CONVOI Project Report

Koen Dercksen Marein Könings
Caspar Safarlou Chris Kamphuis Erik Verboom

January 2014

for the course of Robotica II
taught by Perry Groot and Ida Sprinkhuizen-Kuyper
assisted by Bas Bootsma

as part of Artificial Intelligence
at Radboud University, Nijmegen

Abstract

This is a simple sample of a document created using LaTeX(specifically pdflatex) that includes a figure from the Vergil visual editor for Ptolemy II that was created by printing to the Acrobat Distiller to get a PDF file. It also illustrates a simple two-column conference paper style, and use of bibtex to handle bibligraphies.

Contents

1 Project Outline

1.1 The Assignment

1.2 Task and Execution

What the robot does, brief description of steps involved. Intro to sections about code and physical.

1.3 Development Process

How the project went, timeline, overview of problems and solutions. Intro to sections about unimplemented features and team.

2 Third-Party Software

2.1 Development Tools

Overview of software used for development (language, IDE, VCS).

2.1.1 C#

including: what is it, why we chose it

2.1.2 Visual Studio

including: what is it, why we chose it (name cool features like performance analysis...)

2.1.3 GIT

including: what is it, why we chose it

2.1.4 Miscellaneous

Photoshop...

2.2 Libraries and Algorithms

As part of our project, we used several third-party libraries in our code to solve certain problems. We also made use of (mathematical) algorithms from online sources. All of these are listed below, including our reasoning for using the software rather than writing our own. In most of these cases, a primary factor was a need not to 'reinvent the wheel'. Of course we only applied this reasoning to problems that we didn't find interesting, as we did write our own code in other areas of the project. We would like to note that we have solid ideas on how we would ourselves solve most of the problems below.

2.2.1 Logitech Camera Software

The webcam we used, the Logitech QuickCam Pro 9000, comes with specialized software that includes the ability to set such properties as zoom, focus, contrast and color intensity. We use this software to pre-process the webcam input, removing noise and turning it into something that is more easily analysed by our software.

[image: Webcam input before pre-processing] [image: Webcam input after pre-processing]

We were initially unaware of the Logitech software, and considered writing our own code for pre-processing. We decided this was too much effort compared to the gain, especially since our analysis software was able to handle input without pre-processing. When we discovered the Logitech software and the ease with which is allowed us to pre-process the input, we decided to incorporate it into our analysis pipeline to further decrease errors.

2.2.2 EmguCV

EmguCV is a .NET wrapper of OpenCV, an image processing library developed by Intel. It offers a wide range of functionalities from image and motion detection to machine learning algorithms. We only utilize a very small sample from these functionalities.

Camera Connection The wrapper contains methods for very easily communicating with a connected camera. Only a couple of lines of code are needed to establish a streaming input connection. We didn't anticipate EmguCV to contain this functionality, but we were very glad to discover it as it exactly covered our needs.

We also set up a 'mock' input stream, which supplied pre-determined images to the analysis pipeline. This mock input stream proved very useful for testing purposes, especially at times when we didn't have access to the robots or camera.

Contour Detection An important step in our analysis is the conversion of blobs to contours describing the shape of the blobs. Blobs are defined by a bitmap, with bits representing one of two values, in this way defining one or multiple contiguous regions known as blobs. Contours are defined here as polygons that (roughly) describe the shape of

blobs. The important thing is that blob geometry is converted from a bitmap representation to an approximated vector representation.

[image: Blobs] [image: Corresponding contours]

EmguCV contains functionality to do exactly this, allowing some control over the algorithm used. We found the contour-finding algorithm known in EmguCV as LINK_RUNS led to the best results in our situation. Unfortunately we found no documentation on the inner workings of the algorithm, and the source code does not grant much insight [link].

Minimum Area Rectangle Many of the items we are detecting are rectangular, but may not appear exactly rectangular in the input (due to perspective and colour matching inaccuracies). As such, it is convenient to have a method of easily converting an arbitrary contour, which we have already established to represent a rectangular item, to a rectangular contour. EmguCV's 'minimum area rectangle' (MinAreaRect) functionality solves this problem for us.

[image: A contour] [image: The bounding box of the contour] [image: The minimum area rectangle of the contour]

Much like the common principle of a 'bounding box', the MinAreaRect is a rectangular box around a collection of points, such that the box contains all points, and the box could not be any smaller while still containing all points. Contrary to a regular bounding box, a MinAreaRect is not necessarily aligned to the Cartesian axes. This means that the MinAreaRect of a particular contour usually has a smaller area than the bounding box of that same contour, as the rotation of the rectangle allows for a more optimal fit. In our implementation, with items that we know to be rectangles, the MinAreaRect is simply taken as the actual shape of the item. We were again unable to find information on the algorithm used.

Convex Hull The convex hull of a set of points is the smallest subset that defines a convex polygon, such that the polygon contains all of the original points. Practically, the original set of points may be seen as a polygon, and the convex hull is a modification of that polygon with all concavities removed, resulting in a convex polygon.

