Richard Cubek

Projektarbeit - SS 2010

A Color Blob Based Robot Vision

Written Report

Contents

1 Motivation

Our robot demonstrator *Kate* mainly consists of a *Pioneer* platform, a *Katana* robot arm actuator and a stereo-camera vision system, all connected by an industry PC. Since the stereo-camera vision is not usable yet, a temporary solution is needed, that fits the requirements of a robot vision for our Learning from Demonstration framework. The requirement is to recognize a couple of different objects on a table, and the delivering of their exact 3d-position relatively to the Katana system. The table can be assumed to be always the same. The objects should be recognized by their color only. The camera should be installed somewhere on the robot, what implies a tilted camera view direction. Since a tilted view causes some problems and due to shortage of time, a first prototype will be developed with the camera beeing installed on a simple construction, enabling a vertical camera view on the table.

2 Introduction

2.1 OpenCV

For Image Processing, we use OpenCV [2], the most popular Open Source Image Processing library, written in C. We start with some basics to make later appearing code snippets more understandable. The basic image handling type in OpenCV is `IplImage*`, most functions take this type as parameter input. The most basic OpenCV program is to open an existing image and to display it within the built in OpenCV basic GUI (please note, that all shown code snippets are not always from the original code but more simplified examples that show, how it is done in OpenCV):

```
IplImage* image;
image = cvLoadImage("World.jpg");

while(true)
{
    cvShowImage("world", image);
    // break on 'escape'
    if ((cvWaitKey(10)&0xff) == 27)
        break;
}
```

Of course, one has to include the header `cv.h` and `highgui.h`. Further, in the same program, we could create empty image space (allocate image memory) for further image processing purpose. Hereby, one has to define the bit depth per pixel and the number of channels per pixel:

```
IplImage* colored    = cvCreateImage(cvGetSize(image), 8, 3);
IplImage* grayscaled = cvCreateImage(cvGetSize(image), 8, 1);
```

To avoid memory leaks, it is important to release the image with `cvReleaseImage(&image)` when the image is no longer needed. This should be enough to understand following code snippets. We will not introduce deeper into OpenCV here, very much example code is available on the net, the official documentation can be found at `http://opencv.willowgarage.com/documentation`.

2.2 Camera Access

Usually, it is very simple to access the camera and fetch frames with OpenCV, the basic method is:

```
CvCapture* capture = cvCaptureFromCAM(0);
IplImage* frame = cvQueryFrame(capture);
```

On some linux systems, problems can occur using this direct OpenCV access. Unfortenately, since our systems are affected, too, we need a more low-level access. Most firewire cameras conform the digital camera 1394 specification, therefore, the camera can be accessed using the libdc1394. This library provides an API to control IEEE 1394 (firewire) based cameras. *Unicap* is a more high-level abstraction for libdc1394, but even this lib didn't work without problems. Further research on the internet shows, that these problems can occur on some systems. Finally, an object-oriented approach encapsulating the libdc1394 is used, developed by our project partner from the Mannheim university. The framework consists of a single class `dsCameraDC1394`. The method provides, among others, the method `getRGB24Image`, which returns the pointer to the image data in memory or `NULL`, if the data is still written. Thus, we can access the data and convert it to our OpenCV type `IplImage*` simply with:

```
dsCameraDC1394* cam;
unsigned char* frameArray;
IplImage* camFrame;
...

while (frameArray == NULL)
{
    frameArray = cam->getRGB24Image();
}

// set OpenCV image data from frame
camFrame->imageData = (char*) frameArray;
```

Using libdc1394 (indirectly), we can also configure various camera configuration settings from the program, for example the important values manual or auto configuration, shutter, gain and white balance.

2.3 Correction of Lense Distortion

Most camera lenses are distorting the image, straight edges then appear roundish. Before doing the image processing, the image should be undistorted. OpenCV provides a camera calibration (image undistortion) example program, which as input needs some images with long edges. The best solution are images of a chessboard pattern. We use that program to get the undistortion parameters for our camera. Figure 1 shows three of the 20 recorded calibration images, one can clearly see the lense distortion.

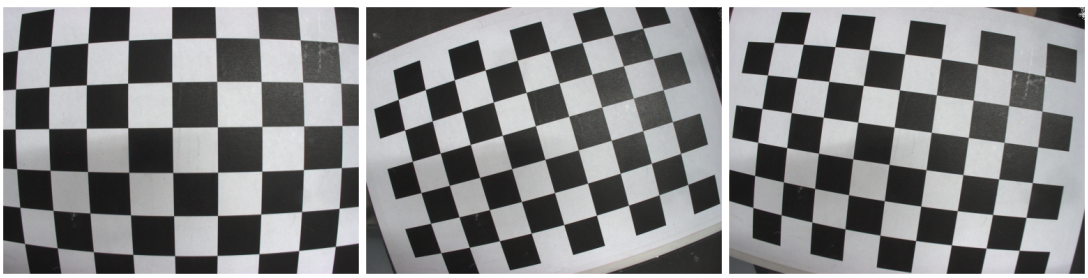
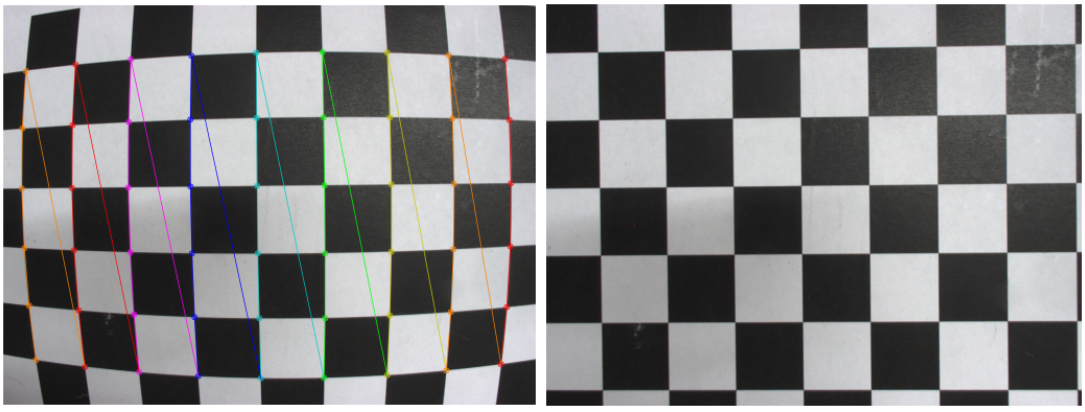


Figure 1: Three of the 20 calibration images.

