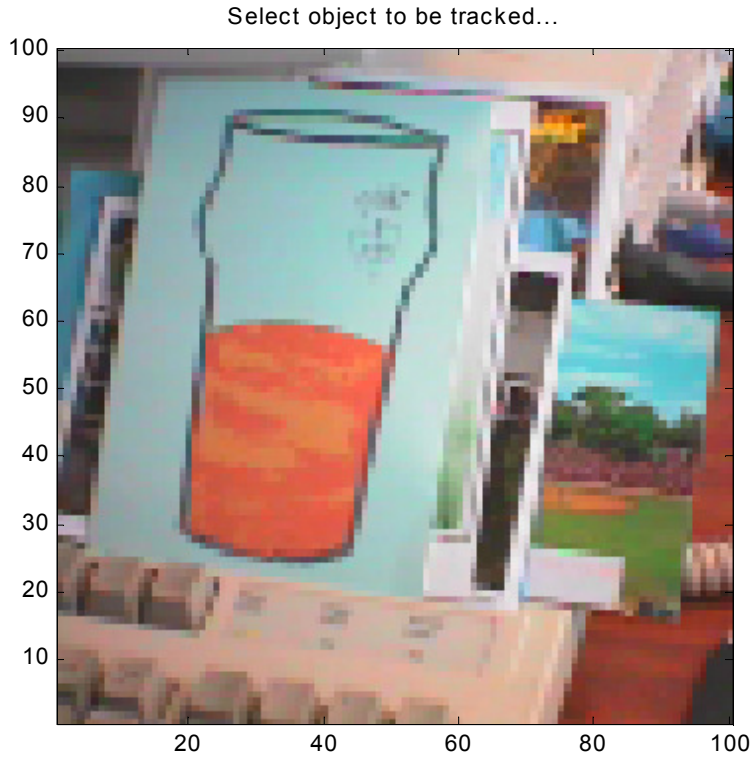
TIS Vision Tools

A simple MATLAB interface to the ”The Imaging Source (TIS)” FireWire cameras (DFK 31F03)



F. Wörnle, Aprit 2005

Contents

1. Introduction

This document describes a small collection of drivers which have been written to provide direct access from within MATLAB to FireWire cameras. The commands of the toolbox make use of MATLAB's standard C-Mex interface and/or Simulink S-Functions. Live images can thus be processed directly from the MATLAB command prompt as well as from within Simulink block diagrams.

A few of the provided drivers include rudimentary image processing capabilities. It is possible to classify image contents according to a programmable colour range and/or object size criteria. Furthermore, the algorithm readily returns the coordinates of the centroid of any detected object. This feature is particularly useful in machine vision based robotics applications including the automatic tracking of coloured objects (e. g. RoboCup, etc.).

The drivers of this toolbox have been written for and work best with "The Imaging Source" (TIS) cameras (www.theimagingsource.com). Access to the low-level Windows Driver Model (WDM) stream class drivers is commonly provided through the DirectX/DirectShow framework. The Imaging Source cameras come with a convenient Software Development Kit (SDK) which reduces the complexity of DirectX/DirectShow. The standard version of this SDK (IC Imaging Control) works with any TIS product (camera, frame grabber card, etc.). To use the toolbox with other WDM stream class compatible cameras (FireWire, USB, etc.), the professional version of the SDK needs to be purchased and the drivers should be re-built. At the writing of this document (April 2005) a single runtime license for *IC Imaging Control Professional* cost around $70 (US).

The classification algorithms are those of the *Color Machine Vision project* (CMVision), a project conducted by the CORAL group at the Carnegie Mellon School of Computer Science. This software has been published under the GNU General Public License (GPL) and can be obtained from the corresponding project page at http://www-2.cs.cmu.edu/~jbruce/cmvision/. CMVision provides "a simple, robust vision system suitable for real time robotics applications" by means of "global low level colour vision at video rates without the use of special purpose hardware".

To install and use the *TIS Vision Tools* toolbox, extract the contents of the zip to a local folder. The _bin folder of the distribution should be added to the MATLAB path variable as well as to the Windows path variable. On Windows XT, the latter can be done by right-hand clicking on *My Computer* and choosing *Properties*. Select the *Advanced* tab and click on *Environment Variables*. The path can now be added to the PATH variable.

The *TIS Vision Tools* are distributed as 'Free Software' under the terms of the GNU General Public License Agreement. Users are therefore given the right to copy, re-distribute and/or modify the source code to suit their needs.

Comments and bug reports are always welcome. Please direct your feedback to the following address:

Frank Wornle (frank.wornle@adelaide.edu.au)
The University of Adelaide
School of Mechanical Engineering

11 April 2005

2. The TIS Vision Tools interface

2.1 Supported video format

The Imaging Source FireWire cameras can be configured to produce digitized image information in a multitude of formats. Two popular video standards are currently supported by Vision Tools:

- RGB24 Each pixel is represented by a colour triplet of three consecutive bytes: **R**ed, **G**reen, **B**lue.

