Color Calibration in ERS-210

Carl Axelsson
Jens Törner

October 2003

Examensarbete, 20 p, Institutionen för datavetenskap, Naturvetenskapliga fakulteten; Lunds universitet.

Thesis for a diploma in computer science, 20 credits, Department of Computer science, Faculty of Science; University of Lund, Sweden.

Abstract

This report is a Master's Thesis in Computer Science written at the Department of Computer Science at University of Lund, Sweden. It deals with the color calibration of the Sony ERS-210 robot, also known as Aibo, for use in a RoboCup robotic soccer tournament environment. The results of this thesis are intended to be a part of Team Sweden, the national Swedish effort in RoboCup, but can also be used independently.

The thesis covers improvements and partial automatization of the current color calibration process used by Team Sweden in RoboCup Sony Legged Robot League.

Our work resulted in software called ColorCalibration that can be used to calibrate the color tables of an ERS-210 robot. The software is written in Java and can be acquired free of charge from the Department of Computer Science, University of Lund.

Sammanfattning

Denna rapport är en magisteruppsats i datalogi skriven vid institutionen för datalogi vid Lunds universitet. Den behandlar färgkalibrering av en Sony ERS-210 robot, även känd under namnet ”Aibo”, för användning i RoboCup fotbollsturnering för autonoma robotar. Resultatet av denna magisteruppsats är tänkt att ingå i Team Sweden, det svenska laget i RoboCup, men kan även användas separat.

Arbetet innefattar förbättring och delvis automatisering av befintlig färgkalibreringsprocess använd av Team Sweden i RoboCup Sony Legged Robot League.

Vårt arbete har resulterat i en programvara som vi kallat ColorCalibration. Den kan användas för att kalibrera färgtabellerna i en ERS-210. Programvaran är skriven i Java och kan fritt erhållas från institutionen för datalogi vid Lunds universitet.

Table of Contents

Chapter 1

1. Introduction

This chapter gives a brief introduction to the Aibo robot, the RoboCup soccer tournament initiative, Team Sweden and the importance of a good color calibration. Finally the problem of our thesis is defined.

1.1 The Aibo Robot

There are several different models of the Aibo robot currently on the market. Aibo is therefore not *a* robot but a whole series of robots developed and manufactured by Sony. In this thesis we deal only with the model called ERS-210, since it is the only[1] one allowed in the part of RoboCup known as Sony Legged Robot League, abbreviated SLRL. When we refer to "Aibo" in this report we mean the ERS-210 robot unless otherwise stated.

Figure 1.1. Aibo.

The hardware [6] senses of the Aibo robot are sophisticated and resemble the senses of an animal or a human. It has multiple touch sensors, hearing, sight and an accelerometer to give it a sense of balance. It also has a built-in distance sensor. It is possible to equip it with IEEE 802.11b wireless network capabilities for communication. The ERS-210 Aibo is pictured in Figure 1.1.

The robot has multiple movable joints in the legs, the head, ears and tail, giving the robot a total of 20 degrees of freedom. The core of the Aibo is a MIPS 64 bit RISC processor clocked at 192 MHz. The processor powers Aperios, an object-oriented, distributed operating system. The Aperios operating system can run programs developed using Open-R, a software development kit supplied by

[1] Recently, the Sony ERS-7 has been certified for RoboCup 2004 in Lisbon.

Sony. Programs are compiled using a version of the gcc compiler and then transferred to a memory stick which is inserted into the Aibo robot that runs the program on startup.

The vision capabilities of the Aibo are the primary source of information during RoboCup game play, and are also our sole focus in this thesis. The CMOS camera capability in the Aibo robot is somewhat limited with a maximum resolution of 352 by 288 pixels. But for processor speed reasons a resolution of only 88 by 72 pixels is used by TeamSweden in RoboCup game play. Images used are in color in the YUV color space with a 21 bit color depth[2].

1.2 RoboCup

RoboCup is an international effort to create a standard problem and forum in AI robotics. The application is a soccer game for robots that necessarily deals with robotics, control theory, path planning, real-time artificial intelligence and machine learning. It was started in 1997 and the ultimate goal of the RoboCup initiative is to beat the world champions in soccer with a team of humanoid robots by 2050. Universities from all over the world compete in the five leagues that are RoboCup; Simulation League, Small Size Robot League, Middle Size Robot League, Sony Legged Robot League and Humanoid League.

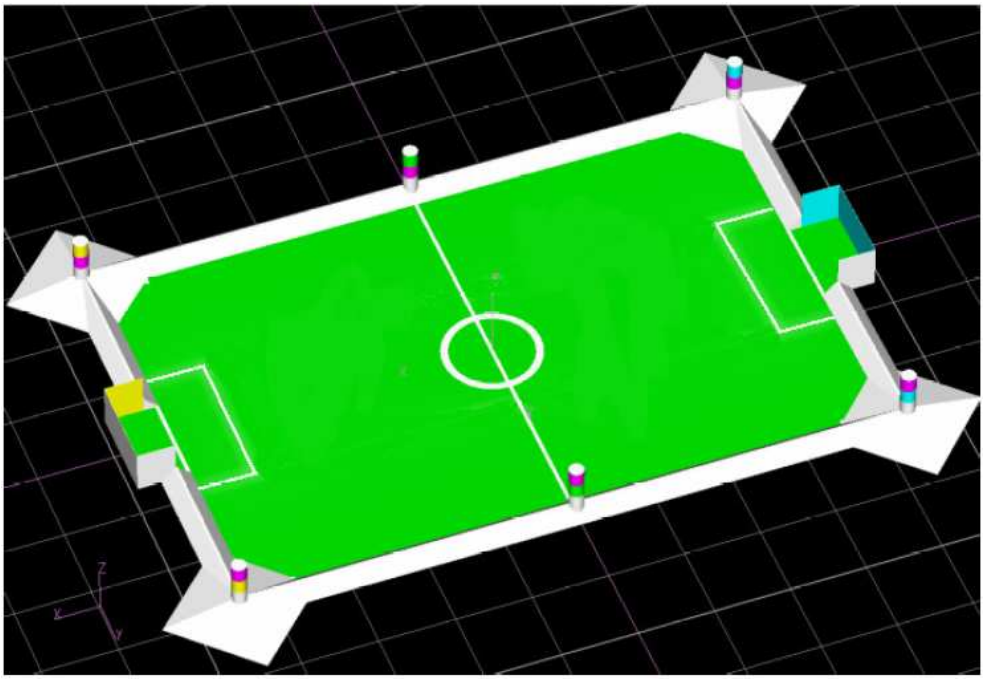


Figure 1.2. The RoboCup SLRL field.

[2] 5 bits Y, 8 bits U and 8 bit V = 21 bits color depth

This report is only concerned with the Sony Legged Robot League (SLRL).

In SLRL the playing field measures 4.6 x 3.1 meters [2], and the robots orient themselves using color coded landmarks along the outer lines of the playing field, pictured in Figure 1.2. There are four robots in each team, and the robots are wearing team colors; either dark blue or red. The ball is bright orange and the goals are colored yellow and sky blue respectively. The playing field is green and the lines and borders are all white. Games are played in two 10 minutes periods with a 10 minute break.

In RoboCup SLRL the robots are totally autonomous, and are not allowed to communicate with anything outside the playing field. They are allowed to communicate with each other via the wireless network. The referee is allowed to communicate with all the participating robots to tell them about for example penalties and when the game is over.

There are also three other tasks to compete in at RoboCup, known as technical challenges. These changes each year and will not be discussed in this report.

1.3 Team Sweden

Team Sweden is the national team representing Sweden in RoboCup. It currently has active members from University of Örebro, Blekinge Institute of Technology, University of Lund and University of Murcia, Spain. The team has participated in RoboCup SLRL since 1999. The four cornerstones of Team Sweden are:

Scientific value: The software should illustrate its scientific approach to autonomous robotics, and demonstrate its research lines in this field.

Generality: The software should embody general principles that are needed to achieve autonomous robot operation, and can be reused in different robots operating in different environments.

Effectiveness: The software should effectively address the specific challenges present in the RoboCup domain in general, and in the legged robot league in particular.

Robustness: The software should degrade smoothly in face of errors and imprecision in perception and execution; in particular, the lower layers should still provide some reasonable response even when higher layers can not compute a reliable course of action.

1.4 The importance of color calibration

In RoboCup the Aibo robots rely heavily on colors in order to find different objects. For the robots to understand the concept of color, a color calibration is needed, defining what color values are what colors. The problem with calibrating is that a color may appear in many shades. For example if one put bright light on an orange object some of it might look like yellow or even white. But the robot must still recognize it as orange. Also if a red object is in front of a yellow object, light reflected from the red object onto the yellow object will most likely result in some pixels on the yellow object being recognized as orange. Since the light setting is never the same at different sites, it is not possible to use a standard calibration and get optimal results.

One way to get around this problem is to use another way of finding objects, without the use of colors. Because of the hardware limitations of the Aibo robot this might be unreasonably slow.

1.5 Problem definition

The goals of the project are the following:

- Improve the calibration process with respect to human-computer interface aspects, compared with the original (PamSim) software;
- In particular decrease the workload of the user;
- Automatize, to the largest possible extent, the process of creating and adjusting the color tables for an ERS-210 robot;
- Make the color calibration system platform independent;
- Facilitate upgrades of the segmentation software imported from the robot and allow easy synchronization between the color calibration tool and the actual software running on the robot.