After running the calibration program under `calibration/`, the undistortion parameters are stored under `calibration/distortion.xml` and `calibration/camera.xml` and can then be read out in the program to undistort the camera frames. This is done by using the OpenCV function `cvUndistort2`:

```
// matrices for undistortion
CvMat *cameraMatrix;
CvMat *distCoeffsMatrix;

// buffers for copy of cam frame and for undistorted frame
IplImage* frameBuffer;
IplImage* undistFrame;
...

// allocate memory
frameBuffer = cvCreateImage(cvGetSize(image), 8, 3);
undistFrame = cvCreateImage(cvGetSize(image), 8, 3);
...

// copy frame data to separate memory to avoiud overwriting effects
cvCopy(camFrame, bufferFrame);
...

// load matrices
cameraMatrix = (CvMat*) cvLoad(CAMERA_CALIB_XML);
distCoeffsMatrix = (CvMat*) cvLoad(CAMERA_DISTORT_XML);

// undistort frame buffer
cvUndistort2(frameBuffer, undistFrame, cameraMatrix, distCoeffsMatrix);
```



(a) (b)

Figure 2: The corners and edges of the first image from Figure 1 beeing detected by the calibration program (a) and the same image after undistortion (b).

On Figure 2(a) we can see one of the calibration images beeing processed by the calibration program, which marks detected corners and some edges. Using the parameters from this cali-

bration and undistorting this calibration image 2(b), we can clearly see that straight edges from the board now appear correctly on the image.

3 Prototype

3.1 Hardware Setup

As described in 1, for the prototype, we simply install the camera on a simple construction in a manner, that the camera view is directed vertically on the table scene (Figure 3). That makes two further steps much easier: the conversion from the camera to the Katana system and the blob error correction approximation.

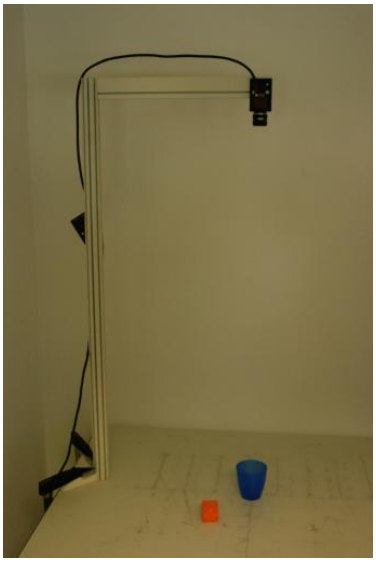


Figure 3: The simple camera construction of the prototype.

3.2 Workflow

The whole forkflow from reading a camera image to the 3d-position relatively to the Katana can be partitioned in the following steps:

1. Read camera frame
2. Undistort camera frame
3. Filter frame by a desired color to an black-and-white image
4. Detect white blobs on image
5. Correct blob coordinates based on error approximation
6. Convert from camera system to Katana system

The first two steps are described in 2.2 and 2.3, the remaining steps will be described in detail below.

3.2.1 Filtering by Colors

As described in 1, we recognize and identify objects by their color. Hence, we need to know which objects can be on the table, and we have to know their colors before. For the beginning, the known objects and colors are hardcoded. Later, this will be defined in a configuration file (4.2). Now, the first image processing step is to filter the desired color. Areas with the searched

color should then appear white in the black-and-white image, the remaining area should be black. Most often, a HSV (Hue-Saturation-Value) color definition is used for color filtering. The advantage of HSV is, that the hue defines the exact color in the visual color range. Further, we can define a minimum saturation for each pixel, that has our desired color. This is needed, because very often, on digital images, there are very dark areas, containing nearly any arbitrary color from the color range. This are areas, which we don't want to filter.

By default, OpenCV encodes images with an BGR format (Blue-Green-Red). Thus, we have to convert the format to an HSV format. First of all, we have to create the hsv image and a grayscale image for the filtering result:

```
IplImage *imghsv = cvCreateImage(cvGetSize(imgbgr), 8, 3);
IplImage *filtered = cvCreateImage(cvGetSize(imgbgr), 8, 1);

// convert
cvCvtColor(imgbgr, imagehsv, CV_BGR2HSV);
```

Since we have to process each pixel from the HSV image, we need two arrays, one for the hsv image data and one to set each pixel of the filtering result:

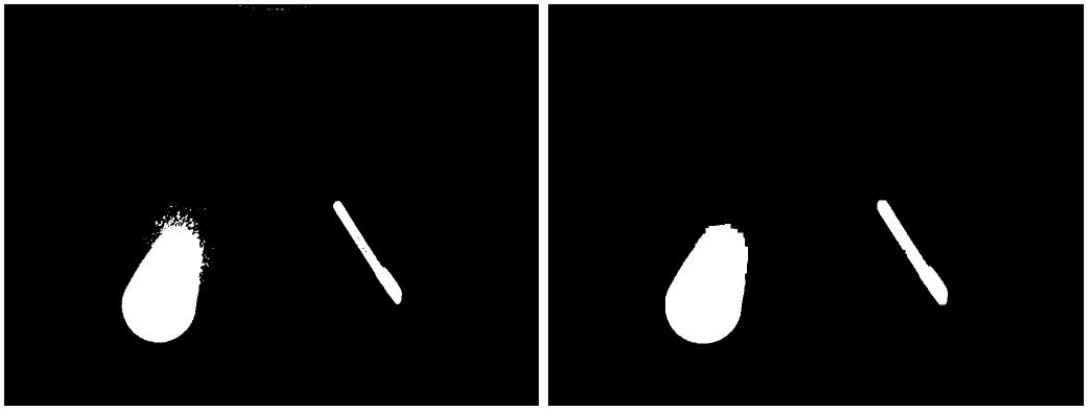
```
uchar *datahsv,*datafiltered;
datahsv = (uchar *)imagehsv->imageData;
datafiltered = (uchar *)filtered->imageData;
```