- YUV Each pixel is represented by a luminance byte (Y), and a corresponding chrominance byte (C_R/C_B); altering red chrominance bytes (U) and blue chrominance bytes (V) are transmitted in between any two luminance bytes. A pair of adjacent pixels are assigned the same colour, thus reducing the bandwidth of the image data. A set of 4 transmitted bytes therefore contains the entire information required to display 2 pixels. Notice that this is only 2/3 of the bandwidth required for RGB24. The reduced bandwidth might be of advantage when an application needs real-time image processing.

The camera used during the development of this toolbox is a DFK 31F03. This camera can be set up to work with a resolution of 640 x 320 or 1024 x 768. The maximum frame rate is 15 fps.

2.2 Image capture commands

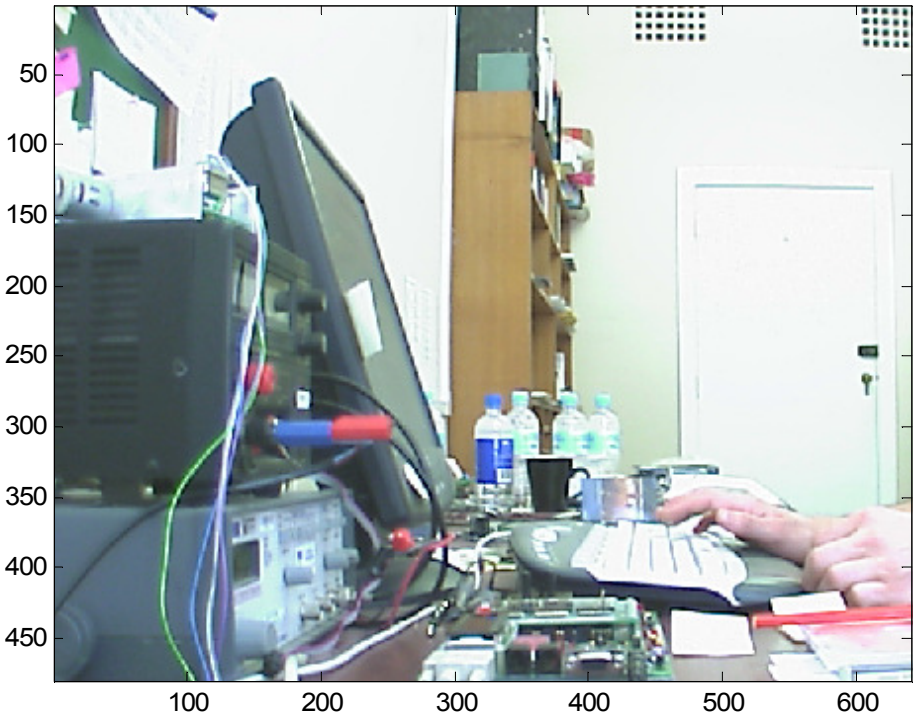
The following list describes the commands which can be used from the MATLAB command line to acquire visual information and return it to the workspace in form of a cell array.

2.2.1 capImage(0)

This command acquires a single frame of the camera. The call-up parameter defines the *video format* to be used (0 = RGB); the command returns an *n x n x 3* cell array containing the corresponding Red, Green and Blue values.

The following line can be used to display an RGB image:

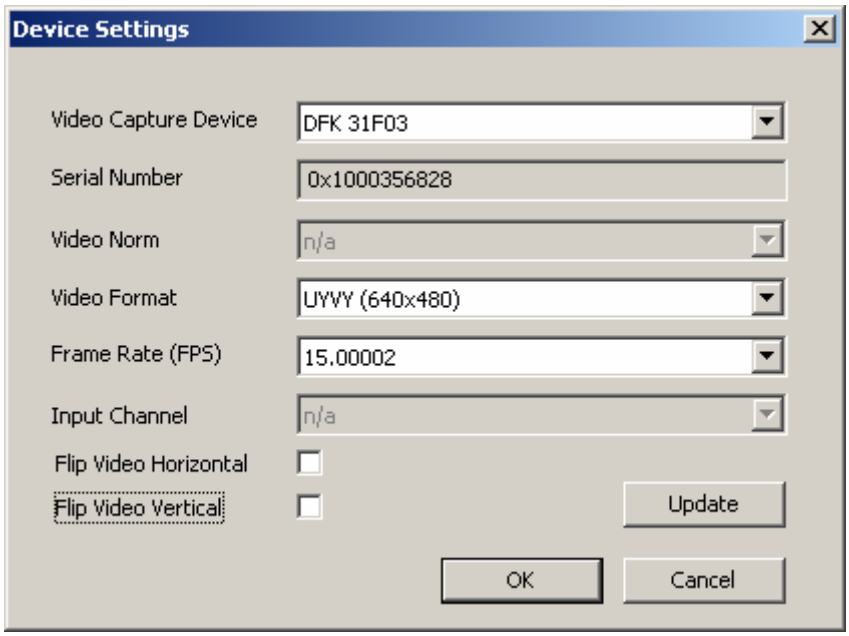
```
>> image(capImage(0));
```



640 x 480 pixel RGB image

A small requester is presented during the initialization of the camera. Upon selection of a camera (here: The Imaging Source DFK 31F03), a number of parameters can be configured. These include *image size* (640 x 480 or 1024 x

768) and frame rate (up to 15 fps on the DFK 31F03). In addition to this, the image can be flipped horizontally and/or vertically.

