An Eleague Robocup Platform

Stephen Tarzia
26 December 2004

Abstract: This paper describes an Eleague robotic soccer platform assembled at Columbia University. This platform is highly modular. The most important module is a customized version of Mezzanine -- an overhead camera object tracking software developed at USC's Robotics Lab. In this paper, I describe the role of each module as well as give usage instructions and evaluate the system.

Table of Contents

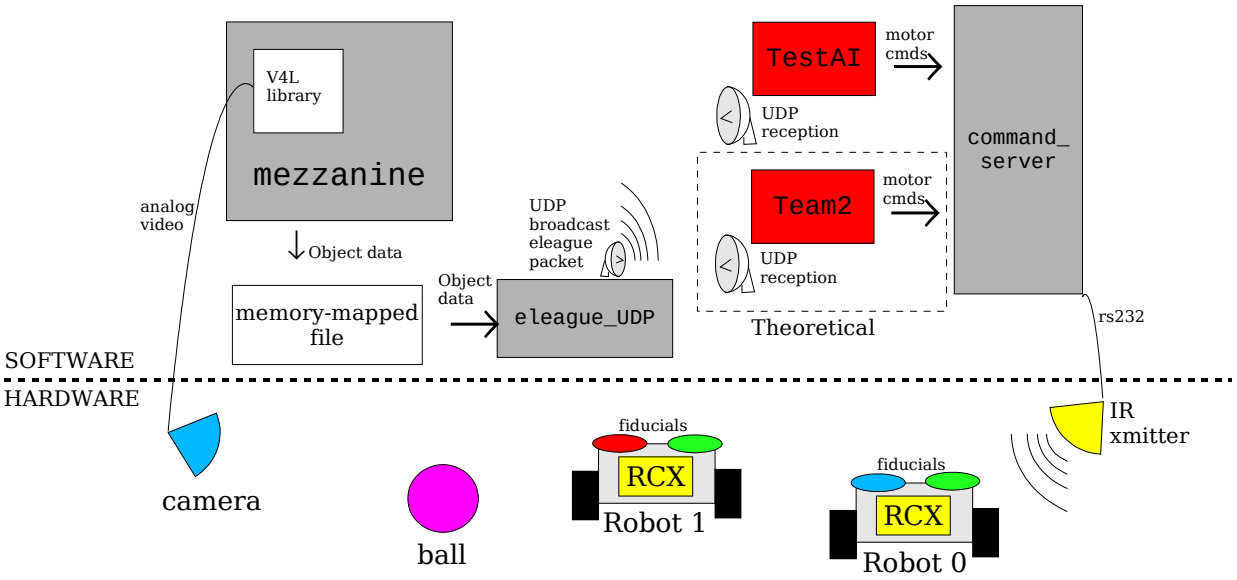
1 Introduction

The collection of hardware and software described in this paper provide a platform for robotic soccer programming

System requirements:

- x86 PC, approximately 700mhz
- Linux and all the basic development tools
- Video4Linux compatible capture card and V4L enabled kernel. see http://www.video4linux.net/ for details
- Color video camera
- Third party libraries for mezzanine: GTK+ 1.2 or better and GSL 0.9 or better.
- Lego RCX based robots
- IR transmitter, see section 8
- brightly colored fiducials and plastic ball

2 System Architecture



3 Camera

Any camera that outputs a composite or s-video signal should work. However, it is important to consider the viewing angle (focal length) when choosing a camera. Fitting the entire pitch withing the camera's field of view may necessitate using a wide angle lens. Cheap color video cameras with built-in wide angle lenses can be purchased for less than $100. We used an X10 wireless color camera. Wide angle lenses exhibit barrel distortion. This type of distortion makes the image appear bulged outward in the middle. Our need to correct this distortion is what motivated our adoption of Mezzanine for tracking. Mezzanine has barrel distortion correction built in [5].

4 Mezzanine Package

Mezzanine is an overhead camera 2D vision tracking system that uses Video4Linux as a video source. Objects are marked with uniquely colored fiducials that Mezzanine searches for when trying to locate the objects. Mezzanine's configuration file, `mezzanine.opt`, is located in the user's home directory. This file stores object and color definitions as well as the dewarp grid, Setup instructions and documentation for mezzanine can be found at the project's website (http://playerstage.sourceforge.net/mezzanine/mezzanine.html).

4.1 `mezzanine` Video Server

`mezzanine` is the program that does all of the video processing and object identification.

