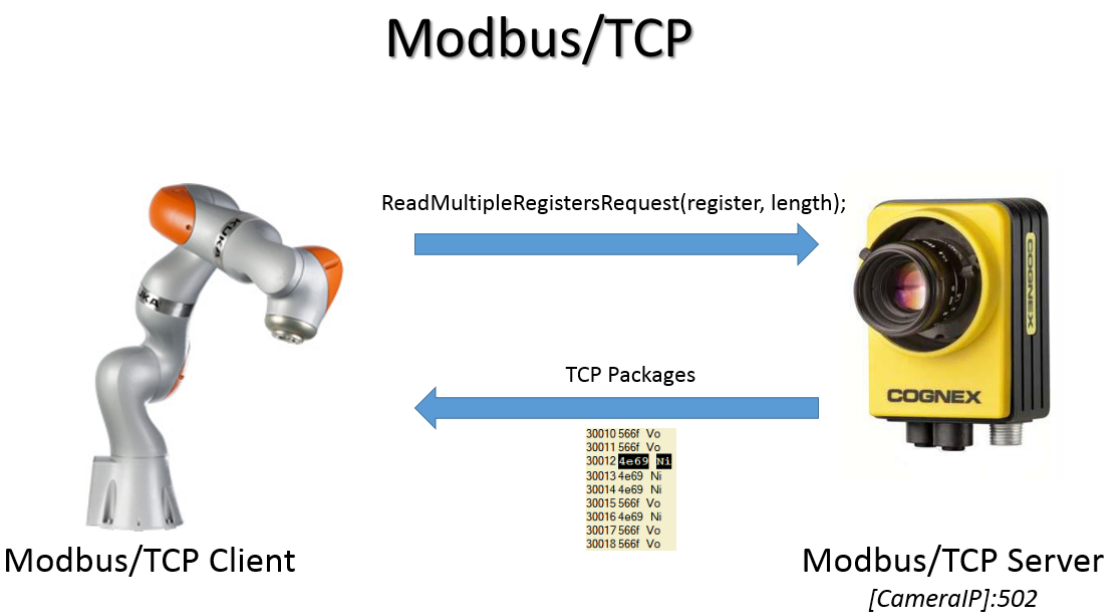

Figure 10. Communication between Camera and Robot

4. CONCLUSION AND FUTURE WORK