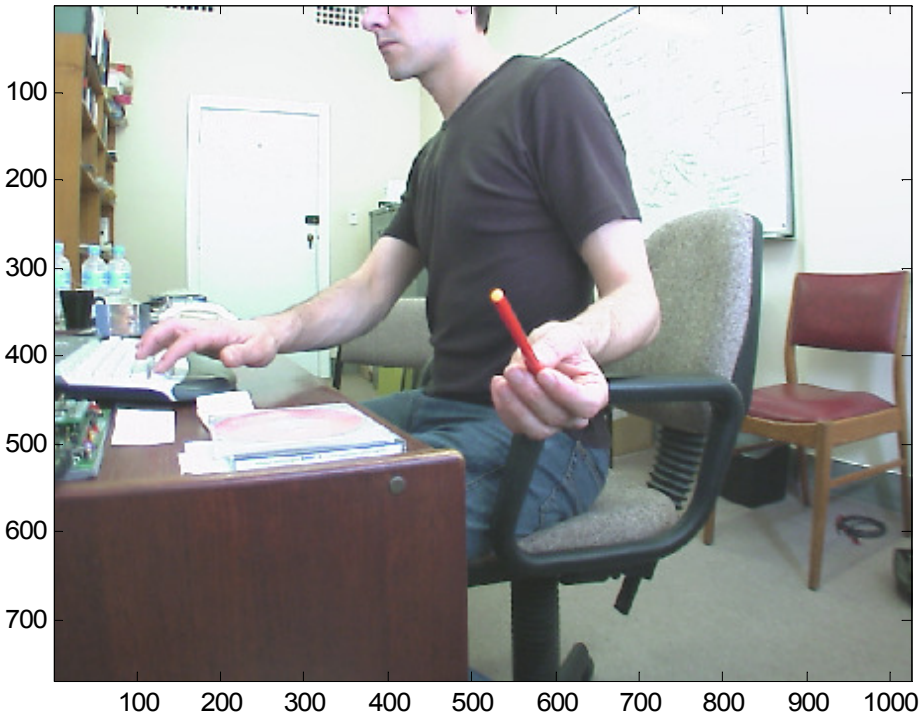


Choosing camera, image size and frame rate

The following lines cause MATLAB to continuously acquire RGB images and display them on screen (the program stops after 100 frames). Upon completion of 100 cycles, the grabber is switched off by calling the driver with call-up parameter '-1'.

```
% start grabber
for(i=1:100)
   image(capImage(0));
   drawnow
end

% stop grabber
capImage(-1);
```



RGB image – continuous acquisition (1024 x 768 pixels)

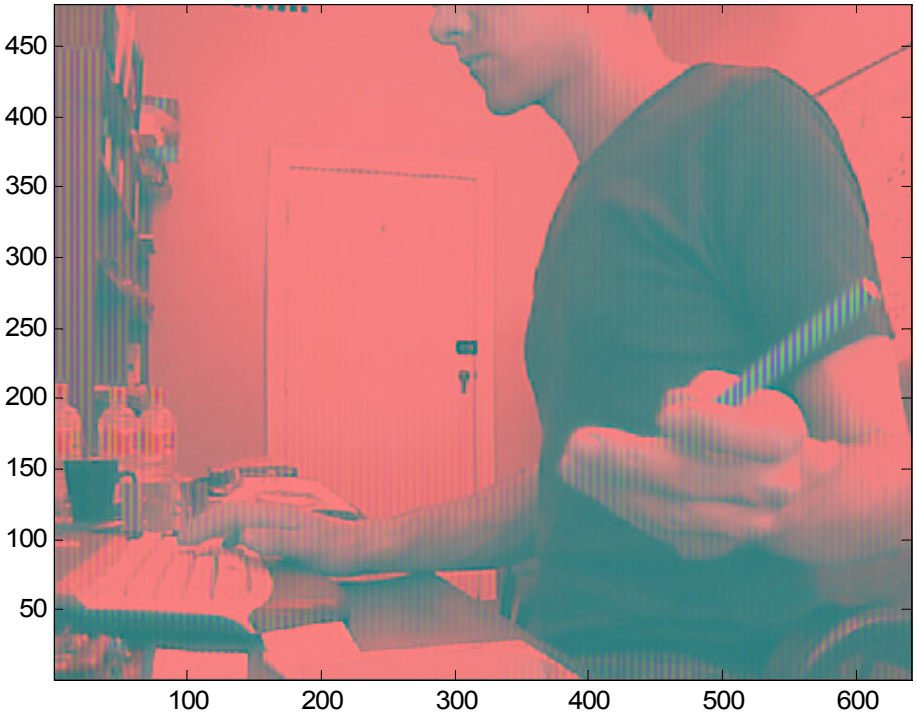
2.2.3 capImage(-1)

A call to *capImage* with call-up parameter '-1' stops the data acquisition and frees the associated grabber object. The next call to capImage will re-initialize the device. This has to be done when switching from low resolution mode (640 x 480) to high resolution mode (1024 x 768).