4.1.1 Algorithm Overview

- load user's configuration from mezzanine.opt
- build the image to world coordinates transformation
- loop for each video frame grabbed from V4L:
    - color classify – Examine each pixel in image. If it fits one of the user's color definitions then assign that color to the pixel.
    - find blobs – Identify groups of adjacent identically colored pixels as “blobs”.
    - find objects – For each object (fiducial) definition, try to find a matching pair of adjacent blobs. Record the position, angle and velocity of the object.
    - send a signal indicating that new data is ready

We are using Mezzanine as a robot and ball location server for eleague. To fill this role, a few changes had to be made to Mezzanine, as follows:

4.1.2 Changes made

1. Support for distinct objects

Mezzanine originally supported the definition of and tracking of only one type of object. For eleague soccer, at least three distinct objects must be tracked (the ball and two different teams). Distinguishing between each member of a given team may also be desired if static roles are to be assigned (especially for a goalkeeper).

The new syntax for object definitions is `ident[i]` where i is the object number, starting with zero. This change applies to section 3.5.5 of the mezzanine manual [1].

instead of defining a single object:

```
ident.class[0] = 0
ident.class[1] = 1
```

we now define multiple objects:

```
ident[0].class[0] = 0
ident[0].class[1] = 2
ident[1].class[0] = 1
ident[1].class[1] = 4
ident[2].class[0] = 3
```

2. Support for single-blob objects

Mezzanine originally only supported two-blob objects. This, of course, would not work for tracking a ball. Single-blob support was added for this reason. The user may simply omit the declaration of a second color class for any object to make it a single blob object. Since single-blob objects are radially symmetric the notion of a pose angle does not apply. Therefore, single-blob objects are assigned a zero angle. Internally, a single-blob object's second color is assigned `NULL_COLOR = MAX_INT.`

In the example above, object 2 is a single-blob object of color class 3.

3. Brightness and Contrast control

Brightness and contrast control from within mezzanine.opt was added because I found that adjusting these settings can improve color saturation and lead to easier distinction between colors. The syntax is:

```
fgrab.brightness = [0-100]
fgrab.contrast = [0-100]
```

4.2 Mezzanine calibration with `mezzcal`

`mezzcal` provides an easy-to-use graphical interface for configuring some aspects of Mezzanine. The following three settings can also be configured manually by editing the mezzanine configuration file, but they are much easier to set using `mezzcal`:

4.2.1 Geometry Dewarping

See the mezzanine manual, section 4.2.4 for details. I found that for lenses with significant amounts of barrel distortion the calibration points had to be set very carefully in order to avoid bogus image to world mappings. I recommend measuring out the calibration points on the pitch and marking them rather than just trying to do it by eye. It is easy to tell if your dewarp grid is inaccurate. In this case the blue object squares will be offset from the actual location of the corresponding fiducial.

4.2.2 Color Definition

See the mezzanine manual, section 4.2.2 for details. Calibrating the color definitions is a long, iterative process. Uniform lighting on the pitch is essential. Also, the fiducial colors must be very bright (unless you are using an an expensive camera).

4.2.3 Image Mask Definition

See the mezzanine manual, section 4.2.1 for details. This is a straightforward procedure.

4.3 The Mezzanine Library: `libmezz`

`libmezz` is an IPC library that allows a program to access the data stored by mezzanine. Most notably, this includes the poses of all of the objects. This is facilitated by a shared memory-mapped file. This library is what makes Mezzanine such a versatile platform. `eleague_UDP` uses this library.

5 Object State Broadcasting: `eleague_output/UDP`

`eleague_output` and `eleague_UDP` are simple programs that broadcast the robot and ball data in the uleague format [2]. They use libmezz to retrieve this information from Mezzanine. `eleague_output` writes to stdout while `eleague_UDP` broadcasts using a UDP socket. Our network admin complained about the network flooding `eleague_UDP` was causing, so use it at your own risk (be sure not to leave it running overnight as I accidentally did).

`eleague_UDP` and `eleague_output` commandline arguments:

`-x [length]`
: sets the pitch length, in millimeters. We use the identifier 'x' because the long side of the field corresponds to the x axis in the camera image. Default value is 4600, corresponding to the Robocup legged-league's field [4].

`-y [width]`
: sets the pitch width in millimeters. Default value is 2700, corresponding to the Robocup legged-league's field.

`eleague_UDP` specific commandline arguments:

`-p [port number]`
: sets the port number to listen on.