[image: A concave contour] [image: Its convex hull]

We apply EmguCV's method for determining the convex hull of a polygon on contours, when we are looking for items that we know to have a convex shape. This accounts for some errors in colour matching, where part of an object was not detected as belonging to the object, resulting in a concave shape. This concave shape is made convex by converting to convex hull, removing the concavity that was missed in detection. Documentation on the algorithm is again lacking [source code].

EmguCV Representation EmguCV uses its own data formats for almost all of its functionality. Since we are using this functionality in key areas of our pipeline, we came to use those data formats in many places ourselves. This includes the representation of polygons and images, and all associated representations such as points and colours.

However, there are two problems with this established setup, both to do with performance issues. These are listed in the section on unimplemented features, as there were plans for a better solution.

2.2.3 Triangle.NET

Triangle.NET is a library that facilitates (Delauney) triangulation. Triangulation is the process of taking a polygon and dividing it up into triangles. This technique is a key part of our navigation mesh construction. Because of the key role of triangulation, and because we found it to be a difficult problem, and such a perfectly-suited library exists, we chose to use this software to perform all necessary triangulation operations.

[image: A polygon] [image: After triangulation]

2.2.4 AForge.NET

AForge.NET is a C# framework containing implementations of various A.I. related algorithms and applications such as image processing, neural networks, machine learning and robotics. We used a very small part of this framework to communicate with the LEGO NXT Brick. AForge.NET handles the bluetooth connection with the brick and supplies convenient methods to issue motor commands.

2.2.5 Native NXT Firmware

LEGO NXT Bricks come with LEGO's own firmware. Since AForge.NET was built with this in mind, we did not have to flash the brick with a different type of firmware.

2.2.6 Algorithms

Next to libraries, our project incorporates many code snippets that we found on the internet. Most if not all of these are specific mathematical or geometric operations (e.g. the angle between two vectors or the centroid of a polygon). In our code, all of the instances where we copy-pasted a third-party snippet contain a reference to the source. In total there are 12 such references.

3 Design and Implementation

— Explanation of how we go from image to movement. Interesting steps explained in more detail. Not an explana-

tion of how our code is set up, but of the logic involved. — introduction of 'pipeline' setup, introduction to different object types as they are known in code

3.1 Camera input

about size and framerate

3.2 Image Pre-processing

camera stuffz... why we change the values the way we do

3.3 Colour Calibration

Obviously, an important aspect of the analysis is knowing which colours signify which objects in the world. For the program to learn this, the user needs to calibrate the colour-object associations. The program then uses this information for all subsequent operations until the user performs a new calibration.

3.3.1 Calibration Interface

Calibration begins with choosing an object type to calibrate the colour of. The live pre-processed image is presented, and the user can click on it to indicate spots that should be seen as the colour of the chosen object type. In these spots, dots appear to indicate that the spot was clicked. These dots are of a fixed size which is of importance, as all pixels that are covered by the spot are seen as belonging to the chosen object type.

[image: Multiple selected calibration spots]

Multiple spots can be clicked to cover a particular object as much as possible, or to cover multiple objects of the same type.

When the user has clicked all spots belonging to a certain object type (or as much as they deem sufficient to calibrate the colour), they can click a button to indicate they are finished. All pixels that were covered by dots are now used as calibration data in the calibration calculation (see below), and the results of this calculation are associated with the chosen object type. The user can now choose the next object type to calibrate.

Once all object types have been calibrated, the user can choose to save the set of calibrated values to a file. This file can then be loaded at a later time, sparing the user the effort of calibrating the values over again. Of course this is only a valid shortcut if the user is running the program under the same physical conditions as when saving the file. Otherwise, the colours of objects, as perceived by the webcam, may have changed.

3.3.2 Calibration Calculation

The calibration calculation, for a certain object type o, takes as input a collection of colours C_o (the colours of those pixels that were covered with spots by the user) with each colour $c_{oi} = [r_{oi}, g_{oi}, b_{oi}]$ (for the red, green and blue channels). From these are distilled two pieces of information that form the calibration profile for that particular object type.

The first piece of information is $\overline{C_o}$, the average colour of the pixels, seen as the base colour of the object type.

$$\overline{C_o} = [\overline{r_o}, \overline{g_o}, \overline{b_o}]$$

The second piece is T_o, known as the threshold value of the object type. It is the maximum distance from the average, over all colours in the set.

$$T_o = \max_i \mathrm{d}(\overline{C_o}, C_{oi})$$

where

$$\mathrm{d}(c_i, c_j) = |r_i - r_j| + |g_i - g_j| + |b_i - b_j|$$

This last equation demonstrates that we use a component-wise colour distance function. We considered using a Euclidean distance function, but this proved less optimizable (during Colour Extraction) and we also didn't find a good reason to interpret colours as a three-dimensional space. In any case, the component-wise distance function serves us well.