2.2.4 capImage(1)

Captures YUV images and returns them to MATLAB. The transmission of a YUV image requires less bandwidth than that of a regular RGB image (YUV only uses 2 bytes to represent each pixel). For more details about YUV see www.fourcc.org (YUV4:2:2 or UYVY). The function returns an n x n x 3 cell array: The first entry is the luminance information, the second entry contains the *blue* chrominance and the last entry is the *red* chrominance (Y x U x V).

YUV image data can be converted to RGB using the m-file script *yuv2rgb* (cf. appendix A).



Picture of the tip of a pen - YUV standard

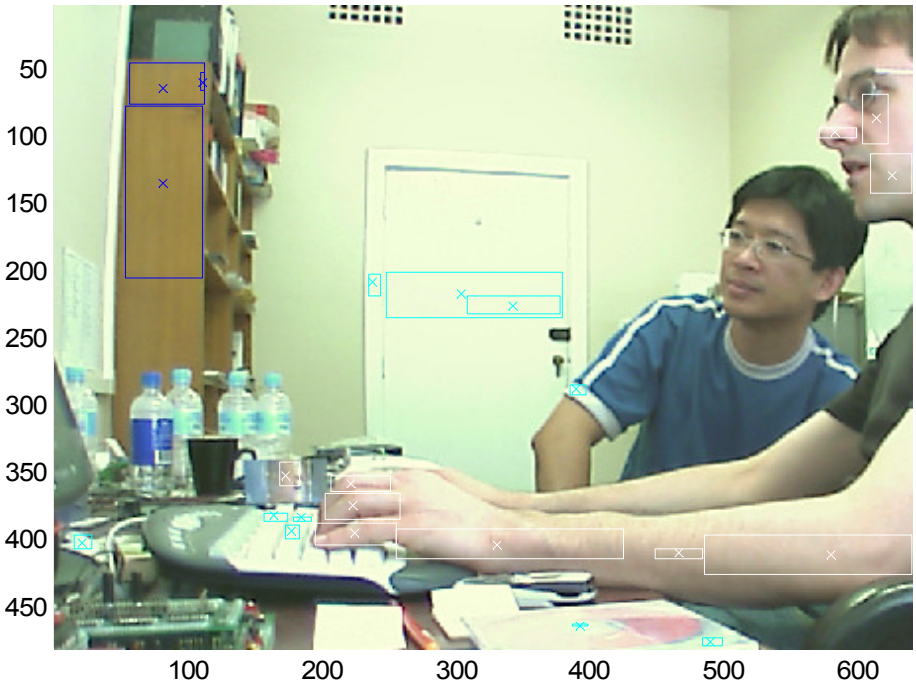
The same picture after conversion to RGB

2.3 Image processing commands

The commands in this section combine image acquisition with the data classification algorithms of the CMVision project (see: The CORAL group, http://www-2.cs.cmu.edu/~jbruce/cmvision/).

2.3.1 capProc()

This command can be used to detect and track objects. A number of CMVision algorithms are used to classify the acquired image data. *Clusters* with the requested size and/or colour qualities are being detected and made available to MATLAB in form of a cluster information structure. This structure include among other things the position and size of a rectangular box around the cluster, as well as the coordinates of its centroid. The location of the centroid can directly be used in tracking applications.



Detecting 3 different colours - Cluster *boundary box* and cluster *centroid*

The above example demonstrates the situation for a relatively selective colour range (only a few shades of each detected colour are accepted). A boundary box has been plotted to reveal the size of the detected cluster. The small star denotes the location of the centroid.

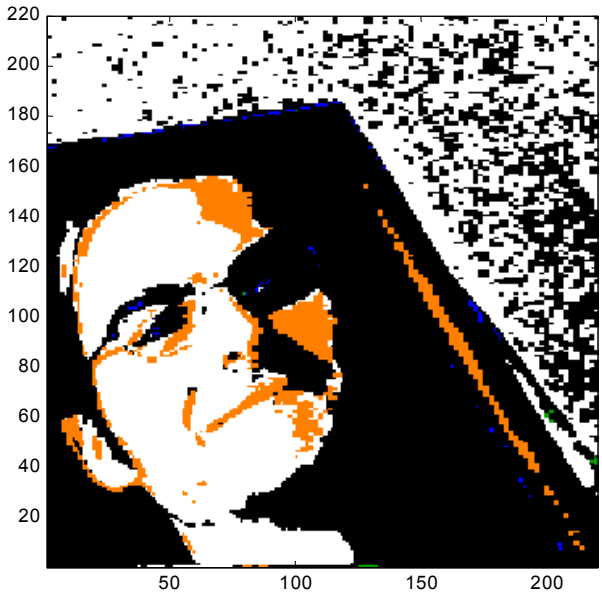
To learn about the admissible colour ranges, the command reads the file ‘color.txt’. This text file contains colour attributes for up to 32 different objects to be detected. The general structure of ‘color.txt’ is shown below:

```
[Colors]
(255,128,  0) 0.5000 2 Ball
(255,255,  0) 0.6000 3 Yellow_Team
(  0,  0,255) 0.6000 3 Blue_Team
(255,255,255) 0.0000 0 White
(255,  0,255) 0.0000 0 Marker_Pink
(160,  0,160) 0.0000 0 Marker_Purple
(  0,160,  0) 0.0000 0 Marker_Green
(  0,  0,  0) 0.0000 0 C08
(  0,  0,  0) 0.0000 0 C09
(  0,  0,  0) 0.0000 0 C10
(  0,  0,  0) 0.0000 0 C11
(  0,  0,  0) 0.0000 0 C12
(  0,  0,  0) 0.0000 0 C13
(  0,  0,  0) 0.0000 0 C14
(  0,  0,  0) 0.0000 0 C15
(  0,  0,  0) 0.0000 0 C16
(255,128,  0) 0.0000 0 Ball_2
(255,255,  0) 0.0000 0 Yellow_Team_2
(  0,  0,255) 0.0000 0 Blue_Team_2
(255,255,255) 0.0000 0 White_2
(255,  0,255) 0.0000 0 Marker_Pink_2
(160,  0,160) 0.0000 0 Marker_Purple_2
(  0,160,  0) 0.0000 0 Marker_Green_2


[Thresholds]
(  7:175, 50:150,160:200)
( 47:120,  5:80, 130:200)
( 76:112,110:190, 67:128)
(130:255, 81:131,125:178)
( 50:181,102:135,190:222)
(103: 96,118:140,144:166)
( 67:134, 96:129, 85:124)
(  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:  0)
(  0:  0,  0:  0,  0:  0)
(  0:  0,  0:  0,  0:  0)
( 86:221, 35: 79,102:150)
(  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)
```