`-d`
: enable verbose debugging output

5.1 Testing with `ClientPrinter`

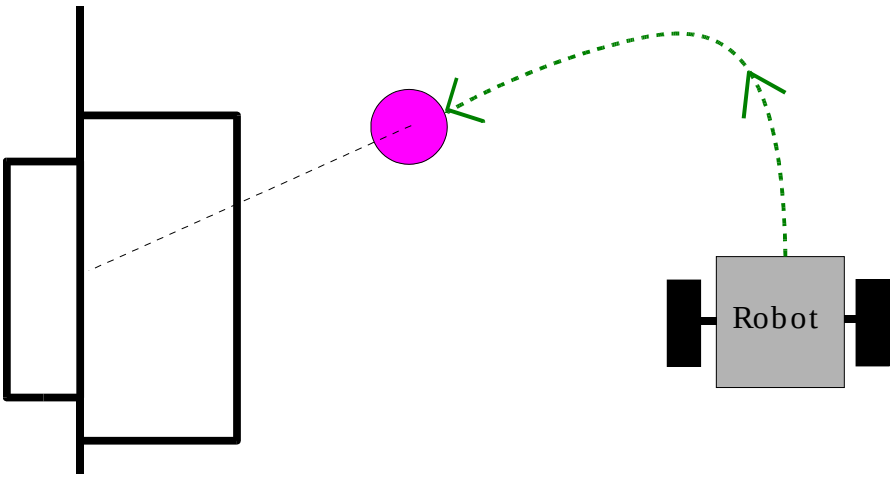
`ClientPrinter` is a simple test utility for eleague_UDP broadcasts. It sets up a UDP socket to receive the data that is being transmitted and prints the packets as they arrive. This program was provided with doraemon, which uses the same broadcast format as eleague_UDP [2].

`ClientPrinter` commandline arguments:

`-p [port number]`
: sets the port number to listen on.

6 Example User Program: `TestAI`

`TestAI` is a simple C++ example robot control program. This code is based on Jake Porway and Gaurav Singal's eleague 2004 code. It receives the object position data from `eleague_UDP` and sends robot instructions to `command_server`. The program simply instructs each robot to attempt to score a goal by moving to the location of the ball with an approach vector pointing at the left-hand-side goal. CMU's differential drive motion algorithm was used for this [3]. `TestAI` also has a useful motor testing mode that does not require the video server to be running.



`TestAI` listens for data on port 6363 and transmits motor commands on port 6364.

optional commandline parameters for `TestAI` are:

`--step`
: Wait for user input before proceeding to issue the next set of instructions to the robots.

`--verbose`
: Verbose output

`--naive`
: Use a naive motion algorithm instead of CMU's complex algorithm. The naive algorithm moves forward if the robot is roughly facing the ball. Otherwise, the robot spins in place.

`--test [0.0 - 1.0]`
: Enable motor test mode. In this mode, the user enters a motor command followed by the enter key to command all of the robots. This

mode is useful for testing the robots, the command server, and the IR transmitter. `--test` must be the first argument passed.
The motor commands are as follows:

'5' = forward
'2' = backward
'1' = turn left
'3' = turn right

(on the keyboard's number pad these form the familiar upside-down 'T')
Optionally, the user may pass a floating point number [0.0 - 1.0] to indicate what motor power level to use. The default is 1.0 = full power.

`TestAI` is intended to be a model for teams to base their own, more complex, robot control programs on. It might also be useful to adapt it into a an eleague user API.

7 Motor Broadcasting: `command_server`

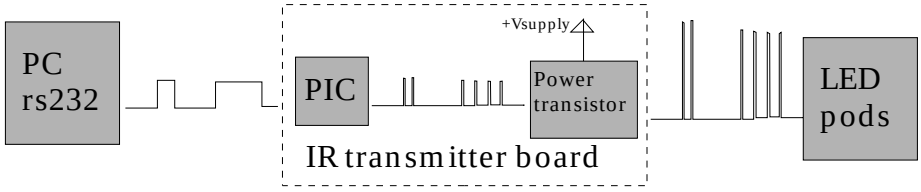
The command server continually sends robot motor commands to the IR transmitter. These motor commands are updated by the team AI programs by sending an ASCII message to command_server through the specified port. This program was written by Jacky Baltes and is part of the uleague package [2].

`command_server` commandline arguments:

`-p [port number]`
sets the port number to listen on.