Chapter 2

2. The earlier software used for calibration

2.1 Working with PamSim

The software used so far by Team Sweden for calibrating the Aibo robots is called PamSimulation or PamSim for short. It is developed by Zbigniew Wasik and Alessandro Saffiotti, both members of the Team Sweden. Pam stands for Perceptual Anchoring Module and is the part of the code developed to run on the Aibo system during RoboCup game play. It contains the color segmentation among other things such as object recognition and other related functions. The name refers to the fact that the PamSim software contains some code identical to code from the Perceptual Anchoring Module. This is because of the

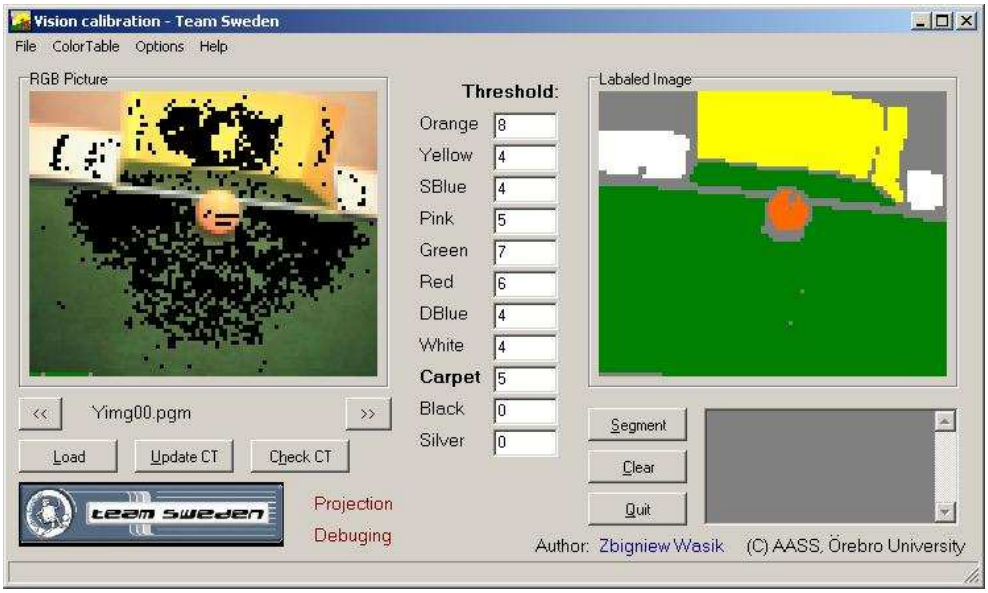


Figure 2.1. The PamSim application.

importance of using the exact same segmentation routine on the robot as in the calibration software.

The PamSim user interface consists of two main windows. One is the image and segmentation window, the other is the color table window. In the image

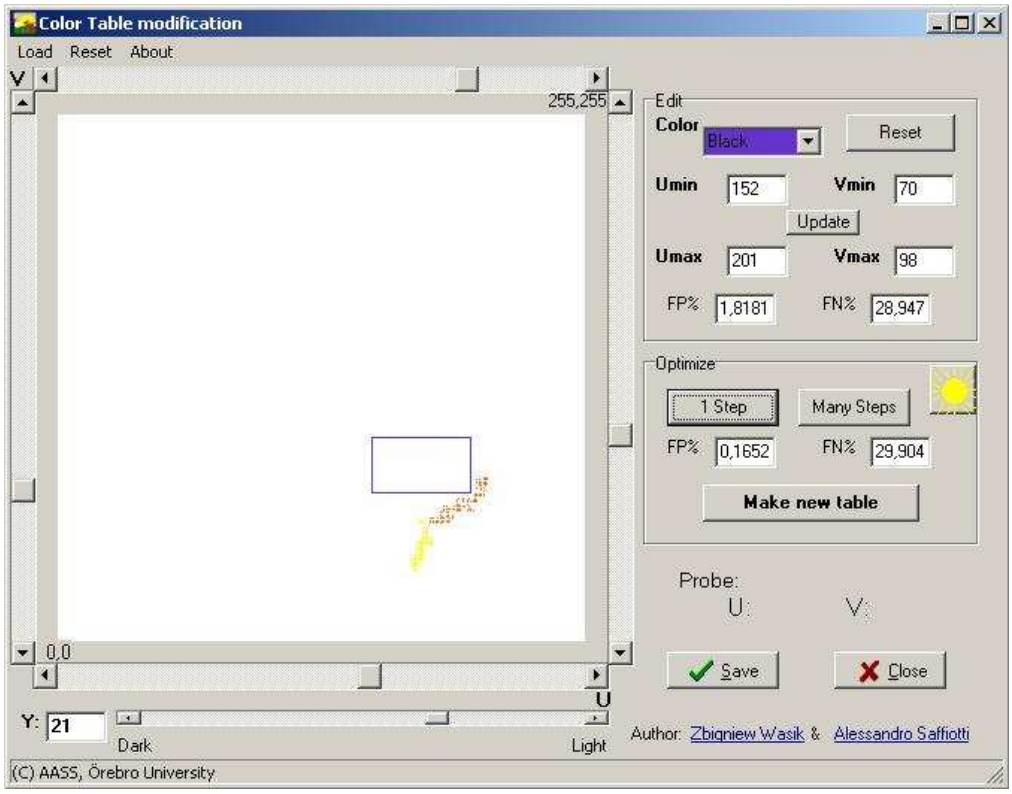
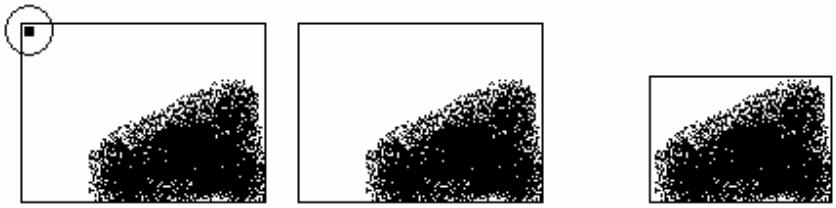


Figure 2.2. The PamSim color table window.

window, shown in Figure 2.1, the user selects which pixels (e.g. color values) in the image are of which of the nine (eleven if you count black and silver, but they are never used) predefined colors. These pixel color values are stored in a color table and, using a threshold for each color, segmentation can be done on the image. This segmentation is displayed next to the actual image and is the same segmentation the robot would have achieved. The user can easily determine if the segmentation is good enough or not.

The other window (see Figure 2.2) shows a graphical representation of the color table, shown to the left. The color domains are shown as boxes representing the U and V values. The boxes may overlap each other resulting in a pixel actually being of two (or more) colors. This problem is solved by giving the colors an order of priority. Another problem is that the colors in a color space are rarely represented by a square. But since the Aibo robots hardware deals with squarely formed chunks of color space, this is the way PamSim does it too.

Since the color table display is only two dimensional, the user has to select the light intencity (Y-value) using a slider (in the range 0 – 31). One very useful feature is the ability to click pixels in the image window and directly see where in the color table they are located. This enables the user to quickly determine if a selected pixel is part of a color value cluster or just a single color value. If it is just a lone value, like the one circled in Figure 2.3a, it might be possible to remove it from the color table without affecting the segmentation much (in Figure 2.3b), but greatly reducing the color domain for that particular Y value, Figure 2.3c.



(a) **(b)** **(c)**

Figure 2.3. The circled color value (a) is removed from the color space (b) resulting in a smalle color area (c)

2.2 Limitations

The current version of the PamSim software is limited to working with only 20 images at a time. This is a severe limitation since calibrating a new playing field usually requires 100 images or more. The user has to build a color table in many steps, making it hard to verify the segmentation as the calibration progresses. Another limitation is the unnecessary number of buttons needed to be pressed by the user during calibration. Typically the user first selects one of the predefined colors, then selects one or many pixels in the image, then presses the "Update CT" to add the color values to the color table, then presses "Check CT" the see how many pixels were selected (e.g., were of the same color value as the ones selected) and finally presses "Segment" to calculate and display a new segmentation. This also does not give the user the advantage of an overview, e.g., all segmented images side by side to determine the quality of the color table on the entire set of images. While this saves a lot of processing power and makes it easy to unselect unwanted pixels, it also slows down the calibration process significantly. This makes the process a very tedious one.

PamSim currently only runs on the Microsoft Windows platform, which is a big disadvantage since the target platform for Team Sweden is Linux. Being able to run on Linux, and possibly other platforms such as Apple's Mac OS X, is therefore one of the most important goals when developing the new calibration software. This goal is a little harder to accomplish by the fact that PamSim is

written using Borland C++ Builder and not ANSI C++, requiring a porting of the source code to ANSI C++. After adding a few things to PamSim for the RoboCup world championship 2003, held in Padova Italy, we stopped working on improving PamSim and started writing new calibration software, this time in Java.

One obvious disadvantage of using Java instead of C or C++, is that all the code developed for the Aibo robot is in C++. And since the segmentation needs to be accurately simulated in order to have a meaningful calibration, this posed one of the biggest problems. The biggest advantage of switching to Java was the platform independence. Another small concern with Java was the speed.

Since PamSim is targeted at a very narrow group of expert users, its illogical user interface might not be a big disadvantage. The total lack of documentation, online help or tool tips is also a disadvantage that might be overlooked. But the main problem is that repeatedly used functions are not easily accessible.

2.3 Input and output

PamSim uses the PGM file format to load images. The PGM file format is basically just a raw non-compressed bitmap format and is used by for example the *Image Observer* program from the Open-R SDK tutorial. Team Sweden uses this program to take pictures to base the calibration on. There are three PGM files to an image, one Y, one U, and one V component.