For each searched object, we know that minimal and the maximal hue value (color range). Iterating over all pixels in the HSV image, we now have to check, whether the pixel is in the desired color range and whether the saturation is bigger or equal the minimal saturation. If so, we set the corresponding pixel in the filtering result image to white. Thereby, we have to be careful, the range can go over the end/beginning of the visual color range (when `hue_min` is smaller than `hue_max`):

```
if (hue_max > hue_min)
    if ((datahsv[pix_index] >= hue_min) && (datahsv[pix_index] <= hue_max))
    inRange = true;
else
    if ((datahsv[pix_index] >= hue_min) || (datahsv[pix_index] <= hue_max))
        inRange = true;

if (inRange && datahsv[pix_min] >= sat_min)
    // set corresponding pixel to white...
```

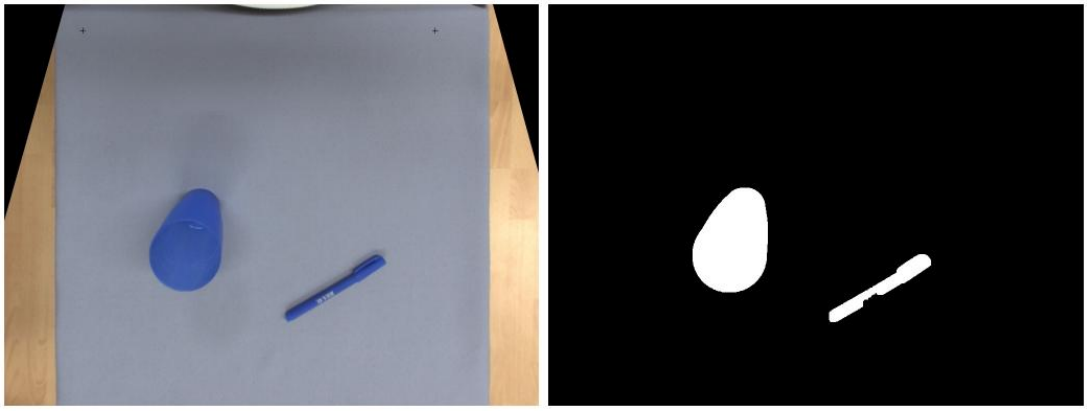
Note that this step has to be done for each object that possibly can occur on the table. Now we have a resulting black-and-white image. But this image is still not clean, it is noisy. We can use further processing steps to get cleaner blobs on the image and to remove noise. The function `cvErode` erodes an image using a specified structuring element (i.e. 3x3 pixel structuring element) that determines the shape of a pixel neighborhood over which a special kind of minimum is taken. The function `cvDilate` again dilates an image by using the specific structuring element. Both functions can be repeated a couple of times. Using these functions, we get a cleaner blobs image (Figure 4).

Figure 5 shows the result of this workflow part, having a frame with blue objects after all three steps, color filtering, color blob erodation and dilation.



(a) (b)

Figure 4: A filtered color image before (a) and after erodation/dilation (b). On (a), we can clearly see some noise at the top of the frame and at the cup border.



(a) (b)

Figure 5: The original image with blue objects on the table (a) and the result after color filtering, erodation and dilation (b).

3.2.2 Detecting Blobs with cvblob

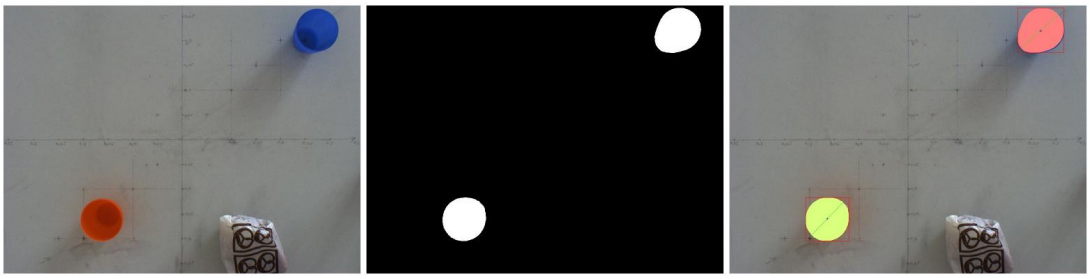
So far, we have filtered the scene by colors. Now, we want to detect the blobs on the black-and-white image. Therefore, several libraries exist, the most suitable one for our case was cvblob, an OpenCV library extension. Using cvblob, we can get different properties of the blobs, for example the blob center, the size in pixels, the min. and max. pixel position in x or y and so on. We can also draw the blob corners with the center on an arbitrary image, which is useful to show the detected blobs on the original (undistorted) camera frame. In our vision framework, we have a special method `getBlobsHSV(hue_min, hue_max, sat_min)` which filters the (undistorted) camera frame and finds all blobs within the desired color range, returning a vector containing instances of an own blob object struct:

```
typedef struct ColorBlob {
    int center_x;
    int center_y;
    int area;
    int min_x;
    int min_y;
```

```
    int max_x;
    int max_y;
} ColorBlob;
```

The blobs in the mentioned vector are ordered by size. The area (size in pixels) is needed to sort the blobs, the min. and max. coordinates are needed to place text at the blob and to calculate the midpoint. The center is similar to the center of gravity. When watching vertically on the scene as done with the prototype, we can use that center, but we will see later (4.3.2), that in the final version with a tilted view, we need the border coordinates(min. and max. values) for the calculation of the exact object position.

Since we identify each object by a special color, having more than one blob found when calling `getBlobsHSV`, we will take the first element of the returned vector always, choosing the biggest blob found for that color. In Figure 6, we can see the result of the blob detection process.



(a) (b) (c)

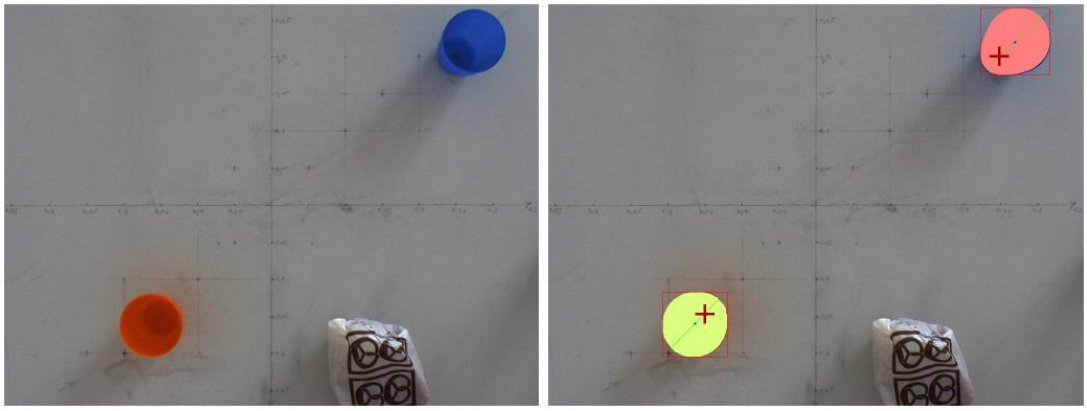
Figure 6: The original image (a), the detected color blobs (b) and the blobs from (b) inserted in the original frame (c).