8 IR Transmitter

`command_server` is designed to work with either the RS232 or USB Lego IR Tower. These are omni-directional devices with a transmission range of several feet. In order to transmit motor commands to all of the robots on a large pitch, a high power IR transmitter must be used instead. We built one using Jacky Baltes' schematic. and PCB. For transmission, it is plug-and-play compatible with the Lego RS232 Tower. This transmitter does not have reception capabilities -- it only transmits. For this reason, when doing Lego RCX firmware updates the standard short-range Lego IR Tower must be used.



8.1 Assembly Notes

- You may only need to use the large pod to transmit across the entire pitch. It you end up with "dead zones" then use the small pods as well.
- The most sophisticated part of the assembly is programming the PIC microcontroller. Your EE department should be able to provide you with a PIC programmer and software for this task. Just cut and paste the code provided and then flash it onto the PIC.
- Be sure to use a null modem cable to connect the transmitter to your computer (the gray cable included with the Lego RS232 Tower will work). You should use a male 9-pin D-sub connector on the board.
- Be sure to match the series forward voltage of your LEDs with the voltage of the power supply. Check your LED's specifications for this specification. If the voltages are matched then you should short the IR pod jumpers to cut out the resistors. If the voltage of the power supply is higher than the operating voltage of the series LEDs then leave the jumpers open to lower the voltage supplied across the LEDs.

9 BrickOS Firmware

In order to understand the commands sent by `command_server`, each robot's RCX must be loaded with BrickOS firmware and Jacky Baltes' uleague BrickOS program. These steps are both done using the `BrickOS/firmware.sh` script.

If the RCX has BrickOS loaded it will not beep when turned on. To activate an RCX, turn it on and push the `run` button. Once, running, the digit on the screen indicates which robot it is acting as. To change this number press the `program` button. To turn off the RCX, first press `run` to disable it and press `power` to turn it off.

If the batteries are taken out for more than a few seconds, this procedure will need to be repeated, as the RCX's program memory is volatile.

10 Conclusion: System Evaluation

The system described in this paper is fully functional, though it is not without shortcomings. Most notably, mezzanine dewarp calibration is less than perfect and the UDP broadcast implementation is prone to network flooding. Overall, though, the system is quite workable.

A Installation Instructions

Installation is a five-step process:

1. Compile the source code. From the top-level package directory:
   `$ make`

2. Optional: move the compiled binaries to a system directory. We will assume that this is not done.

3. Write a configuration file and name it
   `~/mezzanine-0.00/etc/mezzanine.opt`
   An example is included in `Mezzanine/examples/mezzanine.opt`
   See section 4.1.2 and the mezzanine manual for details.

4. Adjust the configuration file using `mezzcal`. See section 4.2 for details.

5. Load the firmware onto the Robots.
   `$ ./BrickOS/firmware.sh`

B Runtime Instructions

This section describes how to run a complete test of the system using the TestAI code.

1. start mezzanine
   `$ ./Mezzanine/mezzanine/mezzanine`

2. start eleague output program
   `$ ./eleague_output/eleague_UDP -p 6363`

3. start the command server
   `$ ./command_server/command_server -p 6364`

4. start the AI program
   `$ ./TestAI/TestAI --verbose`

5. start the robots
   Turn `on`; push `run`; push `program` to set each robot's number.