The section **[Colors]** includes four columns:

(1) A user defined RGB colour triplet which can be used to visualize the bits of a detected cluster. This can be useful to validate the settings of a particular colour detection range. An example with four different detected regions is shown below.



Setting all pixels of a detected cluster to its default colour

(2) The *merge density parameter* is assigned a value between 0 and 1. It defines a colour density threshold beyond which two individual clusters are being merged into one bigger cluster. This parameter therefore allows control of the granularity of detected clusters within an acquired image.

(3) Colour ID; this parameter can be used to identify clusters of a particular colour quality.

(4) Colour name; this optional name can be used to link a detected object to a clear text identifier.

The section **[Thresholds]** contains the RGB colour ranges which are used to detect a particular cluster.

For further details about the CMVision algorithms, please refer to the CMVision manuals (http://www-2.cs.cmu.edu/~jbruce/cmvision/).

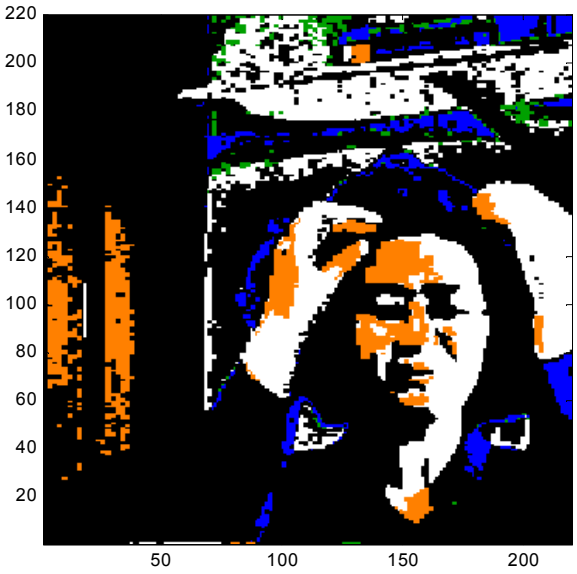
capProc can be invoked with one or two call-up parameters and one or two return parameters:

The first call-up parameter should always be set to 1 to make the camera acquire YUV images (required by the CMVision algorithms). The optional 2nd parameter is a vector of colour IDs to be detected. When omitted, the command only detects objects defined by the first entry in 'color.txt'.

capProc always returns the result of the image processing step (clusters, centroids, colours, names, ...). Additionally, the corresponding image data can be returned in an optional 2nd output parameter (RGB). An example of how to use capProc() can be found in appendix C.

2.3.2 capClassify()

The command capClassify can be used to find appropriate settings for a particular parameter in 'color.txt'. All pixels of a detected cluster are set to their default colour (1st column in section [Colors]).



Finding the correct settings of colour range and density threshold

2.3.3 imgProc()

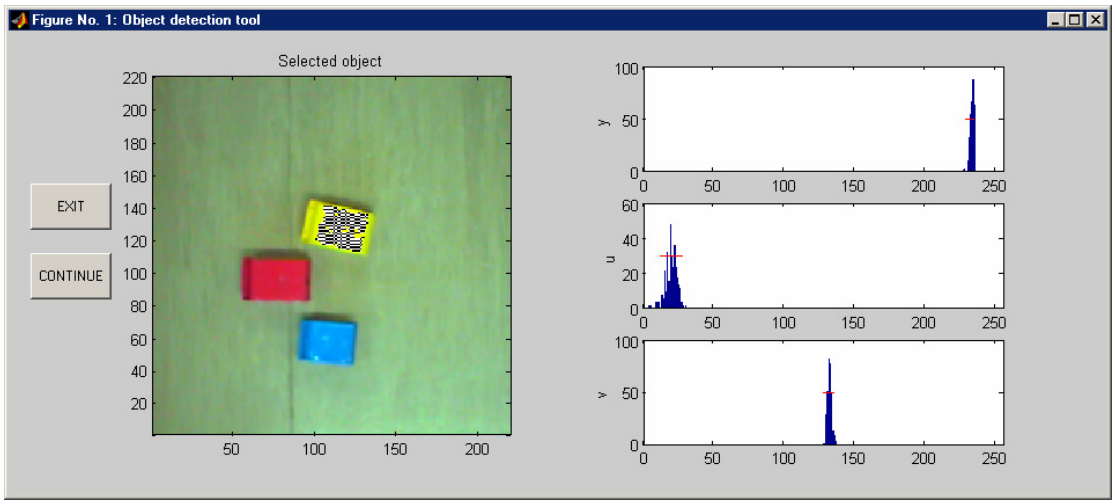
The command *imgProc* can be used to test the validity of a chosen colour definition. This command takes a previously stored RGB image as input as well as an optional vector of colour IDs. At present, *imgProc* expects the colour definition to be stored in a file called 'testcolors.txt'. The command returns a list of all detected regions together with their size, co-ordinates of the centroid, etc. See the test program in folder /imgProc for further details.