3.2.3 Blob Coordinate Error Correction Approximation

Detecting an object on the table, we want its midpoint to be the objects position in x and y. Taking a look at Figure 7, we can see that due to the fact, that a cup is not two-dimensional, the real midpoint (red '+') of the cups and the blob midpoints are not at the same position. There is an error, which is only dependent from the blobs distance to the image midpoint. So, we can try to approximate a polynomial error function to determine the radial error for each blob. The parameters of the error function need to be determined for each kind of object. The radial error is the hypotenuse of an imaginary triangle with the errors in x and y as the remaining sides. Thus, from the radial error and the blobs position, we can easily calculate the error in x and y. Recording nine cup positions and the corresponding errors, we see, that the error function can easily be approximated with a polynom of first order (Figure 8).

3.2.4 Conversion from Camera to Katana System

In our setup, we have two different coordinate systems. The first one is that from the camera, and the second is the one of the Katana robot arm. We want to convert the coordinates of the camera system to that one from the Katana. We already know, where the objects are in the camera system, we can simply read it out from the corrected blob positions on the image, since we assume the camera viewing vertically on the table. That was one of the reasons to start with such a hardware setup. The only thing to do is to convert from pixel to mm, which is easy. The z position (depth) of an object within the camera system is a constant, since we have a fixed installed camera on the table and we only detect objects on the table. That



(a) (b)

Figure 7: A scene with two cups (a) and the detected blobs (b). The blob centers are different to the real cup positions (red '+').

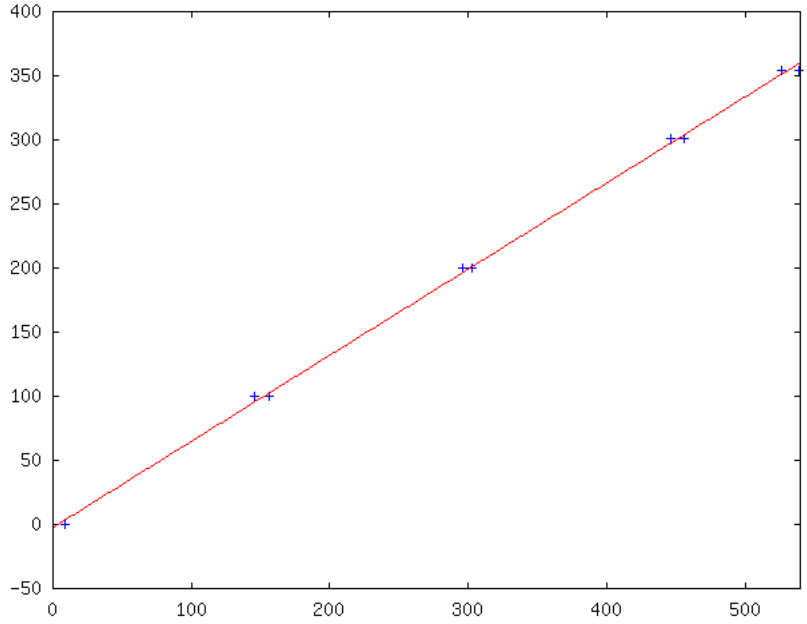


Figure 8: The approximation of the error function (x-axis: blob radial distance to image midpoint; y-axis: real radial distance to image midpoint).

is the natural limitation when using only one camera, but as mentioned in 1, it fullfills our needs.

To repeat, we want to convert from the camera coordinate system to that one of the Katana. In 3D vision, if one has two different coordinate systems, calculating the matrix to convert from a source vector position of the first system to a target vector position of the second system, one needs to move the target system into the source system by rotations and translations. Multiplying the rotation and translation matrices in the reversed order as the systems where moved, the resulting matrix $\mathcal{A}$ will be the conversion matrix from the source to the target system [4]. In our system, that means, that we have to move the Katana system imaginary into that one of the camera.

Thinking of standing in front of the table, the camera construction beeing installed on the right

side of the table, we now put the robot in front of the table (where we stand). Its x-axis will now be directed straight forward (in the direction of where the table is), its y-axis is directed to the left (parallel to the table). Its z-axis (upward) is now the reversed direction of the cameras z-axis. The x-axis of the camera is directed to the right, its y-axis towards us. First, we translate the Katana origin to the cameras origin, this can be done with the matrix:

$$\mathcal{T} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The t values are the distances of the Katana to the camera in each axis. Now, in two rotation steps, we can get to the goal. The first rotation is 180 degrees around the katanas x-axis. The z-axes of both systems then show into the same direction. Therefore, following x-rotation matrix is needed:

$$\mathcal{R}_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(rot_x) & -sin(rot_x) & 0 \\ 0 & sin(rot_x) & cos(rot_x) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The remaining rotation is 90 degrees around the Katanas z-axis:

$$\mathcal{R}_z = \begin{pmatrix} cos(rot_z) & -sin(rot_z) & 0 & 0 \\ sin(rot_z) & cos(rot_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The Katana and the camera now imaginary lie in the same coordinate system. The searched Conversion Matrix $\mathcal{A}$ can be calculated as follows:

$$\mathcal{A} = \mathcal{R}_x \cdot \mathcal{R}_z \cdot \mathcal{T}$$

And finally, converting a position $\vec{p}_{cam}$ from the camera to the position $\vec{p}_{kat}$ in the Katana system:

$$\vec{p}_{kat} = \mathcal{A} \cdot \vec{p}_{cam}$$

In OpenCV, one has to create matrices from arrays. The following code shows how to create the rotation matrices and how to multiply them.

```
double rotXArr[]  = { 1, 0,         0,          0,
                      0, cos(rotX), -sin(rotX), 0,
                      0, sin(rotX), cos(rotX),  0,
                      0, 0,         0,          1 };

double rotZArr[]  = { cos(rotZ), -sin(rotZ), 0, 0,
                      sin(rotZ), cos(rotZ),  0, 0,
                      0,         0,          1, 0,
                      0,         0,          0, 1 };

CvMat rotXMat = cvMat(4, 4, CV_64FC1, rotXArr);
```

```
CvMat rotZMat = cvMat(4, 4, CV_64FC1, rotZArr);
CvMat* result = cvCreateMat(4, 4, CV_64FC1);

cvMatMulAdd(&rotXMat, &rotZMat, NULL, result);
```

The same is done with vectors. In this way, we calculate the resulting position in the Katana coordinate system. It has to be repeated, even if this approach works fine, we have two naive assumptions here. The first is the vertical camera view on the scene, the second is, that the Katana stands right-angled in front of the camera. No assumptions will have to be done with the new system, introduced in 4. By the way, the assumption of the right-angled Katana position could also be avoided using the same solution as in 4.3.3.

