Technical Specification

SDP Group 12

March 7, 2016

1 Introduction

The aim of this project was to create a robot that would play a simplified version of 2-a-side football. We were recommended to use the provided Arduino Uno (Xino RF) board and LEGO pieces, a vision feed of the testing area (pitch), and computers to connect the two and implement any further logic.

This report will examine the decisions made in the design of each part and their impacts. Furthermore it will explain the principles our work relies on.

1.1 Work Distribution and Management

Overall the group was efficiently split into subgroups working on specific tasks. As opposed to many other groups, all team members have made significant contributions to the outcome of the project.

Furthermore, many tasks were shared with Group 11, and large contributions introduced by teamwork between the two halves of the team.

2 Vision

The vision system is aimed to be a simple and robust platform, which would provide the planner (predictor) with a world description without introducing much additional delay. The predictor will deal with smoothing the vision output and buffering object position if an object cannot be detected for a limited number of frames. The result of this will be then fed into the planner, which will control the robots.

2.1 Approach

The vision system is common for both groups in the team. Developed by Group 12 and augmented with details from Group 11's initial vision platform. The vision uses a HSV (hue, saturation, value) coded frame rather than the default BGR (blue, green, red). This enables us to have more control over the colours, rather than the intensity and brightness. When a frame is captured, we first remove radial distortion and then blur the image so that all objects become more connected and are easier to identify. The frame is then converted to HSV and all future processing will take place over the HSV frame.

Main approach to the vision system is to detect clusters of colour, such as yellow and blue being the robot centres and red being the ball. Colour is detected by creating a mask based on configured thresholds, and finding contours within that mask. In case more than two robots for the same team are detected only the best two matches are taken. To make sure a yellow or a blue cluster is indeed a robot, we also check if there are any pink or green circles around. If none are found, then this cluster is skipped and we move onto the next largest one. After a cluster is identified as a robot centre, we identify the green and pink colour clusters in a 40x40 pixel area around the centre. If we find three pink clusters, then the robots is the pink teammate, otherwise it is the green teammate. Once we find the teammate colour, we then find the centre of the circle used for orientation (pink circle for green teammate; green circle for pink teammate). A vector is then taken from the centre of the orientation circle to the centre of the robot, shifted by 45 degrees and drawn from the robot centre to indicate it's orientation. For the ball, the largest red cluster is taken and for the orientation, we use the vector from previous position to current position. In all objects velocity is calculated relative to the movement from the previous frame.

There will be endeavours to augment this approach with movement and shape detection, since colour thresholding can become unreliable if the thresholds are not perfectly adjusted.

2.2 Semi-Automatic Calibration

The vision system has several calibration routines, most importantly a colour calibrator.

The colour calibration routine is run on every execution but can be skipped. The vision platform will prompt the user to click on different coloured objects. Every click will take a 3x3 square of pixels around the click and record those that are in a predefined hue range hard coded for each colour. Inappropriate values are discarded. The gathered pixels will define the saturation and value of the colour ranges, and limit the valid hue values.

This makes it easy to recalibrate the whole vision system on each launch to adapt to environment changes.

2.3 Manual Calibration

The platform implements a manual way of determining threshold values for colour filters, as well as a pitch cropping specification routine.

The vision system offers sliders to manually determine the threshold values for the colour masks. This is useful for debugging and initial calibration of the colours.

The cropping calibration, which is currently disabled (trigger in the vision code is commented), has to be very rarely run (ideally once per camera after every change in the camera angle or position). This enables us to dynamically and efficiently correct the pitch area and ignore any outlying distractions.

3 Planning

3.1 Design

The planner uses a system of goals and actions. Goals define an overall strategy in a given situation leading to a particular aim. Actions define one instruction given to the robot with preconditions for its execution. Goals are composed of a sequence of actions. A goal object selects the next action required to achieve its aim by traversing its ordered list of actions and evaluating their preconditions.