2.4 Selecting colour ranges

Object tracking by colour detection relies on the correct definition of appropriate RGB colour ranges. Too wide a range will lead to an unwanted detection of objects with similar colour qualities; on the other hand, too narrow ranges might cause the algorithm to return without having detected anything.

To determine appropriate RGB ranges, a number of small MATLAB m-files have been provided. These 'training' programs allow a user to select an object to be tracked from either a still image (trainStill.m) or a live stream of images (trainCamera.m). Both programs return the colour characteristics of the selected object. The thus established values (ranges of Y, U, V) can directly be copied to a colour definition file.

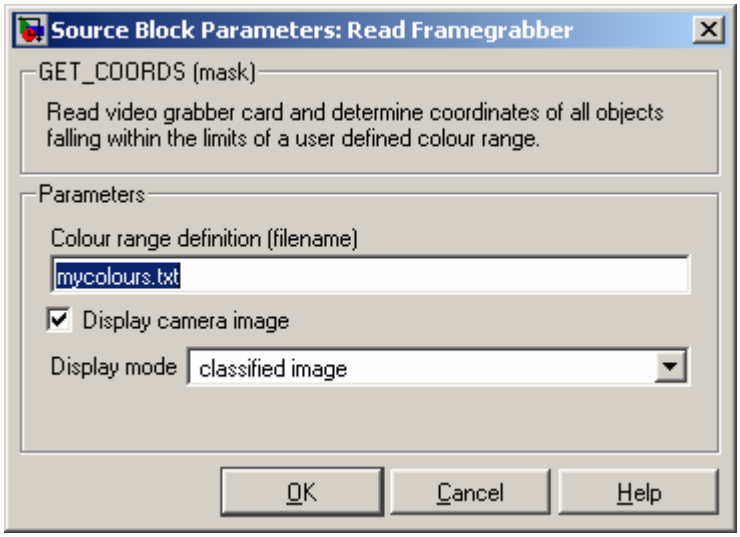
Additional information on the issue of colour training can be found in the document 'Train_UserManual.doc' which has been included in the present distribution of the toolbox.