3.3 Tests

Taking 20 arbitrary cup positions, the mean error is about 2 mm in each coordinate. The maximum error is 4 mm in the x-axis, occuring only on the left side of the table. The error is different in the different sections of the table, because we assumed the camera showing vertically on the scene, but that is difficult to acchieve with such a simple construction. However, the prototype absolutely fullfills our needs. The Learning from Demonstration example works fine with the new vision system. We now develop a version with the camera on the Katana.

4 Final System

4.1 Hardware Setup

As described in 1, we now want to install the camera on the robot. Trying different positions, the best one found is shown in Figure 9, with the camera beeing installed on the Katana, which again holds in a special position. The conversion from the camera to the Katana system will be different than with the prototype, and the blob error correction approximation will no longer depend from the radial distance of the blob center. Probably, we will need different error functions in x and y.

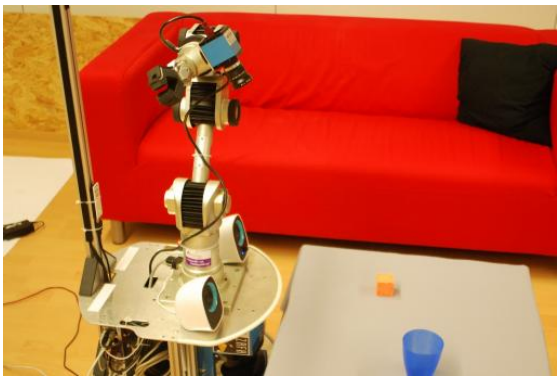


Figure 9: The robot with the camera on the Katana.

The used *Guppy* camera from the prototype would be very suitable to be installed on the Katana, but tests with the *ImagingSource* camera showed, that its auto configuration performs much better than that one from the Guppy. Since the experience also showed, that in most cases, the best camera configuration is the auto configuration (and not the manual one), the ImagingSource is used for the final system, even if it is much heavier than the Guppy. The payload of the Katana is about 0.5 kg, the Imagingsource has about 0.35 kg, but is not installed at the gripper position of the arms end effector. Further, there is no need to carry heavy objects.

4.2 Object and Camera Configuration

Working with the prototype showed that both, the hue range to search and the min. saturation of an object can change with different lighting conditions. This sometimes requires to change object configurations. We don't want to recompile our code, when we change the objects configuration. We also want to be able, to add new objects without the need to recompile. Therefore, XML configuration will be used to configure the objects and the camera.

4.2.1 A Basic XML Parser

A lot of XML libraries exist. Since we only need basic functions, we use a very simple open source lib called *xmlParser*, provided by a single programmer [1]. The configuration files `camera.xml` and `objects.xml` will be stored under `conf/`.

4.2.2 Camera Configuration

Tests with *coriander* showed, that there are four main values to be configured for the camera in manual mode: gain, shutter and white balance for red and blue. Further, there should be the possibility to change between manual and auto mode. Therefore, the `camera.xml` has following XML structure:

```
<camera>
    <!-- 0: auto or 1: manual -->
    <conf_mode>0</conf_mode>
    <!-- if manual conf_mode (1)  -->
    <shutter>2655</shutter>
    <gain>393</gain>
    <wb_blue>727</wb_blue>
    <wb_red>550</wb_red>
</camera>
```

As mentioned in 4.1, experience showed, that nearly always the best camera configuration is the auto configuration and not the manual one. Thus, we always kept `conf_mode` beeing `0`. But the configuration can always be changed, the program is reading the file and configuring the camera accordingly.

4.2.3 An Object Configuration Tool

More often than the camera, the object configurations need to be changed. These are stored in the `objects.xml`:

```
<object>
        <id>1</id>
        <name>Coffee Machine</name>
        <hue_min>0</hue_min>
        <hue_max>20</hue_max>
        <sat_min>116</sat_min>
</object>
<object>
        <id>2</id>
        <name>Blue Cup</name>
        <hue_min>100</hue_min>
        <hue_max>120</hue_max>
        <sat_min>120</sat_min>
```

```
</object>
...
```

This XML structure will be extended in 4.3.2. Since it is very difficult to estimate the needed minimal and maximal hue value and the minimal saturation, we need a configuration tool, that directly shows us the effects of changing object configurations. This tool should directly store the settings to the `objects.xml`. OpenCV provides a simple GUI library (`highgui.h`) for simple GUI elements. We use the lib to show the current frame with detected blobs and three track bars for min. hue, max. hue and min. saturation. The tool iterates over all objects stored in `objects.xml`, visualizing the color filtering and blob detection results. The user can set the configuration directly watching the effects, the settings will be stored to the `objects.xml` (Figure 10).

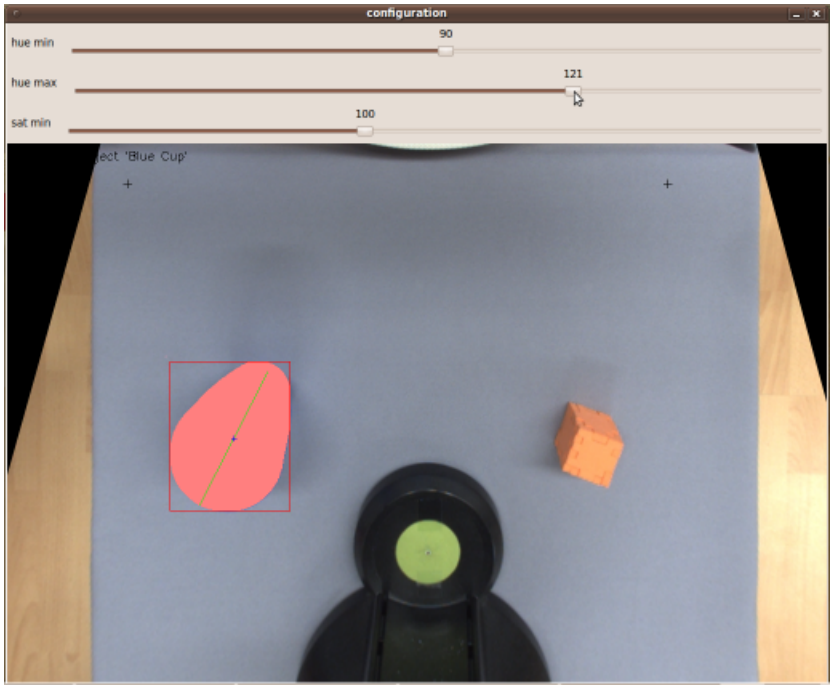


Figure 10: The configuration tool to define objects properties while configuring the blue cup.