As for saving the color tables, they are saved as C++ files, with the .cc extension, in order to make them includable in a compilation. The color tables are saved in the format shown in Figure 2.4.

```
// Dog color tables
// Generated by ColorCalibration on Sat Sep 27 21:10:31 CEST
2003
// Threshold: 8, 4, 2, 3, 4, 5, 6, 7, 8, 0, 0,

#define TableSize 32

const unsigned char orange_generic[TableSize*4] =
{
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
  255, 0, 255, 0,
};

const unsigned char yellow_generic[TableSize*4] =
{…};

const unsigned char sblue_generic[TableSize*4] =
{…};

const unsigned char pink_generic[TableSize*4] =
{…};

const unsigned char green_generic[TableSize*4] =
{…};

const unsigned char red_generic[TableSize*4] =
{…};

const unsigned char dblue_generic[TableSize*4] =
{…};

const unsigned char white_generic[TableSize*4] =
{…};

const unsigned char carpet_generic[TableSize*4] =
{…};

const unsigned char black_generic[TableSize*4] =
{…};

const unsigned char silver_generic[TableSize*4] =
{…};
```

Figure 2.4. Color Table save format. **Only orange is fully shown.**

Chapter 3

3. The ColorCalibration Software

3.1 Working with ColorCalibration

PamSim was the initial color calibration software developed by Team Sweden and our first approach was to make PamSim easier to use and to implement new functions in order to make it more efficient. See Figure 3.1.

Our improvements of PamSim are the following:

- A larger RGB window;
- Six smaller RGB representations from the images loaded, ability to change view from RGB to segmentation for the images;
- An easier way of adding color values to the color table;
- Instant segmentation;
- An undo function with channels for each color, giving the user the ability to undo calibration steps for separate colors independently.

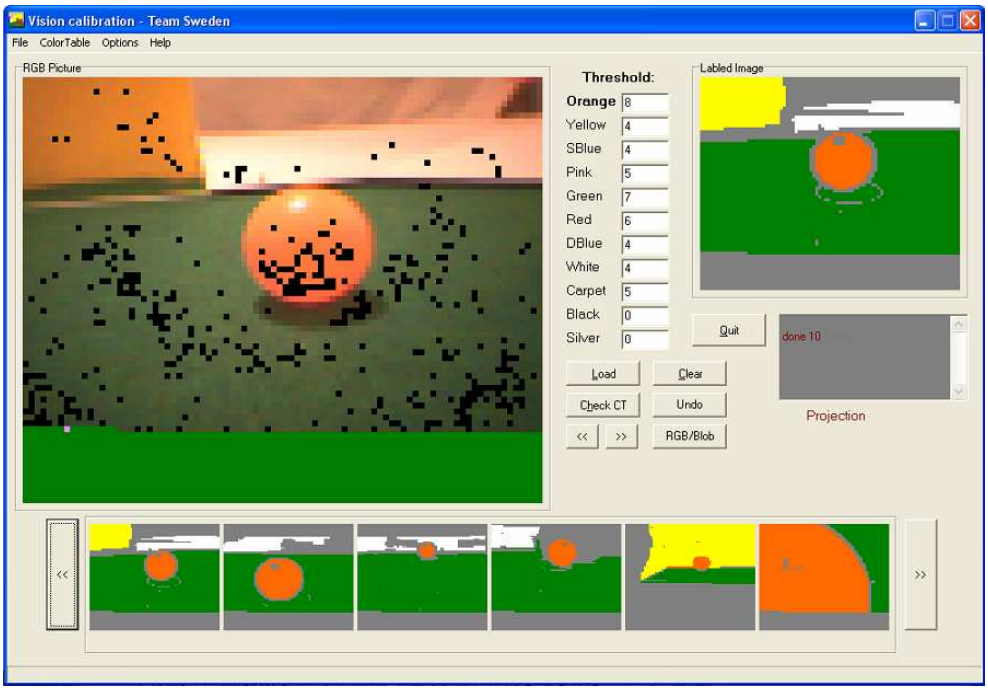


Figure 3.1. PamSim after modifications.

For more information on the difficulties of using the original PamSim application, see Chapter 2.1.

This led us to develop new software, named ColorCalibration. One of the most important features for the software was platform independence. The software needed to run on the Linux platform but a Microsoft Windows version, and possibly one for Mac OS X, was also desirable.

To meet this requirement the software was developed using the Java language, giving platform independence. The Java language is very suitable for graphical interface programming and it is also possible to integrate modules in other languages, i.e., C++.

The structure of the software was rebuilt in a more efficient and easy-to-maintain way. A floating toolbox window, the ability to work on many images in many windows at the same time and an index window showing thumbnails, either in RGB or segmented, of all the images loaded, were some of the features implemented to increase user friendliness. See Figure 3.2.

ColorCalibration software includes:

- Multiple windows for display of more than one image at a time;
- A scrollable window with smaller representations of all the images loaded. The window is resizable so all images can be displayed at the

same time, if screen size allows. This gives the ability to show all images as segmented to get an overview of how good the color table is;

- A toolbox window for easier and more efficient usage of the most commonly used tools;
- A color table window for a practical overview of the color table and an effective way of modifying the color table;
- An automatization process for part automatization of the color calibration;
- The ability to use the native C/C++ segmentation code intended to run on the Aibo robot. The robot hardware is emulated by ColorCalibration.

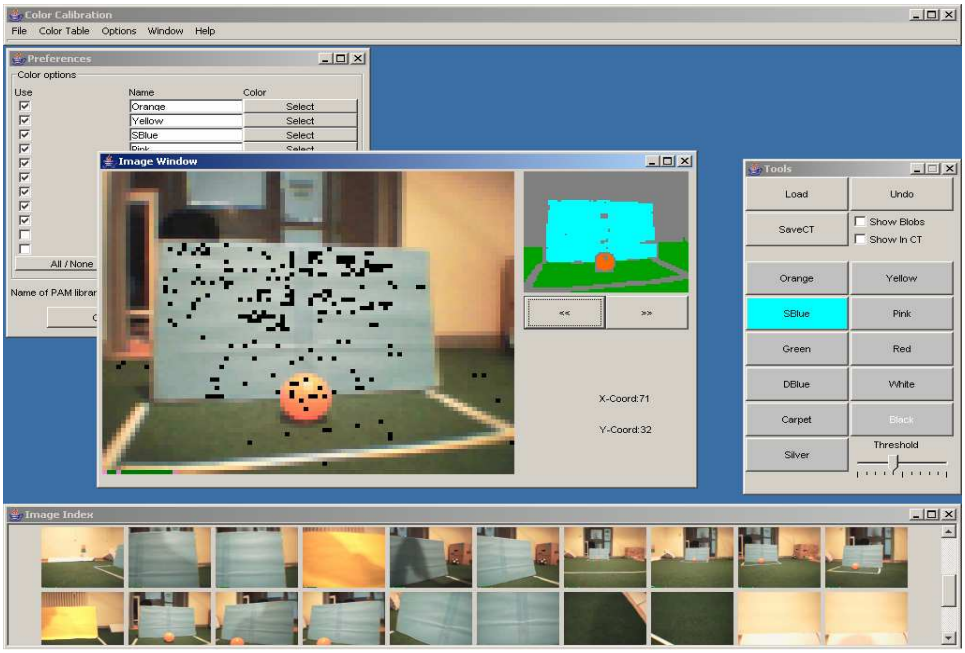


Figure 3.2 ColorCalibration.

For description of all features please refer to the ColorCalibration User Manual in Appendix E.

3.2 Moving from C/C++ to Java

One of the more difficult problems with rewriting the software in Java is the fact that the Java language is typically slower compared to C/C++. It is difficult to benchmark the two against each other but Java is slower on average [11].

When tested on the Microsoft Windows platform the ColorCalibration was slower than PamSim, probably due to the Java switch and a lot more implemented features in ColorCalibration. But in our opinion it was fast enough on a reasonably fast computer.

One of the more critical issues for speed is the segmentation. The ColorCalibration software has the ability to load the native C/C++ segmentation code that could be used in the Aibo robot. This will possibly increase the speed.

If changes in the Aibo robots segmentation code would be made, PamSim could, with some small changes, use that code as its segmentation code. This is more difficult to do in ColorCalibration because it is written in the Java language. To make ColorCalibration useful in the future it has to be able to use the Aibo robots segmentation code directly. A feature in ColorCalibration therefore offers an option to use the native C/C++ Aibo robot segmentation code directly instead of the Java segmentation code. This is done by using wrapper code to convert between native Java and native C/C++ using JNI[3]. See Figure 3.3.

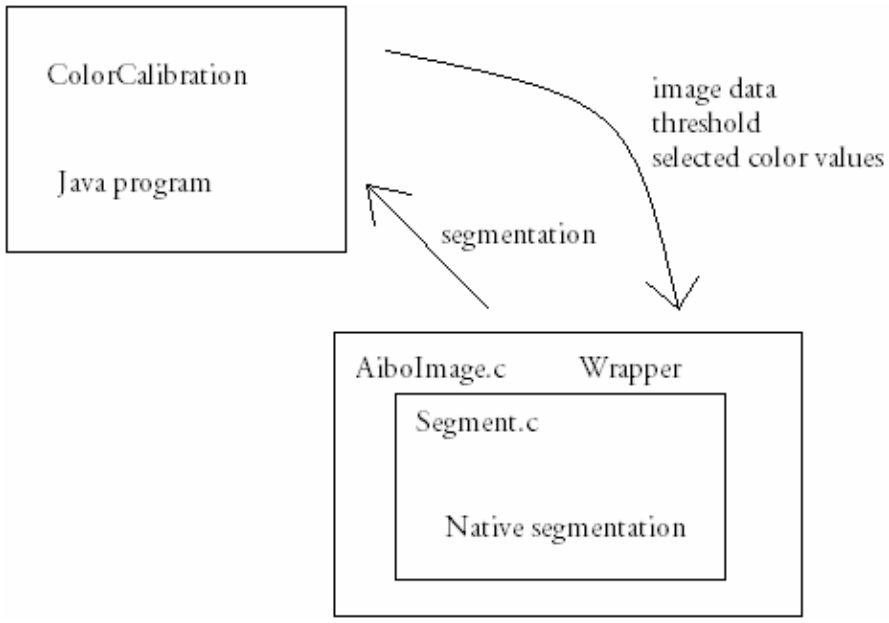


Figure 3.3 The flow of data when native segmentation is used.