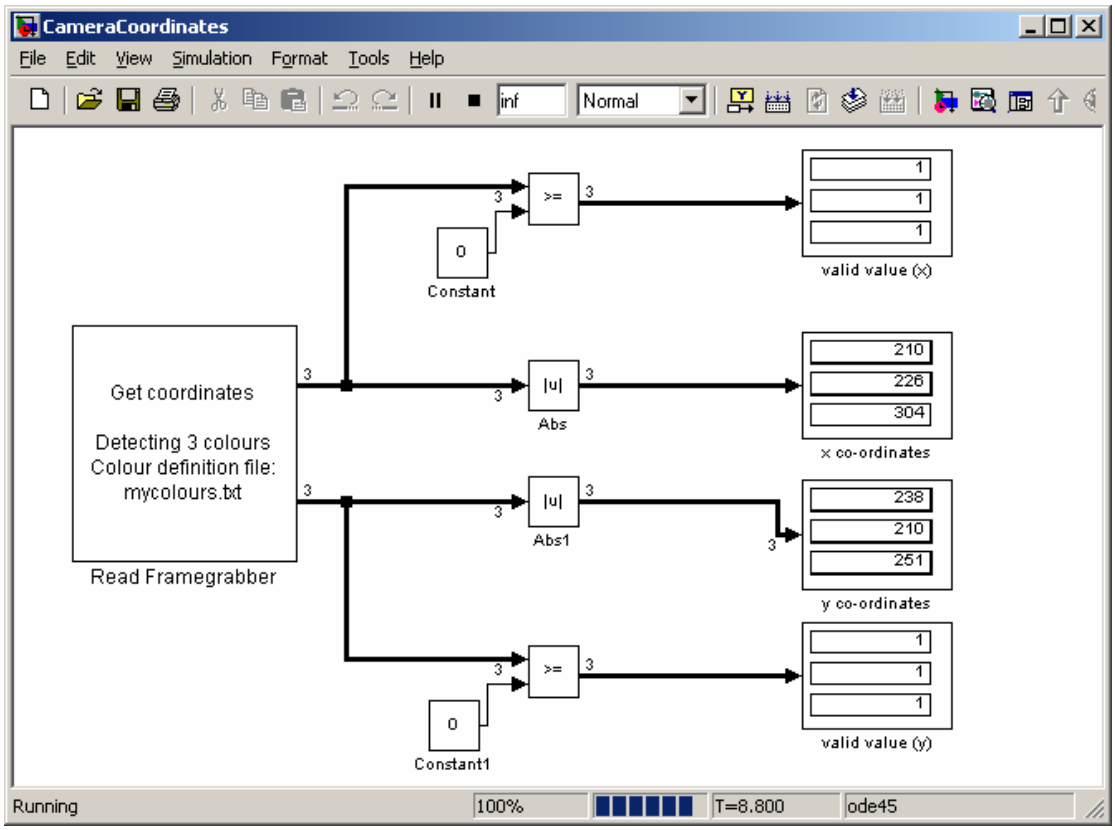
Training the camera to 'see' yellow…

2.5 Simulink interface

Object tracking can also be done in Simulink. A suitable SFunction block has been designed and can be found in folder /SFcapProc. This block analyses the current image of the camera and returns the coordinates (x, y) of the first detected object of each specified colour. The block mask allows specification of the colour definition file. The (optional) display of image information can be enabled using the 'display' check box.

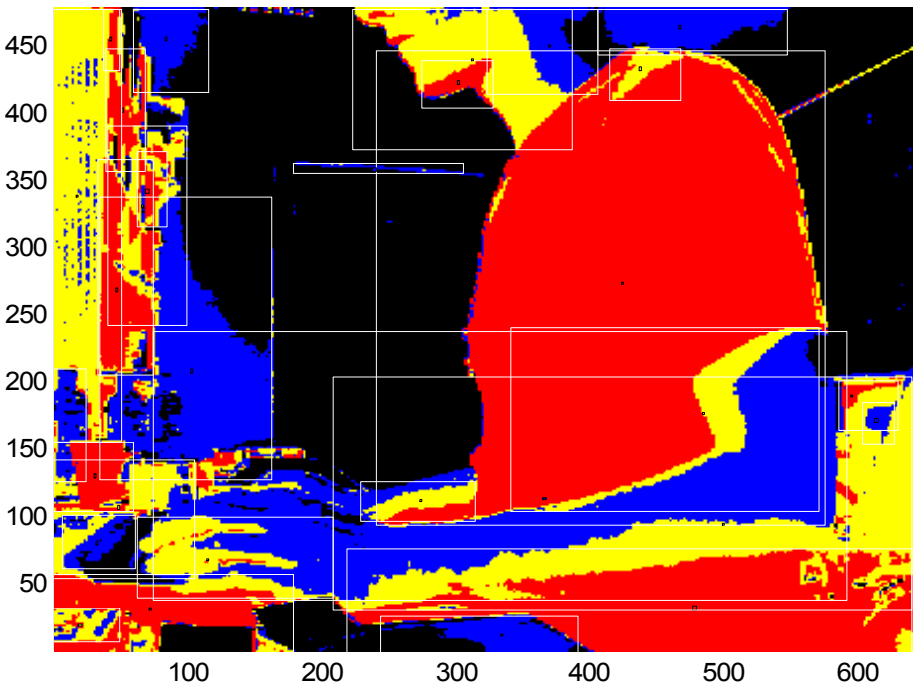


Block parameters: colour definition file and (optional) display mode

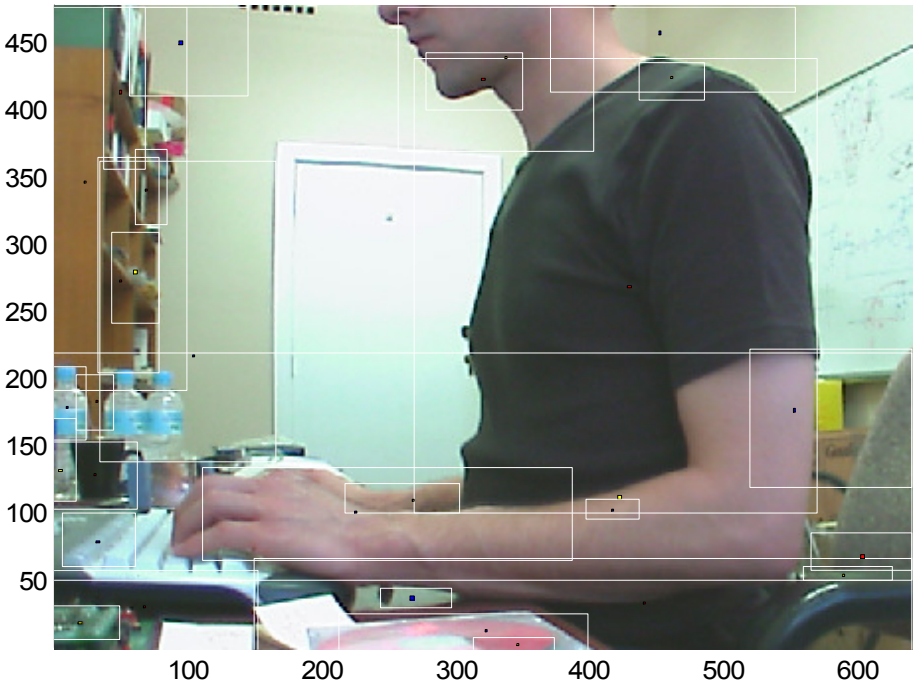


A Simulink based vision processing system

The display of images can be switched to ‘classify’ mode. This reveals which objects the camera currently associates with the provided colour definitions. Each detected object is coloured in the associated display colour (see colour definition file, section 2.3.1). A boundary box is drawn around each selected object and the centroid is displayed.



