Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Cybernetics

Master's thesis

Smartphone as a Geometric Measurement Tool

January 2011

Matej Horváth

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne _____

_____
podpis

Acknowledgement

I would like to thank to everybody who enabled me to complete this thesis. Especially to my supervisor prof. Ing. Jiří Matas, Ph.D. and my advisors Ing. Michal Perd'och, Mgr. Andrej Mikulík and RNDr. Zuzana Kúkelová, whose help, suggestions and overall support significantly helped me in my work. Also, I would like to thank to the whole Center for Machine Perception for supporting me during the time.

Abstrakt

Cílem této diplomové práce bylo vytvořit univerzální aplikaci pro měření geometrických veličin s využitím algoritmů pro zpracování obrazu, běžící na mobilní telefonech. V první kapitole jsou analyzovány obecně možnosti využití mobilních telefonů pro měření geometrických veličin. Je zde představena předchozí práce Mgr. Davida Stacha, na kterou moje práce navazuje, a parametry obdobných aplikací. Další část se zabývá možnými platformami pro běh aplikace a jsou vyjmenovány vlastnosti vybrané platformy. Třetí kapitola popisuje použité algoritmy zpracování obrazu. Čtvrtá kapitola predstavuje samotnou aplikaci z pohledu uživatele. Další část popisuje architekturu aplikace a vybrané algoritmy. Dále následuje kapitola, která obsahuje výsledky testování programu z různých hledisek. A poslední dvě kapitoly shrňují dosažené výsledky, zkušenosti s vývojem a nastiňují další možnosti rozvoje a použití aplikace.

Abstract

The aim of this master's thesis was to create an universal mobile phone application for measuring geometric quantities using computer vision algorithms. In the first chapter, possibilities of measuring geometric quantities by mobile phones are analyzed. This chapter also includes a section, where a previous work of David Stach, in which my work continues, is described. Another chapter analyzes possible platforms for running such software, and the chosen platform is presented. The third chapter describes used computer vision algorithms and their roles. The fourth part reviews the application from the user's point of view. Another chapter contains the overall application architecture description, and selected algorithms are described in detail. The next chapter presents results of application testing. And the last two chapters contain conclusions, and overviews of possible future development.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Geometric measurements are most probably the oldest kind of measurements undertaken by human beings. In the earliest times, there were just simple A to B length measurements. As the time went by, new quantities were discovered and new measurement techniques were invented. Nowadays we can measure a plenty of quantities in various geometric spaces. The most common are measurements in 2 dimensions. These quantities have been measured since the oldest times mechanically by contact measurement devices, such as rulers or protractors. Quantities which cannot be easily measured, because e.g. a measurement tool for them have not been invented yet, are usually computed from the former ones by some set of relevant mathematical formulas.

The aim of my work was to modernize these measurements, make them more precise and easily accessible. For this purpose, I decided to utilize, currently higher-class, cell phones and turn them into a universal, always available measuring tool.

1.1 Smartphone As a Measurement Tool

As written in the introduction, in this work I'm using cell phones called smartphones as geometric measuring tools. The online edition of the Encyclopædia Brittanica defines *smartphone* as: *"Mobile telephone with a display screen (typically a liquid crystal display, or LCD), built-in personal information management programs (such as an electronic calendar and address book) typically found in a personal digital assistant (PDA), and an operating system (OS) that allows other computer software to be installed for Web browsing, e-mail, music, video, and other applications. A smartphone may be thought of as a handheld computer integrated within a mobile telephone."* ... *"Smartphones contain either a keyboard integrated with the telephone number pad or a standard "QWERTY" keyboard for text messaging, e-mailing, and using Web browsers. "Virtual" keyboards can be integrated into a touch-screen design. Smartphones often have a built-in camera for recording and transmitting photographs and short videos. In addition, many smartphones can access Wi-Fi "hot spots" so that users can access VoIP (voice over Internet protocol) rather than pay cellular telephone transmission fees. The growing capabilities of handheld devices and transmission protocols have enabled a growing number of inventive and fanciful applications—for instance, "augmented reality," in which a smartphone's global positioning system (GPS) location chip can be used to overlay the phone's camera view of a street scene with local tidbits of information, such as the identity of stores, points of interest, or real estate listings."*[8]