[3] See Appendix D; Using the Java Native Interface.

3.3 Backward Compatibility

The ColorCalibration software is backward compatible with PamSim. Old color tables from PamSim are saved in the same way as in ColorCalibration. The ColorCalibration software can therefore read all color tables and vice versa. ColorCalibration can load the same PMG image files as PamSim uses. The new KJI format is *not* supported by PamSim though.

3.4 New and Improved Calibration Tools

ColorCalibration is equipped with a partial automatization tool. The purpose of this tool is to make the manual calibration easier. It does not fully automatize the calibration; however, it extracts the most significant color values from the segmentation of each image. The tool has two options, one that allows getting the most significant color values from each of the loaded images, and one that allows obtaining the most significant color values from all the loaded images. The amount of significant color values chosen can be adjusted. The tool allows the user to manually add color values that could be important but not too common. For example, if an object is partly covered with a shadow, the darker (shadowed) color value might be added to get the entire object. The darker area could, on the other hand, be an object instead of a shadow and the color value should not be added. This is left up to the user to decide.

The color table window in ColorCalibration has been derived from PamSim. It has a similar layout but is improved with easier viewing and modifying of the color table. Also added is a real time color reference to increase the view of what colors are actually represented in the images and a color value intensity scale. It is triggered by the “show pixels” check box and it shows the amount of pixels in the entire set of images corresponding to a certain color value. The higher intensity the darker the color will get for that color value. This feature is very useful when tuning the color table. See Figure 3.4.

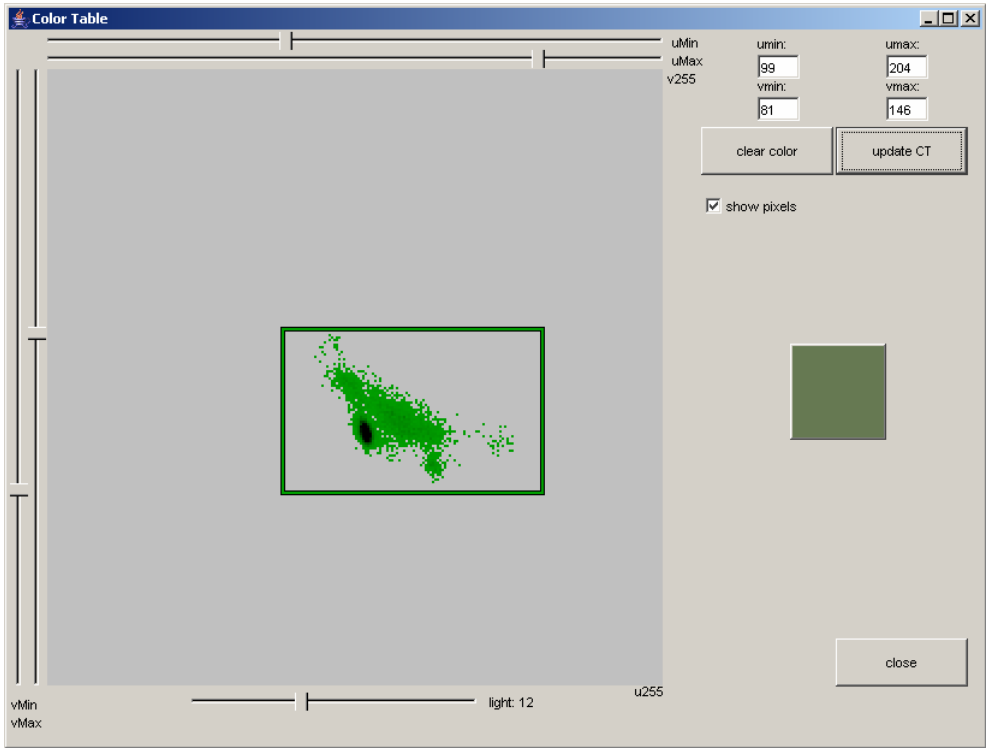


Figure 3.4. Color Table window.

Chapter 4

4. Segmentation

Why are the light conditions so important for the Aibo robot? The robot must have a very good sense of the environment surrounding it to effectively interpret different objects. A human has a very sophisticated way of using her senses to interpret objects and surroundings (visually with color and pattern recognition, touching different objects and in some cases hearing or smelling), the Aibo heavily relies on its visual observations. Another drawback for the Aibo robot is that it only has one camera, resulting in no stereoscopic view. The information a human can collect with her senses is often more than enough to get a clear view of the surroundings. The robot only uses sight and it lacks the depth perception, therefore all the information it can gather is crucial to find objects in its environment, since the IR-based distance sensor is very unreliable. To be able to differ between different light settings is therefore of great importance.

4.1 The drawbacks with different light settings

Different light set on an object can give different color perceptions of the object. For example, if an orange ball, shown in Figure 4.1, is submitted to a strong light source, the top of the ball will be closer to yellow than orange. The Aibo robot will then see an object with both yellow and orange color in it.

Figure 4.1. Ball.

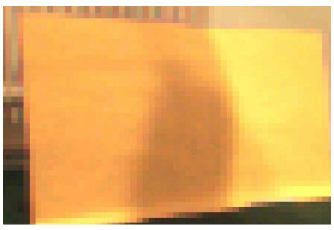


Figure 4.2. Shadow.

The system is also very sensitive to changes, for example, if a shadow appears over an object. This changees the appearance of the object even if it is of the same color, and this could cause the Aibo to not recognize the object.

4.2 Segmentation Methods

There are a few different methods to find objects with the help of pattern recognition. The most common ones are threshold techniques, edge-based methods and region-based methods.

Threshold techniques:
These techniques are based on the idea that pixels with almost the same color (color value) lie within the same object. A threshold sets the margin for how close the color values must be to one another. A big threshold would encapsulate too many color values. This could lead to merging two distinct objects. See Figure 4.3a. If on the other hand the threshold is set too low, only part of the object is found. See Figure 4.3b. For an optimal threshold in this case, see Figure 4.3c.

The threshold technique is very effective on images that blur at the edges of the objects. It is very sensitive to light conditions though. This technique is used most effectively in combination with other techniques.

(a) **(b)** **(c)**

Figure 4.3. Too high threshold, ball merges with goal (a), too low threshold, only part of the ball is found (b). Optimal threshold (c).

Edge-based methods:
These methods are based on the assumption that the pixels on the border surrounding a region differ drastically from the pixels outside the region, finding the edges between regions. A threshold operation is used on a gradient image to determine if an edge is found. When a pixel has been found to be on the edge it must be connected with the other pixels on the edge. This will form a boundary surrounding the region. A common simple technique is to take the difference between two groups of pixels into a high-pass linear filter [10].

The two matrices of the filter combined will be used to find and enhance the edges for all angles in the image. A high value on the constant *c* will enhance the edges, but it also has a tendency to increase noise. To get a reasonable level of noise, typically the value of *c* would be 1 or 2.

Matrices of the filter:

$$H_{horiz} = \begin{bmatrix} -1 & -c & -1 \\ 0 & 0 & 0 \\ 1 & c & 1 \end{bmatrix} \qquad H_{vert} = \begin{bmatrix} 1 & 0 & -1 \\ c & 0 & -c \\ 1 & 0 & -1 \end{bmatrix}$$

An example of simple edge detection is illustrated in Figure 4.4 and 4.5.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	4	4	4	4	4	4	4	4	4	4	4	4	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	4	4	4	4	4	4	4	4	4	4	4	4	4	4	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	5	5	5	5	5	5	5	2	2	2	5	5	5	5	5	5	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	5	5	5	5	5	2	2	2	2	2	2	5	5	5	5	5	5	2	2	2	2	2	2	2	2	2
3	3	3	6	6	6	6	6	6	3	3	3	3	3	3	3	3	3	6	6	6	6	6	6	3	3	3	3	3	3	3
3	3	3	6	6	6	6	6	6	3	3	3	3	3	3	3	3	3	6	6	6	6	6	6	6	6	6	3	3	3	3
4	4	4	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	4	4	4
4	4	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	4	4
5	5	8	8	8	8	8	8	8	5	5	5	5	5	5	5	5	5	5	5	5	5	8	8	8	8	8	8	8	8	5
5	8	8	8	8	8	8	8	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	8	8	8	8	8	8	8
6	9	9	9	9	9	9	9	9	9	9	6	6	6	6	6	6	6	6	6	6	9	9	9	9	9	9	9	9	9	9
6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7

Figure 4.4 Can you see the letter?

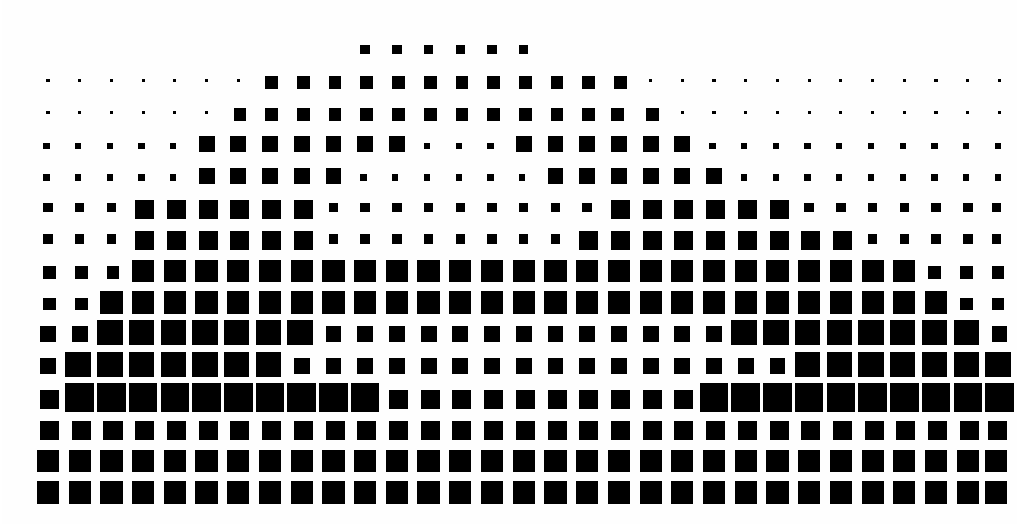
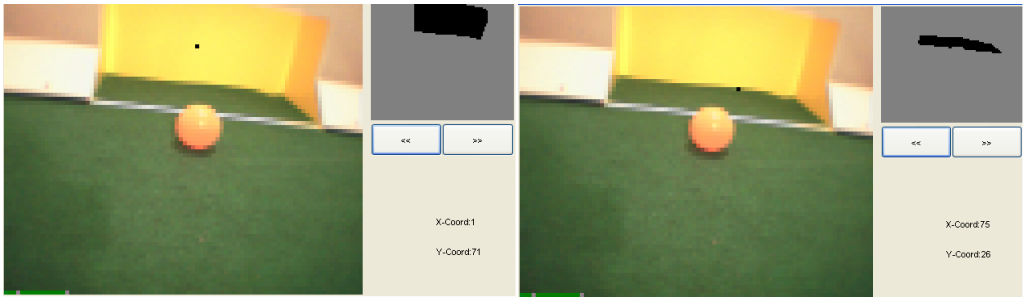


Figure 4.5 Clarifying view of the letter.