In the implementation, all goals and actions are derived from their respective super classes. These are defined as follows:

```
class Goal(object):
    '''
    Base class for goals
    '''
    def __init__(self, world, robot):
        self.world = world
        self.robot = robot

    # Return the next action necesary to achieve the goal
    def generate_action(self):
        info("Generating action for goal: {0}"\
                .format(self.__class__.__name__))
        for a in self.actions:
```

```
            if a.is_possible():
                return a
        return None


class Action(object):
    '''
    Base class for actions
    '''
    preconditions = []

    def __init__(self, world, robot, additional_preconds=[]):
        self.world = world
        self.robot = robot
        self.preconditions = self.__class__\
                .preconditions + additional_preconds

    # Test the action's preconditions
    def is_possible(self):
        info("Testing action : {0}".format(self.__class__.__name__))
        for (condition, name) in self.preconditions:
            if not condition(self.world, self.robot):
                info("Precondition is false: {0}".format(name))
                return False
        info("Action possible: {0}".format(self.__class__.__name__))
        return True

    # Do comms to perform action
    def perform(self, comms):
        raise NotImplementedError

    # Get messages relating to action
    def get_messages(self):
        return []

    def get_delay(self):
        return DEFAULT_DELAY
```

The planner makes its decision based on only on the world state passed to it. This is described by an instance of the class passed to the planner. This object's state is updated by passing a new set of positions (as a dictionary of vectors) to the `update_positions` method of `World`. The `World` class describes the state of the world from the vision, including position vectors for the robots, the ball and the goals. The `World` class and its associated classes also provide methods on their data providing the planner with information about the world. Further utility functionality can be found in "utils.py"

The overall planner is used by calling `plan_and_act(world)` . This selects a goal based on the given world state using its `get_goal()` method. From this goal an action is generated using `generate_action()`. The method then runs (using `actuate(action)`) and returns a delay giving the time until the planner should be run again.

3.2 Implementation

Our attacker robot's goals and their respective actions are as follows:

Table 1: AttackPosition

Action	Preconditions
TurnToDefenderToReceive	Attacker in score zone
GoToScoreZone	Attacker is facing score zone
TurnToScoreZone	None

Table 2: Score

Action	Preconditions
Shoot	Attacker has ball and attacker can score
TurnToGoal	Attacker has ball

Table 3: GetBall

Action	Preconditions
GrabBall	Attacker can catch ball and attacker's grabbers open
GoToGrabStaticBall	Ball static, attacker facing ball, attacker's grabbers open
OpenGrabbers	Ball in attacker's grab range, attacker's grabbers closed
GoToBallOpeningDistance	Attacker is facing ball
TurnToBall	None

Table 4: AttackerBlock

Action	Preconditions
TurnToBlockingAngle	Attacker in blocking position
GoToBlockingPosition	Attacker is facing blocking position
TurnToFaceBlockingPosition	None

The planner chooses a goal based on certain situations, as described in following table:

Table 5: Goals chosen dependent on ball position

Robot in possession	Goal
Our attacker	Score
Our defender	AttackPosition
Their attacker	AttackPosition
Their defender	AttackerBlock
Ball free	GetBall

3.3 Extensibility

Extending the planner is simply a case of adding goals and actions then adding the logic to select these in `select_goal(world)`. Any new goals or actions should subclass Goal and Action respectively and override methods were stated.

In actions, the logic for performing should be placed in `perform(comms)`. This method is passed a `CommsManager` object through which the robot can be sent instructions. No more than one call should be made to this object in any given action. A new action should also override `get_delay` giving an appropriate delay (in seconds) before the planner should run again. New actions can also have preconditions defined in a variable `preconditions`.

In goals, it may only be neccessary to write a Goal subclass with an ordered list of actions (from last to first). Otherwise the `generate_action()` method can be overriden but similar logic should be followed.

3.4 Integration

An instance of the `Planner` is kept by the "main.py" script. Based on the delays given by the planner, it calls the planner at varying intervals, passing it the latest world model provided by the vision. The planner uses a `CommsManager` object to make control the robot.

4 Communications

Communications between the arduino and computer are done over the supplied RF link. The frequency is set to the group frequency specified. Further the optional encryption was enabled and the PAN ID set to 6810.

Data is send to and from the devices in packets, the basic form of a packet is:
'`<target><source><data><checksum>\r\n`'
`<target>` and `<source>` are the ASCII character '1' for the group 11 robot, the ASCII character '2' for the group 12 robot, and the ASCII character 'c' for the computer. For robot to computer messages, the targets 'd' and 'e' are also valid, used to send debug and error messages respectively. `<data>` is the base64 encoded binary message, and `<checksum>` a checksum of the preceding message. The base64 encoding does not use padding bytes, but is otherwise standard.