C Sample mezzanine.opt

```
# ~/mezzanine-0.00/etc/mezzanine.opt

fgrab.offline = 0
fgrab.norm = ntsc
# Image size to work with
fgrab.width = 640
fgrab.height = 480

# normal brightness and low contrast
fgrab.brightness = 50
fgrab.contrast = 35

# Image mask -- defined in mezzcal
# this limits the area of interest.
# No pixels outside of this boundary are considered
classify.mask.poly[0] = (607, 216)
classify.mask.poly[1] = (598, 127)
classify.mask.poly[2] = (578, 33)
classify.mask.poly[3] = (519, 17)
classify.mask.poly[4] = (430, 9)
classify.mask.poly[5] = (304, 11)
classify.mask.poly[6] = (180, 23)
classify.mask.poly[7] = (73, 51)
classify.mask.poly[8] = (47, 168)
classify.mask.poly[9] = (42, 213)
classify.mask.poly[10] = (39, 265)
classify.mask.poly[11] = (57, 390)
classify.mask.poly[12] = (172, 417)
classify.mask.poly[13] = (276, 429)
classify.mask.poly[14] = (363, 433)
classify.mask.poly[15] = (455, 428)
classify.mask.poly[16] = (572, 409)
classify.mask.poly[17] = (598, 282)

# Color Definitions

# the color name is entered manually
classify.class[0].name = green

# this is the rgb color used to denote this color class in mezzcal.
# this is also entered manually
classify.class[0].color = (0, 255, 0)

# The color-space polygons were set using mezzcal
classify.class[0].vupoly[0] = (126, 111)
classify.class[0].vupoly[1] = (120, 144)
classify.class[0].vupoly[2] = (71, 160)
classify.class[0].vupoly[3] = (70, 137)
classify.class[0].yupoly[0] = (182, 141)
classify.class[0].yupoly[1] = (188, 113)
classify.class[0].yupoly[2] = (80, 134)
classify.class[0].yupoly[3] = (115, 157)
```

```
classify.class[0].vypoly[0] = (57, 130)
classify.class[0].vypoly[1] = (112, 72)
classify.class[0].vypoly[2] = (123, 191)
classify.class[0].vypoly[3] = (81, 160)

classify.class[1].name = pink
classify.class[1].color = (255, 105, 180)
classify.class[1].vupoly[0] = (216, 167)
classify.class[1].vupoly[1] = (229, 164)
classify.class[1].vupoly[2] = (223, 191)
classify.class[1].vupoly[3] = (193, 189)
classify.class[1].yupoly[0] = (197, 183)
classify.class[1].yupoly[1] = (173, 187)
classify.class[1].yupoly[2] = (166, 168)
classify.class[1].yupoly[3] = (196, 173)
classify.class[1].vypoly[0] = (226, 202)
classify.class[1].vypoly[1] = (204, 196)
classify.class[1].vypoly[2] = (193, 208)
classify.class[1].vypoly[3] = (222, 222)

classify.class[2].name = red
classify.class[2].color = (255, 0, 0)
classify.class[2].vupoly[0] = (180, 96)
classify.class[2].vupoly[1] = (156, 108)
classify.class[2].vupoly[2] = (162, 136)
classify.class[2].vupoly[3] = (215, 120)
classify.class[2].yupoly[0] = (188, 105)
classify.class[2].yupoly[1] = (183, 132)
classify.class[2].yupoly[2] = (120, 122)
classify.class[4].vupoly[3] = (152, 152)
classify.class[2].yupoly[3] = (125, 106)
classify.class[2].vypoly[0] = (151, 186)
classify.class[2].vypoly[1] = (159, 199)
classify.class[2].vypoly[2] = (206, 131)
classify.class[2].vypoly[3] = (173, 80)

classify.class[3].name = yellow
classify.class[3].color = (255, 255, 0)
classify.class[3].vupoly[0] = (155, 79)
classify.class[3].vupoly[1] = (157, 107)
classify.class[3].vupoly[2] = (125, 114)
classify.class[3].vupoly[3] = (131, 82)
classify.class[3].yupoly[0] = (193, 72)
classify.class[3].yupoly[1] = (159, 124)
classify.class[3].yupoly[2] = (239, 135)
classify.class[3].yupoly[3] = (227, 90)
classify.class[3].vypoly[0] = (123, 239)
classify.class[3].vypoly[1] = (126, 195)
classify.class[3].vypoly[2] = (186, 128)
classify.class[3].vypoly[3] = (200, 142)

classify.class[4].name = blue
classify.class[4].color = (0, 0, 255)
classify.class[4].vupoly[0] = (122, 156)
```

```
classify.class[4].vupoly[1] = (122, 131)
classify.class[4].vupoly[2] = (160, 120)
classify.class[4].vupoly[3] = (152, 152)
classify.class[4].yupoly[0] = (91, 128)
classify.class[4].yupoly[1] = (167, 136)
classify.class[4].yupoly[2] = (155, 159)
classify.class[4].yupoly[3] = (84, 145)
classify.class[4].vypoly[0] = (148, 88)