Special edge detectors such as Canny [4] and Susan[4] [5] are used for a more advanced detection. Often the image is post-processed in combination with a detector to get the best result. Edge-based methods are not sensitive to light and are therefore highly reliable. The processes of edge-based methods are very computationally expensive though, while the Aibo robots' hardware is very limited. Therefore Aibo robots have no use for this method as it is today. It could be used in the calibration but it would require different segmentation in the ColorCalibration and on the Aibo. This might change in the future as the robots hardware improves.

Region-based methods:
The edge-based methods and the threshold techniques are based on differences of pixel values. The region-based method tries to find regions with the same color value. The Seed Region Growing (SRG) method is a region-based method which will use a few pixels as start pixels to grow regions with the same color values. These start pixels are called seeds and the regions they build will be determined by a threshold value. The seed will first start to grow by checking its closest neighboring pixels and determine if they have close enough color value to be included in the region. The threshold value will determine how large range of color values will be included in the region.

To automatically choose seeds is a difficult task, because the seed must be representative for the region that should be grown, as in Figure 4.6a. If a bad seed is chosen, the region it will grow will most likely be only a fragment of the desired region, i.e., if a seed is chosen from a blurry edge of an object, the growing region could contain only that seed or even worse, a not desired region, as shown in Figure 4.6b.



(a) **(b)**

Figure 4.6. Growing from good seed (a), growing from bad seed (b).

This method is not very computationally expensive and is therefore very suitable for the Aibo robots' limited hardware capabilities.

[4] Smallest Univalue Segment Assimilating Nucleus

Segmentation on the Aibo

The robots use a color segmentation that integrates the threshold and the SRG method. The threshold technique is used to generate a set of initial seeds. There is a different set of seeds for all interesting colors. The SRG method is then used on the initial seeds to grow the desired color regions. The SRG can be adjusted for more than one seed in the same region. To be able to use the best qualities from both methods the segmentation process improves the lighting conditions because we use a conservative threshold and blurred edges because we use a conservative homogeneity criterion in SRG. However we still inherit the problem with a too high threshold merging objects (Figure 4.3a), and a too low threshold not giving the entire object (Figure 4.3b) and also the sensitivity with the seed pixels. To minimize these weaknesses, the Aibo uses more than one pixel in each object in combination with a low threshold. This results in many different segments, called blobs, in the same object. A blob does not have to include only one seed. The blobs from the same object are then merged into one region and build the object.

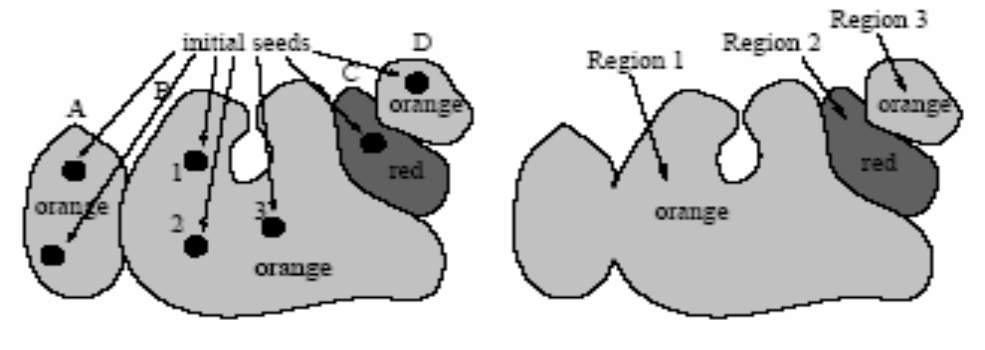


Figure 4.7 Merging blobs grown from multiple seeds into regions.

Blob A and blob B are two blobs with the same color grown from different seeds belonging to the same region. After merging, blob A and B have become region 1. However, the orange blobs A, B and D will never merge because of the red blob C which lies in between. Example from [3] is shown in Figure 4.7.

How is a blob found? The original seed is compared to its 8 connected neighbors. The neighboring pixels that satisfy the criterion of how close a color value should be to be included with the original seed will be added to the blob. This neighbor comparison step is repeated for every new pixel assigned to the blob until the blob is completely bounded by the edge of the image or by pixels that do not satisfy the criterion. The blob is now found. The algorithm has originally appeared in [1].

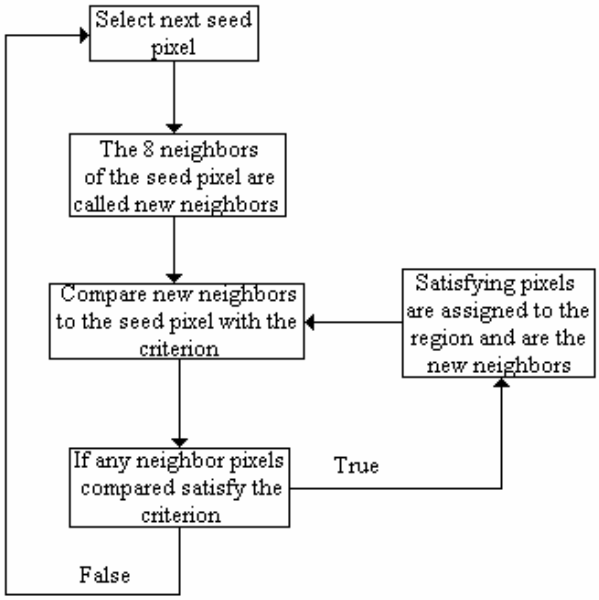
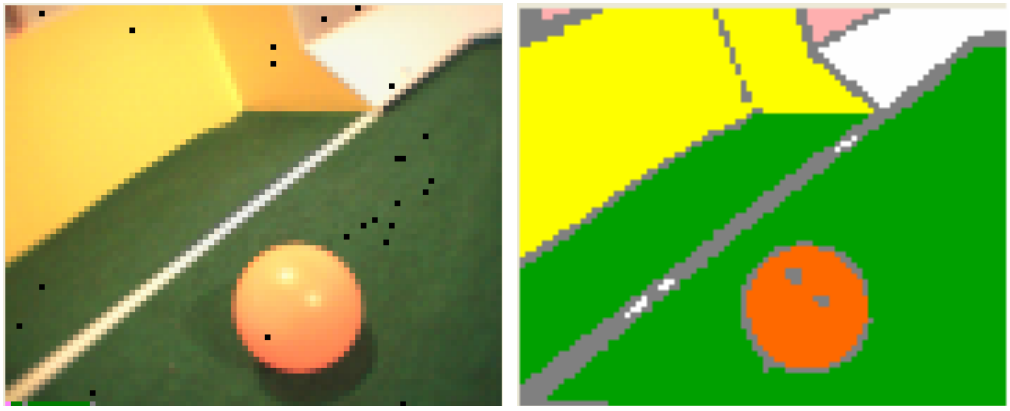


Figure 4.8 Segment building flow chart.

The final result from merging the blobs into regions could look as shown in Figure 4.9.



(a) **(b)**

Figure 4.9. The original image (a), and the segmentation (b).

The technique of merging color segmentation and the SRG method has proven to be very successful in the RoboCup environment.

Chapter 5

5. Automatization

5.1 Introduction

The calibration of the Aibo robot is currently done by hand. The user must add every color value manually to a color table by selecting a pixel with the desired color value. The color table is the table consisting of all the color values the user has chosen. When the calibration is done, the color table is used on the Aibo robot. The Aibo robots will use it as a reference to find objects and guide it through the surrounding environment. The output from the ColorCalibration program is such a color table. The entire goal of the program is to make this color table as good as possible. But the job to manually choose all the right color values is very monotonous.

It is important that the user chooses a color value which is representative for the color chosen. If the color value is poorly chosen, an object could appear in one color on the first image and in another color or not at all on the next image. This means it is not enough to just know one object and its color. The robot can not automatically adjust the color value for different environment changes, for example, stronger light or shadows. Changes like these will make it very difficult to automatically predict the pixel with the most representative color value.

A human finds an object mainly by recognizing patterns and shapes. The color and texture of the object is then determined by using the human ability to interpret the reflection of the light from an object. This can change the appearance of the object's real color. For example, if the color of an object is known to be orange but the light of the surroundings makes it appear yellow, the human brain would most likely interpret the color of the object as orange.

The color surrounding an object could also make it appear in a slightly different color. For example, if an orange object has a bright yellow object close to it, it would appear more yellow. The Aibo must work in the same way, i.e., if the color of an orange object changes to yellow, the robot must still interpret the color of the object as orange. This is why it is so difficult to make an automatization of the color calibration.