4.3 Workflow

Due to the tilted view, the workflow differs from that in 3.2.

1. Read camera frame
2. Undistort camera frame
3. Warp frame by camera-to-table system matrix
4. Filter frame by a desired color to an black-and-white image
5. Detect white blobs on image
6. Correct blob coordinates twice (different for x and y) based on error approximations
7. Convert from imaginary table system to Katana system

The Reading of the camera frame and the undistorion are already described in 2.2 and 2.3. Steps 4 and 5 are exactly the same as within the prototype (3.2.1 and 3.2.2). The steps 3, 6 and 7 are new and will be described in the following.

4.3.1 Conversion from Camera to Imaginary Table System

Within the prototype, assuming a vertical camera view, after undistortion we already had the coordinates for the camera system. With a tilted view, at this step, we still have a perspectively distorted image. Now, there are several approaches, how we could find the coordinates in the Katana system. We take a special way: beside the coordinate systems of the camera and the Katana, we introduce a third system: an imaginary one on the scene table, lying exactly on the pattern stuck on the table (Figure 11). This has the advantage, that on the pattern, we have much points where we know the exact 3d position in the table system. That is very helpful:

1. We can see the position of these points on the recorded images, thus we have the positions in the table system and the corresponding ones on the image. This will help us to convert from the camera system to the table system.
2. We already have a program to build a conversion matrix from one coordinate system to another by providing some position pairs. Since we know the exact table positions on the pattern, we can easily collect pairs by approaching some points on the table with the Katana.
3. Having the pattern with the table system, we can also easily record positions for the error approximation.
4. Testing the vision system can be done with the pattern.

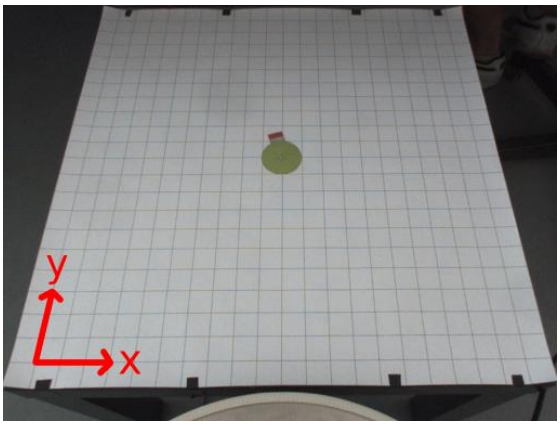


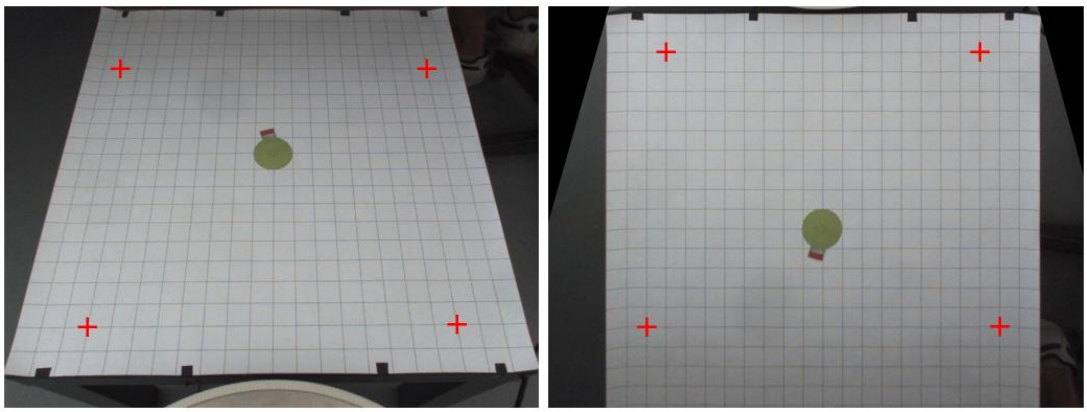
Figure 11: The imaginary coordinate system on the table.

Back to the conversion from the camera system to the imaginary table system. OpenCV provides the special function `cvGetPerspectiveTransform`, which can be used to get a conversion matrix from four corresponding points of two planes. Recording the first points from a perspectively distorted plane and the second from a desired target plane, we can use the matrix to undistort the plane from a tilted view. That can be acchieved by using `cvWarpPerspective`. This function warps an image with a perspective view (or any arbitrary image) by the perspective matrix (a single point can be converted using `cvPerspectiveTransform`). We will use this function, to directly convert the tilted view into one, as it would result with an exact vertical view as assumed with the prototype. Thus, we have also to consider the mm to pixel ratio and the offset (shift) from the image origin. Figure 12 shows the original perspective image and the result after perspective transformation (warping). The used 8 corresponding points used for `cvGetPerspectiveTransform` are shown, too. The image is flipped due to the opposite direction of the y-axis in the table and the camera system. The corresponding code follows.