classify.class[4].vypoly[1] = (152, 164)
classify.class[4].vypoly[2] = (132, 168)
classify.class[4].vypoly[3] = (124, 96)

classify.class[5].name = orange
classify.class[5].color = (255, 127, 0)
classify.class[5].vupoly[0] = (186, 78)
classify.class[5].vupoly[1] = (202, 97)
classify.class[5].vupoly[2] = (145, 119)
classify.class[5].vupoly[3] = (151, 90)
classify.class[5].yupoly[0] = (110, 89)
classify.class[5].yupoly[1] = (106, 104)
classify.class[5].yupoly[2] = (201, 113)
classify.class[5].yupoly[3] = (184, 79)
classify.class[5].vypoly[0] = (150, 203)
classify.class[5].vypoly[1] = (147, 173)
classify.class[5].vypoly[2] = (200, 128)
classify.class[5].vypoly[3] = (199, 142)


# Blob finder
# these are the blob size constraints.  they will depend on
# the size of your fiducials and the distance of the camera.
# These settings can be edited from within mezzcal
blobfind.min_area = 20
blobfind.max_area = 250
blobfind.min_sizex = 5
blobfind.max_sizex = 20
blobfind.min_sizey = 5
blobfind.max_sizey = 20


# Dewarp calibration points
# This defines the world coordinates of your calibation points
# It is best to leave these alone.  We map from these coordinates
# to the eleague coordinates in the eleague_output/UDP program.
dewarp.wpos[0] = (-1.000, -0.500)
dewarp.wpos[1] = (-0.500, -0.500)
dewarp.wpos[2] = (0.000, -0.500)
dewarp.wpos[3] = (0.500, -0.500)
dewarp.wpos[4] = (1.000, -0.500)
dewarp.wpos[5] = (-1.000, 0.000)
dewarp.wpos[6] = (-0.500, 0.000)
dewarp.wpos[7] = (0.000, 0.000)
dewarp.wpos[8] = (0.500, 0.000)
dewarp.wpos[9] = (1.000, 0.000)
```

```
dewarp.wpos[10] = (-1.000, 0.500)
dewarp.wpos[11] = (-0.500, 0.500)
dewarp.wpos[12] = (0.000, 0.500)
dewarp.wpos[13] = (0.500, 0.500)
dewarp.wpos[14] = (1.000, 0.500)
dewarp.wpos[15] = (0.000, -0.250)
dewarp.wpos[16] = (0.000, 0.250)

# These are the image coordinates of the above defined calibation points.
# These were set by dragging the calibration points in mezzcal.
dewarp.ipos[0] = (65, 358)
dewarp.ipos[1] = (155, 385)
dewarp.ipos[2] = (297, 403)
dewarp.ipos[3] = (442, 408)
dewarp.ipos[4] = (565, 397)
dewarp.ipos[5] = (60, 203)
dewarp.ipos[6] = (154, 201)
dewarp.ipos[7] = (296, 200)
dewarp.ipos[8] = (458, 210)
dewarp.ipos[9] = (589, 214)
dewarp.ipos[10] = (84, 53)
dewarp.ipos[11] = (181, 29)
dewarp.ipos[12] = (304, 18)
dewarp.ipos[13] = (452, 25)
dewarp.ipos[14] = (568, 45)
dewarp.ipos[15] = (294, 310)
dewarp.ipos[16] = (301, 100)

# Object identification

# The number of objects
ident.object_count = 5

# The maximum displacement of an object between two consecutive video frames
# mezzanine will only look for an object within this radius of where it
# was last located
ident.max_disp = 0.3

# The maximum blob separation for the two blobs of an object's fiducial.
ident.max_sep = 0.1

# object 4 is an orange ball, notice that ident[4].class[1] is left out.
ident[4].class[0] = 5

# object 0 has a green and blue fiducial.
ident[0].class[0] = 0
ident[0].class[1] = 4

# object 1 has a green and red fiducial
ident[1].class[0] = 0
ident[1].class[1] = 2

# object 2 has a red and blue fiducial
ident[2].class[0] = 2
```

```
ident[2].class[1] = 4

# object 3 has a yellow and blue fiducial
ident[3].class[0] = 3
ident[3].class[1] = 4

# NOTE:
# Two teams of four robots and one ball would require 9 object declarations.
# This configuration file was created for one team and a ball.
```

D References

[1] Andrew Howard. Mezzanine User Manual. <http://playerstage.sourceforge.net/mezzanine/mezzanine.html>, 2002.

[2] John Anderson, Jacky Baltes, David Livingston, Elizabeth Sklar, and Jonah Tower. Toward an Undergradiate League for Robocup. 2003.

[3] Michael Bowling, and Manuela Veloso. Motion Control in CMUnited-98.

[4] Robocup Technical Committee. Sony Four Legged Robot Football League Rule Book. 2003.

[5] Janez Perš, and Stanislac Kovai. Nonparametric, Model-Based Radial Lens Distortion Correction Using Tilted Camera Assumption.