5.2 Our Automatization method

The purpose of the partial automatization of this process is to help the user with the tiresome work of manually choosing all the color values. This automatization will not completely remove the user's role in the calibration. Some parts of the calibration are too important to be made automatically. The method is also intended to make the process of calibration as stable and effective as possible, without removing the user's influence on the process.

An object in an image consists of a set of pixels with the same region of color values (a threshold value determines how far away the color values can be from each other to be in the same region). An object can consist of many pixels with exactly the same color value. If a color value represents many pixels in an object that color value would most likely represent the color very well. This color value may then be used as a seed to grow a segment of the object. More details are provided in chapter 4.

The color table should include as few color values as possible and still be able to find all the objects. The automatization in the ColorCalibration program is therefore based on finding the color values that represent the largest amount of pixels in an object. This will generate a histogram over the most significant color values in the object. See Figure 5.1.

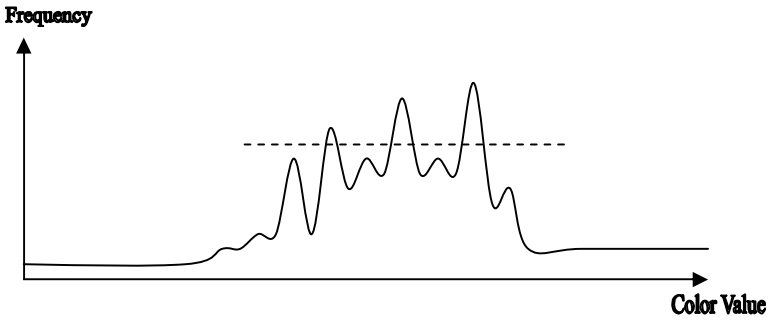


Figure 5.1 A histogram showing the frequencies for all color values of one color

There are some problems with this approach though. The light on an object can differ from image to image. Even if a color value is found that represents many pixels in the object it might not represent a good color value for that object if many images are considered. For example if a color value represents 40 pixels in an object, but only in one of 50 images, that color value is not very representative for that object. On the other hand it could be very important to find the object on that specific image.

We have solved this problem in ColorCalibration by providing the user a configuration screen. The user is able to take the most significant color values from the segmented objects for each image one at a time, or to get the most significant color values from the segmented objects for a whole set of images. The user can also decide how many color values should be taken from each, starting with the most significant color value.

To get color values from the segmented object for each image tends to generate a lot of color values (depending on how many color values the user has chosen to get), but on the other hand the object will be found on most images containing the object. To obtain color values from the segmented object for a whole set of images generates fewer color values, but very representative for all images.

A drawback with our histogram method is that an initial seed must be chosen for each color. This seed is the reference for the color. Only one seed is needed, although more seeds could be useful for a more accurate automatization. The user can also choose to load a color table from disk and use its color values as seeds to automatize.

5.3 Other methods for automatization

- *Random method:* This method uses an initial seed as in the histogram method, but instead of choosing the most significant color value it will randomly find color values from the segmented object. This will lead to color values with random significance, i.e., a color value could be added to the color table only reflecting one pixel in all the images. When tested this method added a very large number of color values to the color table, including other objects with similar color values. This method is very comprehensive but not very useful. It is not very stable because it will generate more color values every time automatization is run. The method is not implemented in ColorCalibration.

- *Pattern recognition:* This method is based on pattern recognition used to find the objects. When an object is found the histogram method could be used to add the right color values to the color table. The advantage with

this method is that it does not need an initial seed, making it more automatic than the histogram method. However the method is unstable. If, for example, the process must differ between two objects with different colors but of the same shape. The objects are found with the pattern recognition but not their colors. This could be determined only if the objects colors differ heavily.

The pattern recognition could have a problem with parts of objects looking like other objects. For example, if a piece of ground with the color dark green is identical to the shape of a dark blue object, then the pattern recognition method might interpret the ground as the dark blue object because of the similarity of the colors.

This method is very hard to implement. Making it good and stable enough to find all types of objects is out of the scope of this thesis. It is one of the most important features for future implementation, see 6.2.

Chapter 6

6. Conclusions

6.1 Conclusions

The purpose of our thesis is to improve the calibration process for the ERS-210 Aibo robot. The current tool used by Team Sweden is unintuitive and hard to use, making our first and foremost goal to produce easy-to-use, functional software.

With new and ongoing research on the Aibo, the calibration process needs to be able to adapt to new demands. The calibration method can therefore never be considered fully developed; there will always be room for improvements. We have tried to make the software for the calibration as easy to maintain as possible and we hope this software will be helpful for Team Sweden in the future.

The problem definition stated that the software should improve the calibration process for the user and as far as possible automatize the color calibration. The calibration process has improved in our opinion. It is now easier to get a good overview of the work in progress. It is easier to correct mistakes and to maintain control of the final result. An automatization of the calibration is implemented, partially removing some of the manual work of calibrating the color tables.

One of the more important requirements for the software was platform independence. This is achieved by using the Java language. If the Java Native Interface is used, the software is no longer platform independent, but able to use the exact same segmentation routines that the Aibo robot does. For future development of the segmentation software on the robot, the JNI option is very useful and the user always has the ability to use the native Java segmentation as a backup.

6.2 Possible future developments

The Aibo robot is constantly upgraded with faster and better hardware and OPEN-R library routines. This must reflect the calibration software as well. The segmentation process currently used is a color segmentation method that integrates the threshold and the SRG method. A better way of doing the segmentation would be to use an advanced edge detection method in combination with a color recognition method. This would remove the difficult problem caused by changes in light conditions. A more powerful pattern recognition algorithm would also be very desirable, as it would partially remove the need of a color calibration or could be used during the calibration itself. The hardware in the Aibo robot is too limited at the moment though.

Pattern recognition could also be used to remove the need for seeds in our automatization process.

A connection between the Aibo robot and the calibration software could also be very useful. It would be possible to see the images from the Aibo camera and use a created color table to display the segmentation in real-time. This would speed up the calibration process significantly.

References

[1] N. Ikonomakis, K. Plataniotis, A. Venetsanopoulos: User Interaction in Region-Based Color Image Segmentation. VISUAL'99, LNCS 1614 pages 99-106, Springer Verlag, 1999

[2] RoboCup Technical Committee: Sony Four Legged Robot Football League Rule Book, 2003 http://www.robocup.org (Verified Oct 13th 2003)

[3] Z. Wasik, A. Saffiotti: Robust Color Segmentation for the RoboCup Domain. Proc. of the Int. Conf. on Pattern Recognition (ICPR), volume 2, pages 651-654, 2002.

[4] J Canny: A Computational Approach to Edge Detection, IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), volume 8, no 6, 1986

[5] S. Smith: SUSAN - A New Approach to Low Level Image Processing, Technical Report TR95SMS1c, Oxford University, 1995

[6] Sony Corporation: Entertainment Robot AIBO Operating Instructions, Sony Corporation, 2000

[7] Web site of the Java language: http://java.sun.com (Verified Oct 1st 2003)

[8] The Operating Color Space: http://www.canopus.com/US/pdf/Storm_comparison.pdf (Verified Oct 1st 2003)

[9] D. Farin: Praktikum Multimediatechnik. http://www.informatik.uni-mannheim.de/informatik/pi4/stud/veranstaltungen/ss2003/multimedia/h261teil2.pdf (Verified Oct 1st 2003)

[10] T. Pavlidis: Algorithms for Graphics and Image Processing. Springer Verlag, 1982

[11] Linux Number Crunching: Benchmarking Compilers and Languages http://www.coyotegulch.com/reviews/almabench.html (Verified Oct 11th 2003)

Appendix A: Dictionary

Aibo	Artificial Intelligence roBOt – Aibo is a four legged robot manufactured by Sony. Also, “Aibo” in Japanese means companion. See appendix B.
ERS-210	See Aibo.
AI	Artificial Intelligence.
Color Space	A coordinate system with the different axes representing different color components, i.e., R, G and B, or Y, U and V. See Appendix C.
Histogram	A table presenting the frequency of an occurrence, i.e., a value.
Calibrate	Adjust something to fit a purpose or environment.
RGB	A way to represent colors by their Red, Green and Blue components. See Appendix C.
YUV	Also known as YCbCr. Colors represented by their overall brightness (Y), the blue component (U) and the red component (V). Used for example in television. See Appendix C.
Color table	A list with color values assigned to a specific color.
IEEE 802.11	A standard for wireless network communication.
Linux	A UNIX-like operating system.
PAM	Perceptual Anchoring Module – part of the Team Sweden software for Aibo.
ColorCalibration	The calibration software that resulted from this thesis work.

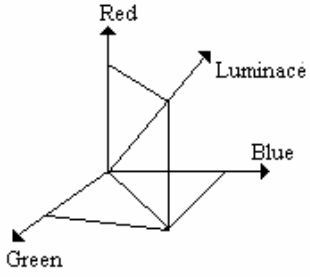
Appendix B: The Sony ERS-210 Specifications

CPU	64-bit RISC by MIPS @ 192MHz
Main Memory	32 MB
Removable Media	Sony Memory Stick
Movable parts	Mouth: 1 degree of freedom Head: 3 degrees of freedom Leg: 3 degrees of freedom x 4 = 12 Ear: 1 degree of freedom x 2 = 2 Tail: 2 degrees of freedom Total: 20 degrees of freedom
Input	PC Card type II slot Battery charger
Video	100k pixel CMOS image sensor
Audio	Stereo microphone Speaker
Sensors	Thermometric sensor Infrared distance sensor Acceleration sensor (3-way) Pressure sensor (head, chin, back and paws) Vibration sensor
Operable duration	Approx. 1.5 hours fully charged
Dimensions	152 x 281 x 250 mm (w/h/d not including ears or tail)
Mass	Approx. 1.5 kg with battery

ref. [6]

Appendix C: The RGB and YUV Color Spaces

The Red, Green and Blue (RGB) standard is one of the most common color space standards. The combination of these three colors can produce almost all colors visual to the human eye.