It should be repeated, that the perspective conversion only works correctly in the x-y plane of the table system. That is the mentioned limitation, when using only one camera. It is clearer, what this means exactly, when taking a look at figure 13. The undistortion of the table plane

is correct, the undistortion of the blue cup, which lies not only on the x-y plane, is still clearly distorted (how still correctly to determine the cup position is described in 4.3.2).

```
double zoom = 1.5;
int x_shift = 150;
int y_shift = 50;

CvPoint2D32f src[4] = {cvPoint2D32f(159, 614), cvPoint2D32f(867, 608),
                       cvPoint2D32f(809, 119), cvPoint2D32f(223, 120)};
CvPoint2D32f dst[4] = {cvPoint2D32f(50*zoom+x_shift, 25*zoom+y_shift),
                       cvPoint2D32f(450*zoom+x_shift, 25*zoom+y_shift),
                       cvPoint2D32f(475*zoom+x_shift, 400*zoom+y_shift),
                       cvPoint2D32f(25*zoom+x_shift, 400*zoom+y_shift)};

CvMat* persp = cvCreateMat(3, 3, CV_64FC1);
cvGetPerspectiveTransform(src, dst, persp);

IplImage *dstImg  = cvCreateImage(cvGetSize(srcImg), 8, 3);
cvWarpPerspective(srcImg, dstImg, persp);
```

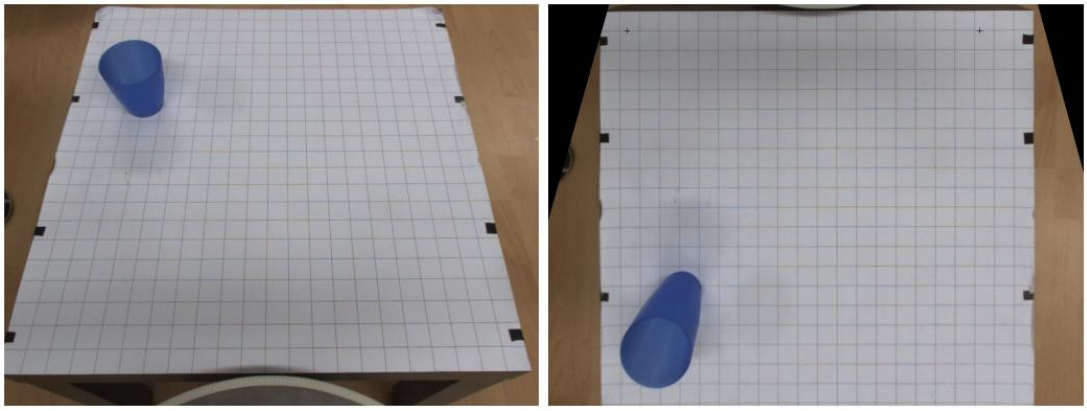


(a) (b)

Figure 12: The original perspective view (a) and the result after perspective warping (b). The eight corresponding points used for `cvGetPerspectiveTransform` are marked with a red '+'.

As mentioned, to make the whole scene appear centered on the warped frame, one has to consider the shift in x and y. If we would not apply a shift in the warped image, the shown origin from figure 11 would be in the left top corner of the warped frame, objects standing at the borders could then appear cropped. Now, using the perspectively corrected (warped) and shifted frame to detect objects, the resulting object positions in x and y will (after error approximation) already be those lying in the table system shown in figure 11!

In the future, the table does not have to stand at the same position always. That restriction has only to be fullfilled when using the pattern on the table for determining the conversion matrices. But, we always will have to use a table of the same height. Using a new height requires new conversion matrices.

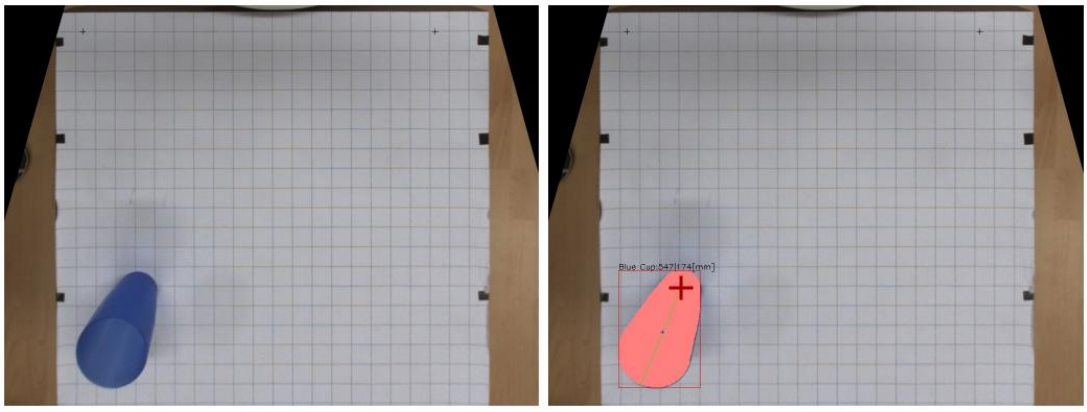


(a) (b)

Figure 13: The original perspective view (a) and the result after perspective warping (b). We can clearly see the distortion of the cup in (b).

4.3.2 Blob Coordinate Error Correction Approximation

As mentioned in 4.1, in the new hardware setup, we have a different color blob error effect than shown in figure 7. For the new hardware setup, we can see the corresponding visualization of the error in figure 14.



(a) (b)

Figure 14: The original perspective view (a) and the detected color blob (b). The red '+' marks the real cup position, which is clearly different to the blobs midpoint or to the blobs center of gravity.

Due to the tilted view, we have different errors in x and y, thus we need to approximate twice. The interesting question now is, whether each error depends only from one parameter. In this case, we could still use polynomial two-dimensional error approximations. Making different tests, one can find out, that the most suitable step to make the error approximation is right the one after the perspective warping. Only in this case, the color blob borders (not the blob center!) in one dimension (x or y) are only dependent from that single dimension, not from the other one. This fact is visualized in figure 15.

So, we can apply two-dimensional error approximations for the blob midpoint errors in x and y. The input to the function is the blob midpoint either in x or y, the output should be the real cup position on the frame (either in x or y). We record sample data points with the help of the

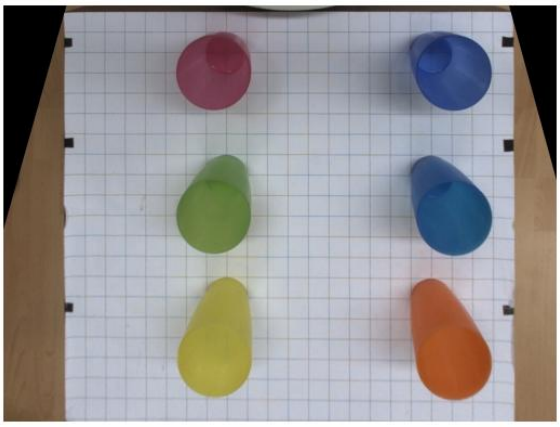


Figure 15: A perspectively corrected scene with six cups showing, that the cup borders in x depend only from the x position, the same applies to y.

grid pattern and try to approximate the error function. For the cup, it is not exactly linear in x. At the end, choosing a third order polynomials for the approximations gives very good results (figure 16).