Classify mode: Detected objects are coloured, undetected objects are black



Regular display mode: Detected objects have centroids and boundary boxes

Appendix

Appendix A – The MATLAB script *yuv2rgb*

```
% converts rgb data to yuv  (FW-04-03)

function dst = rgb2yuv(src)

% ensure this runs with rgb images as well as rgb triples
if(length(size(src)) > 2)

    % rgb image ([r] [g] [b])
    r = double(src(:,:,1));
    g = double(src(:,:,2));
    b = double(src(:,:,3));

elseif(length(src) == 3)

    % rgb triplet ([r, g, b])
    r = double(src(1));
    g = double(src(2));
    b = double(src(3));

else

    % unknown input format
    error('rgb2yuv: unknown input format');

end

% convert...
y = floor(0.3*r + 0.5881*g + 0.1118*b);
u = floor(-0.15*r - 0.2941*g + 0.3882*b + 128);
v = floor(0.35*r - 0.2941*g - 0.0559*b + 128);

% ensure valid range for uint8 values
y(y > 255) = 255;
y(y < 0)   = 0;
u(u > 255) = 255;
u(u < 0)   = 0;
v(v > 255) = 255;
v(v < 0)   = 0;


% generate output
if(length(size(src)) > 2)

    % yuv image ([y] [u] [v])
    dst(:,:,1) = uint8(y);
    dst(:,:,2) = uint8(u);
    dst(:,:,3) = uint8(v);

else

    % yuv triplet ([y, u, v])
    dst = uint8([y, u, v]);

end
```

Appendix B – The MATLAB script *yuyv2rgb*

```
% converts yuv data to rgb  (FW-04-03)

function dst = yuv2rgb(src)

% ensure this runs with yuv images as well as yuv triples
if(length(size(src)) > 2)

    % yuv image ([y] [u] [v])
    y = double(src(:,:,1));
    u = double(src(:,:,2));
    v = double(src(:,:,3));

elseif(length(src) == 3)

    % yuv triplet ([y, u, v])
    y = double(src(1));
    u = double(src(2));
    v = double(src(3));

else

    % unknown input format
    error('yuv2rgb: unknown input format');

end


% convert...
u = 2*u - 256;
v = 2*v - 256;

r = y + v;
g = floor(y - 0.51*v - 0.19*u);
b = y + u;

% ensure valid range for uint8 values
r(r > 255) = 255;
r(r < 0)   = 0;
g(g > 255) = 255;
g(g < 0)   = 0;
b(b > 255) = 255;
b(b < 0)   = 0;


% generate output
if(length(size(src)) > 2)

    % rgb image ([r] [g] [b])
    dst(:,:,1) = uint8(r);
    dst(:,:,2) = uint8(g);
    dst(:,:,3) = uint8(b);

else

    % rgb triplet ([r, g, b])
    dst = uint8([r, g, b]);

end
```

Appendix C – A test program for *capProcs*

```
% test program to continuously capture pictures from the camera

function    test(varargin)

% run variable is global -> stop button callback can reach this
global run;

run = 1;

% check inputs of function 'test'
if(isempty(varargin))
    scan4col = [];
elseif(nargin == 1) % one input argument -> colour vector
    scan4col = varargin{1};
    if(scan4col == -1)
        % just stopping grabber...
        kk = capProc(-1);
        run = 0;
    end
end

if(run)

    % open figure window
    figure
    title('Click STOP to stop data acquisition...');

    % push button to stop data acquisition...
    stop_h = uicontrol('Style', 'pushbutton', 'String', 'STOP',...
        'Position', [20 200 50 50], 'Callback', 'zeroGlobalRun');

    % endless loop...
    while(run)
        [out, R] = capProc(1, scan4col);
        image(R);
        axis image xy;

        if(~isempty(out))               % no regions detected
            hold on;

            nCol = length(out);     % number of colours with valid regions
            for(j = 1:nCol)
                kk = out(j).nRegions;
                col = out(j).Colour;
                while(kk)
                    cx = out(j).Regions(1,kk);      % centroid
                    cy = out(j).Regions(2,kk);
                    x  = out(j).Regions(3,kk);      % boundary box
                    y  = out(j).Regions(4,kk);
                    w  = out(j).Regions(5,kk);
                    h  = out(j).Regions(6,kk);
                    patch([x x+w x+w x],[y y y+h y+h],col);
                    plot(cx,cy,'w*');
                    kk = kk - 1;
                end
            end

            hold off
        end

        drawnow
    end
```

```
    % delete 'freeze' button
    delete(stop_h);

    % stop grabber (recursive calle)
    test(-1);

end
```