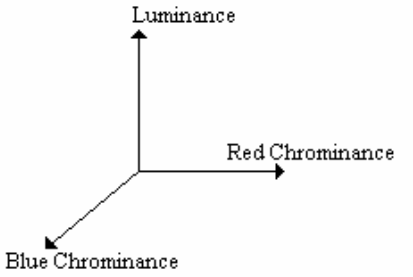


Figure C.1a RGB **Figure C.1b YUV**

Instead of using three equally weighted color components like the RGB standard, see Figure C.1a, the YUV uses one luminance/brightness component Y and the U, V components as chrominance/color [8], as in Figure C.1b. The YUV values are created from RGB. If the R, G and B components are weighted and added together they will produce a Y component which represents the overall brightness. The U component is created by taking the weighted R, G and B components in Y and subtracts the B component. The V component is created in the same way, except that in this case subtraction will be with the R component instead of the B component.

Conversion (in 24 bit color depth) between RGB and YUV is easily done with the following formula [9]:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1.37 \\ 1 & -0.34 & -0.70 \\ 1 & 1.73 & 0 \end{pmatrix} \begin{pmatrix} Y-16 \\ U-128 \\ V-128 \end{pmatrix}$$

Appendix D: Using the Java Native Interface

When we decided to use Java to develop ColorCalibration, one obvious obstacle was the segmentation code. The segmentation code, a part of PAM, is a piece of C++ code that is run on the Aibo robot and does the segmentation. This code is present, with some very minor adjustments to make it run, in PamSim, and must also be included in our program. One way to solve the problem is to translate the C++ code to Java. But a translation can not be guaranteed to be exact, so while it is a good solution it is sometimes not good enough. The other option was to run the compiled C++ code directly from Java. Java offers support for this via the Java Native Interface (JNI), but it is a bit awkward to use, and there is not much documentation. We implemented both native Java segmentation and the ability to use the JNI, keeping our software platform independent.

D1 Java Native Interface types

JNI is basically a set of wrappers to convert between native java primitives/objects and native C/C++ primitives/objects. These can then be converted or type casted. JNI defines eight primitive types [7]. See Table D.1.

Java primitive type	Native C/C++ type	Description
boolean	jboolean	Unsigned 8 bits
byte	jbyte	Signed 8 bits
char	jchar	Unsigned 16 bits
short	jshort	Signed 16 bits
int	jint	Signed 32 bits
long	jlong	Signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits

Table D.1. Primitive types in JNI and C/C++.

JNI also defines twelve other types in order to cover all the java reference types, as shown in Table D.2.

Java reference type	**Native C/C++ reference type**	**Description**
java.lang.Object	jobject	Java object instances
java.lang.Class	jclass	Java class instances
String	jstring	Strings
Array	jarray	Arrays
-boolean[]	jbooleanArray	Arrays of boolean
-byte[]	jbyteArray	Arrays of byte
-char[]	jcharArray	Arrays of char
-int[]	jintArray	Arrays of int
-long[]	jlongArray	Arrays of long
-float[]	jfloatArray	Arrays of float
-double[]	jdoubleArray	Arrays of double
java.lang.Throwable	jthrowable	Throwable objects

Table D.2. Java reference types and their C/C++ counterparts.

All reference types are subtypes of the jobject type. When used in the C++ programming language, the JNI introduces a set of dummy classes to express the subtyping relationship among the various reference types, as shown in Figure D.1.

```
class _jobject {};
class _jclass : public _jobject {};
class _jthrowable : public _jobject {};
class _jstring : public _jobject {};
class _jarray : public _jobject {};
class _jbooleanArray : public _jarray {};
class _jbyteArray : public _jarray {};
class _jcharArray : public _jarray {};
class _jshortArray : public _jarray {};
class _jintArray : public _jarray {};
class _jlongArray : public _jarray {};
class _jfloatArray : public _jarray {};
class _jdoubleArray : public _jarray {};
class _jobjectArray : public _jarray {};

typedef _jobject *jobject;
typedef _jclass *jclass;
typedef _jthrowable *jthrowable;
typedef _jstring *jstring;
typedef _jarray *jarray;
typedef _jbooleanArray *jbooleanArray;
typedef _jbyteArray *jbyteArray;
typedef _jcharArray *jcharArray;
typedef _jshortArray *jshortArray;
typedef _jintArray *jintArray;
typedef _jlongArray *jlongArray;
typedef _jfloatArray *jfloatArray;
typedef _jdoubleArray *jdoubleArray;


typedef _jobjectArray *jobjectArray;
```

Figure D.1. JNI dummy classes.

With these types available it is easy to write one's own wrappers to integrate the native C/C++ code with the Java program. The platform independence is however lost.

D2 A small example

In order to illustrate how to work with the JNI, here is a small "Hello World" example.

First the native method is located using a *System.loadLibrary* call. Then the native method needs to be declared using an abstract method. Finally the method is called in the *main* method. See Figure D.2.

Java code "*HelloWorld.java*"

```
class HelloWorld {

     static {
         System.loadLibrary("Hello"); // load native routines
     }

     private native void hello(); // the native method call

     public static void main(String[] args) {
         new HelloWorld().hello(); // call the native method
     }
}
```

Figure D.2

First the Java code is compiled. The Java header generator, *javah*, is then used on the class file to generate a C header:

```
javah –jni HelloWorld
```

This will generate a C header file containing the method prototype:

```
JNIEXPORT void JNICALL
Java_HelloWorld_hello (JNIEnv *, jobject);
```

C code with the same call parameters as in the method prototype is needed, as shown in Figure D.3.

C code "*hello.c*"

```
#include <jni.h>
#include <stdio.h>
#include "HelloWorld.h"

JNIEXPORT void JNICALL
Java_HelloWorld_hello(JNIEnv *env, jobject obj)
{
    printf("Hello World!\n");
    return;
}
```

Figure D.3

Finally, the C code is compiled into a system library (i.e. *libHello.so* on UNIX systems and *hello.dll* on Microsoft Windows systems) and run within the java program. The compiled C code should be placed in the system library path.

Appendix E: ColorCalibration User Manual

ColorCalibration

Software for color calibration of Sony Aibo robots in a RoboCup environment

By
Jens Törner
Carl Axelsson

USER MANUAL

Table of contents

1 Introduction

ColorCalibration is developed with a RoboCup environment in mind but can be reconfigured for other color calibration needs. It will let you calibrate a color table for use with your Sony Aibo robot with little effort. It provides powerful tools to view, modify and automatize the calibration.

Previous experience with Sony Aibo robots, RoboCup and Open-R programming is assumed throughout this user manual.

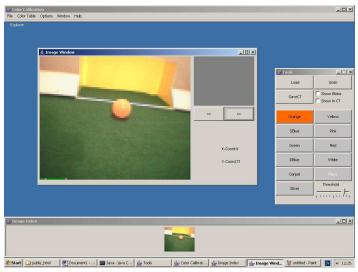
If you are a first time user please browse through the Quickstart section of this manual to get started as soon as possible.

Thank you for choosing ColorCalibration.

1.1 Quickstart

Install the program files in a directory. Start up the ColorCalibration program with:

```
java ColorCalibration
```



When the program is running, click the “Load” button in the toolbox to open up the load file dialog. Select the *quickstart.kji* image included in the ColorCalibration distribution and click load. The image is loaded and displayed as a thumbnail in the image index window at the bottom of the screen. Single click on the image to open it up in an image window. Your screen should now look something like the screen pictured to the left.

Select some pixels from the ball by clicking on them in the image window, and watch how the segment grows. Select other colors from the toolbox and try to make the segmented image as good as possible. If you make a mistake, you can always click “Undo” to delete your last selection. But remember that the undo function works with channels, one for each color, so you have to be in the right color in order to undo. There are twenty steps of undo in each channel.

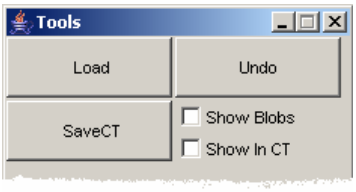
When you are satisfied with the segmentation, click the "Save CT" button in the toolbox to save your color table. Just press ok when the program asks for suffix.

ColorCalibration is so much more, but you should now have a general idea of how to calibrate your Aibo!

2 Working with ColorCalibration

2.1 The Toolbox

Here you will find the most commonly used tools and operations. The toolbox resides in a window of its own, and can easily be placed anywhere on the screen you find convenient.



Load – Opens the load image dialog which lets you select and load one or more images from the hard drive or directly from your Memory Stick reader. Images loaded are displayed in the image index window on the bottom of your screen.

Save CT – Opens the save color table dialog which lets you specify a filename and path to save the color table. You will be prompted for how to label the color table arrays, just press ok to save it as "generic". For more information on the format color tables are saved in, please see 3.3, *Color Table file format.*

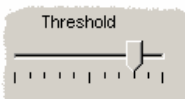
Undo – Enables you to erase your last selected pixel from your color table. Undo works with channels, so only pixels selected in the color currently active is erased. To undo a selection in another color, simply select that color in the toolbox and press undo the desired number of times. There are twenty steps of undo in each channel. Please note that the results from undo will not be visible until the image window is in focus. Undo effects all images, not just the one(s) currently displayed.

Show Blobs – Checking this option displays all the images in the index window as segmented. This is a great way to get an overview of how good your color table is at the moment. Please note that keeping this option checked at all time might drastically reduce program performance.

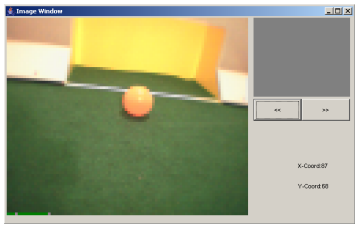
Show In CT – Checking this option lets you click a pixel in an image and see where in the color space it is located. This is a powerful tool to find stray pixels in your color table.