The checksum consists of 2 characters, which are calculated as the checksum of all characters at even (starting 0) and odd indices respectively, including `<target>` and `<source>`. These checksums are done by retrieving the value from 0-63 of the character in base64 encoding, XORing these together. The resulting value from 0-63 is then encoded back into a base64 character.

For example, the checksum of '`dt6bas2`' is '`La`'.

Devices ignores all packets not addressed to it, as well as malformed packets (including incorrect checksum). Upon receipt of a packet, the device sends an acknowledgement. An acknowledgement packet has the form '`<target>$<checksum>\r\n`'. An acknowledgement packet is not itself acknowledged.

The computer side will resend packets which have not received an acknowledgement. The Arduino side does not, and discards packets which are longer than the internal buffer (60 bytes). To prevent flooding, only the previously sent packet is resent. This suits the application well, as previously sent packets contain instructions to the robot which are superseded by the more recent one.

4.1 Bytecode

The data received by the Arduino from the controller is interpreted as a sequence of bytecode instructions. Each instruction starts with a single byte opcode specifying the instruction itself, followed by a number of bytes for the argument of the instruction. The number of bytes the argument used depend on the instruction itself (See the arguments column below; An instruction with a single `uint16_t` argument will take 2 bytes for its arguments).

Opcode	Arguments	Description
`0x00`	time (`uint16_t`)	Waits for the given time
`0x01`	time (`uint16_t`)	Brakes for the given time
`0x02`	distance (`int16_t`)	Moves the given distance to the right (or neg. left)
`0x03`	angle (`int16_t`)	Spins the given (possibly negative) angle clockwise
`0x04`	time (`uint16_t`)	Activates the kicker for the given time
`0x06`	*none*	Opens the grabbers
`0x07`	*none*	Closes the grabbers
`0x09`	speed (`uint8_t`)	Sets the global speed modifier (value from 0-255)

Times are given in milliseconds, distances in mm and angles in minutes. The arguments are read directly from instruction sequence memory as the given type, and the next command starts after the current one finishes. Numbers are encoded in little-endian.

The instructions, when received, are compiled by the Arduino into low-level commands, and inserted at the earliest possible point in the low-level command queue.

4.2 Low-Level Commands

The arduino internally maintains a queue of low-level commands which capture what the robot is currently doing. Once a command finishes, the Arduino immediately moves on to the next. Structurally, low-level commands work similarly to high-level ones. They start with an opcode, depending on which some arguments may follow. The available low-level commands are:

Opcode	Arguments	Description
`0x00`	time (`uint16_t`)	Waits for the given time
`0x01`	time (`uint16_t`)	Brakes for the given time
`0x02`	time (`uint16_t`)	Opens the grabbers for the given time
`0x03`	time (`uint16_t`)	Closes the grabbers slowly for the given time
`0x04`	time (`uint16_t`)	Closes the grabbers quickly for the given time
`0x05`	time (`uint16_t`)	Activates the kicker for the given time
`0x08`	distance (`int16_t`)	Moves right by the given distance
`0x09`	angle (`int16_t`)	Spins clockwise by the given angle
`0x0b`	speed (`uint8_t`)	Sets the global speed modifier to the given value
`0x7f`	*none*	Does nothing

Further, the most significant bit of a low-level commands opcode signifies if this command is interruptible or not. If it is 1, the command is not interruptible.

The Arduino periodically updates the instruction list to decrease the time argument of the current instruction by the time passed, or, if the instruction is finished, moving onto the next one. If a new command sequence arrives, it is inserted in place of the first interruptible command in the sequence, with this, and all later commands being discarded. This ensures that non-interruptible commands are always run, and are run even if they consist of complex sequences which should not be interrupted. This is also the main reason for a no-op existing; it allows separating multiple sequences of non-interruptible commands.

Instructions setting values relating to the communications test are set immediately, and the Arduino moves on to the next instruction. The send instruction `0xf2` suspends normal operation and executes a writing the buffer to the I^2C port with the previously set delay.

5 Hardware

Will submit this section in the next few days