3.4 Colour Extraction

At this point in the pipeline the program has received an image from the webcam and has access to calibration data detailing which colours are associated with which objects, and analysis can begin. The first step is to create masks of the input image, indicating which pixels fall within the thresholds of which objects. One such mask is created separately for each object type.

[image: Input image] [image: Colour mask for object type X]

3.4.1 Colour Masking Algorithm

To create a colour mask of a certain object type o in a given image P consisting of pixels with colours p_i, the average colour $\overline{C_o}$ and threshold value T_o of the object type are retrieved. The mask M then is an image of the same size and shape as P, with pixels of colours m_i, where

$$m_i = \begin{cases} 1 & \text{if } \mathrm{d}(\overline{C_o}, p_i,) < T_o \cdot \mu \\ 0 & \text{otherwise} \end{cases}$$

μ here is what's known as the 'threshold multiplier', a constant value that allows for more or less leniency when matching colours. $\mu = 2$ would mean that the distance of a colour to the average may be twice as large as the distance found during calibration, and still be masked as belonging to the object type. Through experimentation we settled on $\mu = 1.5$.

3.4.2 Optimization

Although this report avoids code-specific details, special efforts went into optimizing the colour masking function, which are worth pointing out. This function needed to be quite fast, as it needs to run each frame, for all object types, over all pixels in the input stream. There were several optimizations made to aid in this.

First, as much of the required information is declared and initialized before the pixels are looped over, so that it need not be extracted over and over for each pixel.

Then, there is an 'early-out' mechanism, based on the fact that colour distance is the sum of the component distances. If a single component's distance (or the sum of two components' distances) is larger than the allowed colour distance, it is clear that the total distance is too large.

Since calculation of the absolute value of a number is used in the repeated colour distance calculation, it is an intensively used function and should be as fast as possible. Because colour component values can be represented by 8 bits, we implemented an absolute value function that works with 16-bit 'short integer' variables rather than the existing 32-bit function. We use a function invented and patented by Volkonsky in 1997.

Finally, we utilize the .NET Parallel class to parallelize the loops over pixels and object type. This creates threads for each iteration of the different loops, causing the many iterations to be executed mostly simultaneously. This means that not all operations may be executed in order, but in our case this causes no problems, as pixels and object types are evaluated independently from each other.

3.5 Contour Detection

different for each object?

3.6 Object Matching

different for each object

3.7 Navigation Mesh

3.8 Pathing

3.8.1 A*

KOEN (weet nog niet in voor context het komt)

3.9 Communication

AForge.NET supplies various convenience methods to communicate with LEGO NXT Bricks through bluetooth. A serial port connection is established between the computer and the brick, and the framework converts high-level messages such as "set motor A's speed to 50" to a correctly formatted bytestring.

3.10 Control

KOEN

4 Physical Considerations

Problems, choices, etc. to do with physical things.
robot construction world construction lamp
CASPAR

5 Legacy Code and Unimplemented Features

Any features that we wanted to add but didn't, code that we worked on but dropped, thoughts on these...

5.1 Multi-Agent Interaction

5.2 Sockets

ERIK

5.3 Microsoft Robotics Studio

ERIK/CHRIS

5.4 Smart World Model

KOEN/MAREIN

5.5 Path Refinement

MAREIN

5.6 HSV Color Model

KOEN/MAREIN

5.7 Smaller Robots

5.8 EmguCV Data Formats

As described, we make use of the data formats that EmguCV brings to the table, a necessity when using the library's functionalities. There are two problems with this, with an eye on performance.

First, the EmguCV formats are not optimized for our purposes. For example, when we are using EmguCV images to store blob information, which are really binary bitmaps. The EmguCV format dictates that the image is at least a grayscale image, which is unnecessary for our purpose of storing binary values. Things such as these may contribute negatively to performance.

Second, we make use of Windows Presentation Foundation (WPF), which is not directly compatible with the EmguCV formats. WPF is a system that facilitates the creation of user interfaces for .NET applications. We use it, among other things, to display results at various stages of the analysis process. Since these results are often in EmguCV formats, they need to be converted before being handled by WPF. This takes time and probably has a large negative impact on performance. Note that one optimization we apply is to only convert the relevant data that the user is looking at, at a particular moment.

We had vague plans of taking the EmguCV formats out of our own code and only converting from and to it when interacting with the EmguCV functionality (which is actually only in very few places). This should provide a boost in performance, and create cleaner code, with only our own representations directly visible, and none of EmguCV's.

6 Demo Evaluation

7 Team

7.1 Workload Distribution

Individual description/list of tasks completed

7.2 Remarks

Individual remarks on the project.

8 Glossary of Terms

Explanation of terms that the reader may or may not be familiar with, that would be a hassle to explain in the main body of the report (example: smallest-area-rectangle). These terms, where used in the document, would have a reference to the glossary (LaTeX).

Not sure if this is super useful but we'll see.

References

Figure 1. placeholder figure