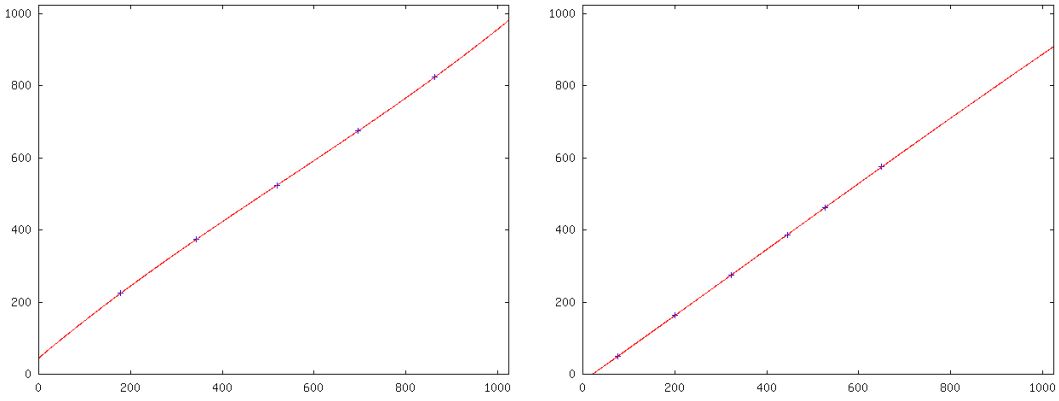


Figure 16: Approximation of the error functions in x (a) and y (b) (x-axis: blob midpoint position; y-axis: real cup position in the frame).

We need an own error approximation for each different object shape. Sample points for the determination of the polynomial coefficients can be recorded very fast, but the coeffiecients for each object have to be stored somewhere. Since we already have an XML configuration file for the objects, we store the coefficients there, too. The resulting overall configuration in `conf/objects.xml` for a single object then is as follows:

```
<object>
    <id>2</id>
    <name>Blue Cup</name>
    <hue_min>90</hue_min>
    <hue_max>121</hue_max>
    <sat_min>100</sat_min>
    <error_x_poly_coeff_1>2.6491e-07</error_x_poly_coeff_1>
    <error_x_poly_coeff_2>-0.00042601</error_x_poly_coeff_2>
```

```
    ...
    <error_y_poly_coeff_3>0.90314</error_y_poly_coeff_3>
    <error_y_poly_coeff_4>-19.088</error_y_poly_coeff_4>
</object>
```

Now, in the whole workflow, when detecting a color blob from a special object, the blobs midpoint position in x and y is read out and used as input for the approximated error function, using the cooefficients for this object. The result will be the real object position in the frame, beeing used in the further workflow.

4.3.3 Conversion from Imaginary Table System to Katana System

At the end, we need a matrix to convert the 3d position from the table system to our Katana system. As mentioned before, we already have a programm which is based on a least square algorithm described in [3]. All we have to do is, collect sample position pairs by approaching some positions on the table with the Katana, and to record both, the Katana 3d end effector position and the corresponding position in the table system. We took twelve positions as input for the program. In tests, taking some positions on the table, the resulting matrix produces very exact positions for the Katana system.

4.4 Software Design

The Software mainly consists of two classes. `CameraBlobObjectDetection` uses `dsCameraDC1394` and is a general approach to detect color blobs on frames from a firewire camera. `SpecialSetupObjDet` handles the special case we have with the camera installed on the Katana. Figure 17 shows the classes, the most important methods and the dependencies.

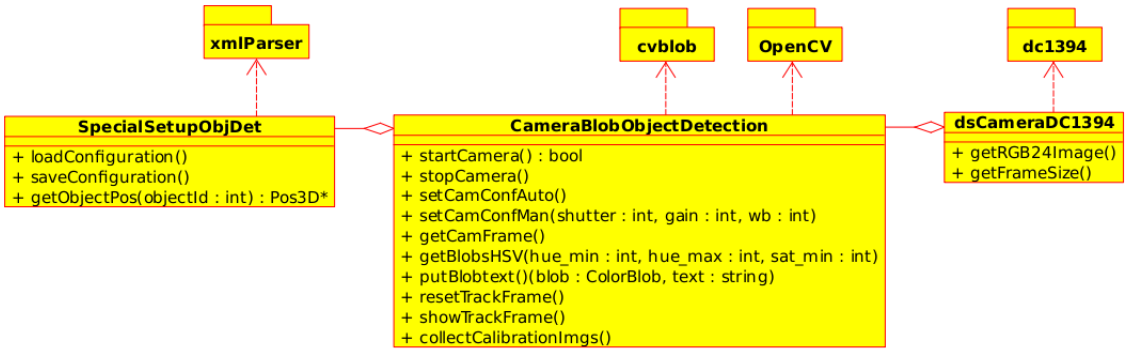


Figure 17: The UML diagram showing the main classes, the most important methods and the dependencies.

4.5 Tests

Using the pattern to make position tests with the cup, 50% of the positions are exact (without any error). At the other 50%, there is an error of 1 mm in one dimension. No case occured with an error in both dimensions. The max. error was 1 mm. It has to be mentioned, that this positions are not the final positions in the Katana system. These are very difficult to measure exactly by 1 mm. Doing tests with the Katana, no errors could be found. The final version works stable and exact in the Learning from Demonstration framework. The only major erros detected where those after changing lighting conditions. In this case, one has to use the configuration tool to reconfigure the objects (Figure 10).

References

[1] Frank Vanden Berghen. Small, simple, cross-platform, free and fast C++ XML parser. Homepage. Website, 2010. http://www.applied-mathematics.net/tools/xmlParser.html; retrieved 16 July 2010.

[2] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[3] Larry Davis, Eric Clarkson, and Jannick P. Rolland. Predicting accuracy in pose estimation for marker-based tracking. In *ISMAR '03: Proceedings of the 2nd IEEE/ACM International Symposium on Mixed and Augmented Reality*, page 28, Washington, DC, USA, 2003. IEEE Computer Society.

[4] Werner Gampp. Computergrafik, Skript zur Vorlesung. HS Ravensburg-Weingarten.

List of Figures

List of Tables