Color palette – Clicking on these selects which color you are currently working with. The button is colored in the

active color. This also affects which undo channel is used.



Threshold – This slider shows the threshold value for the current color. Just drag it to change the threshold. Thresholds affect how many pixels will be included when growing the segments. Thresholds are saved with the color table.

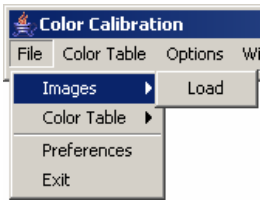


2.2 Image window

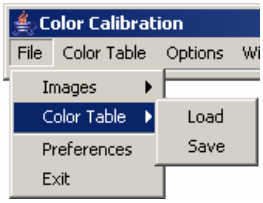
The image window is the main working window. In this window you click to select pixels (color values) to be included in your color table. The small display to the right in the window shows the segmented version of the image. Clicking on the buttons with arrows displays the next/previous image.

2.3 Menus

The menus contain many options not available anywhere else in ColorCalibration. There are five main categories: File, Color Table, Options, Window and Help.



File – Images – Load – Opens the load image dialog which lets you select and load one or more images from your hard drive or directly from your Memory Stick reader. Images loaded are displayed as thumbnails in the image index window on the bottom of your screen. Single click on a thumbnail to open it up in a new window.

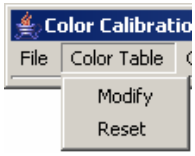


File – Color Table – Load – Opens the load color table dialog which lets you load a previously saved color table.

File – Color Table – Save – Opens the save color table dialog which lets you specify a filename and path to save your color table. You will be prompted for how to label the color table arrays, just press ok to save it as “generic”. For more information on the format color tables are saved in, please see 3.3, *Color Table file format.*

File – Preferences – Opens the preferences dialog. For more information on preferences settings, please see 2.7, *Preferences*.

File – Exit – Exits the program.



Color Table – Modify – Opens the color table window. This lets you see how the color values in your color table are distributed in the color space. For more information on how to work with the color table window, please see 2.5, *Color Table*.

Color Table – Reset – Resets the color table. Please be careful with this function. You must undo for all the color channels to get your color table back.

Options – Automatization – Opens the automatization window. For more information on the automatization process, please see 2.6, *Automatization*.

Window – Show Toolbox – Opens the toolbox window if it is not already on open.

Window – Show Image Index – Opens a new image index.

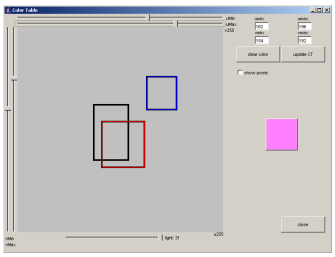
Help – Help – Shows this manual.

Help – About – Shows info about ColorCalibration.

2.4 Image index

The image index window shows small, thumbnail size, versions of all images loaded. Images can be displayed as regular images or segmented ones, based on the current color table. Just check “Show Blobs” in the toolbox to show segmented images. Please note that updating many segmented images might drastically reduce program performance.

2.5 Color table



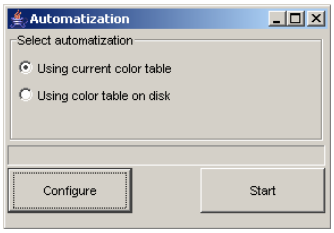
The color table window shows how the color values in the images loaded and in the current color table are distributed in the YUV color space. There are 32 levels of Y (light), which one is displayed is indicated by the “light” slider. For each Y value, the U and V values in the current color table are shown by a box in the color space area. If the “show pixels” option is checked, all color values that are actually in the images are also plotted in each box, with darker colors representing a higher density. Boxes showing the color value boundaries may overlap each other, and in that case the color with the highest priority takes precedence. The colors are ordered

in the same order they are displayed in the toolbox palette, with the default order being orange, yellow, sky blue, pink, green, red, dark blue, white, carpet, black and silver where orange has the highest priority. Priority rules are the same for the “show pixels” option, in case of overlapping.

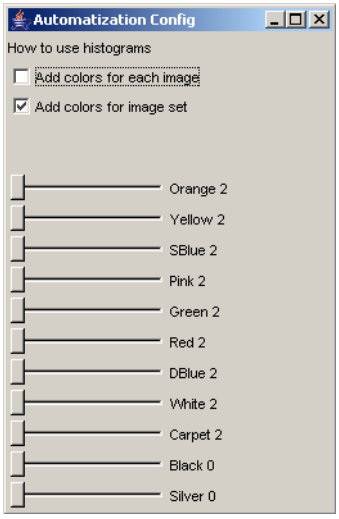
When the mouse is moved around in the color space area the color directly under the mouse pointer is displayed to the right. Also displayed to the right are the actual U- and V values for the color currently selected. The button “Clear color” clears the color boundaries for the color currently selected and only for the current Y (light) value. At all times you need to press the “Update CT” buttons for the changes to take effect.

Clicking the button “Close” closes the color table window.

2.6 Automatization



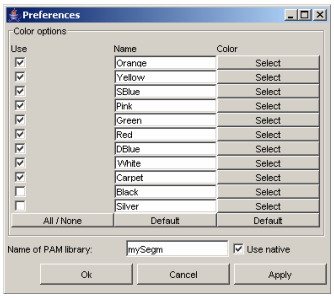
The automatization window lets you specify whether to run the automatization process with the current color table or one loaded from disk. Clicking “Configure” lets you specify how many color values to select during the automatization and if you want to use local histograms (e.g., one histogram for each image and color values are selected from each) or global histograms (e.g., one histogram for the entire set of images to select color values from) or both. It also lets you specify how many color values of each color you want to add to you color table. Note that a low number of values are recommended if using local histograms. With the global histograms a larger number is probably better.



A good way to automatize the calibration is to run the automatization first, getting a low number of color values from each image and then run it again, getting a high number of color values from all images. This will generate color values for most objects.

Experiment to find values that work for your type of playing field and light conditions.

The automatization process needs a large set of images to be effective.



2.7 Preferences

The preferences window lets you set your colors and their names. It also lets you check/uncheck colors. Unchecking a color makes ColorCalibration ignore the color for all processes, e.g., segmentation and automatization. However, you are still able to select pixels from an image with an unchecked color; it just won't show in the segmentation.

Preferences also lets you specify a system library that contains segmentation routines, if you want to use your own segmentation routines. Just specify the name, and make sure the systems library path environment variable is set to where the segmentation library is located and check the option "Use native" to use your own segmentation routines. You can read more about native segmentation routines in Chapter 4, *Using native segmentation code*.

3 Working with files

There are a few file formats you need to be familiar with, to get the most out of ColorCalibration. Here is a short description of the file formats.

3.1 PGM file format

The PGM file format is the format of the tutorial Open-R program *Image Capture*. It is divided in three files; the Y-, U- and V component. No compression

3.2 KJI file format

The KJI file format is, unlike PGM, a one file per image format. It is a YUV format and bears a close resemblance with the PGM format. No compression.

3.3 Color Table file format

The color tables are saved as C++ source code, giving you the ability to just include one when you compile your Open-R source code. Example:

```
#define TableSize 32

const unsigned char red_generic[TableSize*4]= {
      127, 127, 127, 127,
      …}
```

3.4 Converting PGM files to KJI files

To convert image files created for example by the Open-R tutorial program *Image Capture*, a batch conversion tool is included in the ColorCalibration distribution. It is a stand-alone application that can be run either as a command line application or with a graphical interface. It is called *FileConverter*. To use FileConverter in command line mode, just run it like a normal java program with your Y images as parameters. Example:

```
java FileConverter Yimg00.pgm Yimg01.pgm
```

To use FileConverter in a graphical environment, just run it without any parameters and a file selection dialog is displayed, letting you specify files to convert.

The files converted are saved in the same directory as the PGM files.

4 Using native segmentation code

ColorCalibration gives you the ability to run your own segmentation routines, written, for example, in C or C++.

4.1 Why use native code?

Why use your own code when ColorCalibration provides built-in, robust segmentation routines? The reason is simple. Calibration is done in order to make the segmentation optimal. If you are not using the same segmentation on your Aibo robot then you can not be sure the color table made with ColorCalibration is the optimal one. This does not mean that using the built-in segmentation is a bad idea. It is based on the latest segmentation routines from Team Sweden used successfully in RoboCup. But for serious Aibo applications we strongly recommend using your own routines.

4.2 Compiling the code to work with ColorCalibration

In order to use your own segmentation routines in ColorCalibration, you first need to compile your segmentation code into a system library with a small ColorCalibration wrapper included. A wrapper working on Team Sweden's segmentation routines is included in the ColorCalibration distribution. Modify it to work with your segmentation code and compile it. Example for linux:

```
g++ -I/usr/java/jdk1.4/include -I/usr/java/jdk1.4/include/linux -fPIC Segm.cc –c
g++ -shared -Wl,-soname,libMySegm.so -o libMySegm.so Segm.o
```

Where *Segm.cc* is your segmentation routine with the ColorCalibration wrapper included:

```
#include "AiboImage.cc"
```

The wrapper contains three methods. You may not change the input or return parameters of these methods. But you may freely alter the methods to fit your segmentation code.

```
/* set the image from Y-, U- and V-arrays */
jint Java_AiboImage_setImage(JNIEnv*, jobject, jintArray, jintArray, jintArray);

/* set the threshold from an array of thresholds */
jint Java_AiboImage_setThreshold(JNIEnv*, jobject, jintArray);

/* get the segmentation from array with colors indexes matching color table */
jintArray Java_AiboImage_getSegmentation(JNIEnv*, jobject, jintArray);
```