It's clear smartphones offer a vast amount of ways how to measure geometric quantities,

among others also applications of computer vision, for which I decided in this thesis. Their cameras are not perfect, but good enough for everyday life measurements. Their computational power nowadays is high enough for running advanced high-demanding algorithms, which computer vision algorithms in general are. Another positive feature of these devices is their incorporation of touchscreen technologies, which makes user-device interaction much easier. These features create a potential for smartphones to become universal measuring tools. As mobile devices they can be carried almost anywhere, and their long-time decreasing cost and widespread use make the available for a significant amount of people, with a future overview of practically everyone possessing a mobile phone [4] [1] [12].

1.2 Requirements

A system, which is supposed to run computer vision algorithms necessarily has to include these parts:

- computer system
- image input subsystem
- image output subsystem

The first requirement assumes the computer system has enough computational power for running the algorithms in a reasonable time. The second requirement is in case of smartphones satisfied either by camera, and/or input from a file stored on some storage device, e.g. a flash memory, or a miniature hard disk drive. Image output is in case of smartphones usually done on their screens, or to their storage devices. Another very important requirement, but in fact optional, is the requirement of a user input subsystem with easy positioning capabilities. This is case of smartphones satisfied by the use of touchscreen interfaces.

Requirements specific for the developed software are described later in chapter 6.

1.3 Previous Work

1.3.1 David Stach's Contribution

The previous work was done predominantly by David Stach in his master's thesis [15], also under supervision of doc. Dr. Ing. Jiří Matas. Mr. Stach presented and developed there the core functions of the computer vision algorithms used in my thesis, most importantly the Hough transform and detection of rectangles in transformed images. These functions are included in a library file called PMLibrary.dll written in the `C++` language. He also developed a very simple graphical user interface written in `C#` and naturally running under the .NET Compact Framework from Microsoft. An important "heritage" of his thesis, which significantly influenced my work is the choice of the Microsoft Windows Mobile operating system, the .NET Compact Framework and the Microsoft Visual Studio as the program's runtime and development environment.

My work was more focused on the user interface, which I developed from scratch, and extensions and corrections of the computer vision library. The aim was to make the program more commercially attractive and user friendly.

Figure 1.1: The David Stach's PocketMeter

1.3.2 Market Survey

Before any work was done, I did a simple survey of the state-of-the-art commercial solutions. To have as many inputs as possible, I decided for the Apple App Store, which is nowadays the most vital smartphone software market place.

I found 6 applications in total, none of them using any kind of image recognition. There were several approaches, but the principle in all of them was marking a known distance by hand, and then marking the unknown. Some applications let the user input the known dimensions directly, some demanded the user to fit a template silhouette to an appropriate object in the scene. Especially the latter approach cannot perform very well by means of measurement precision, because there is just one silhouette for a whole class of objects, e.g. sedan cars.

Another feature common to these applications was that they were not counting with any other measurements than those in a single x-y plane, perpendicular to the camera's axis. Any kind of breaking these constraints would result in even more measurement inaccuracy and growth of measurement error.

By means of included functionality, all the considered applications where offering just simple A to B length measurements. Measurements of areas, angles or 3D distances would be impossible, or would demand the user to compute them by hand.

Chapter 2

Runtime Environment

2.1 Platform

As stated in the previous chapter (Chapter 1), the chosen hardware platform is a smartphone capable of running 3rd party software, equipped with camera, storage subsystem and touchscreen display. According to section 1.3, the chosen software platform is .NET Compact Framework running on Microsoft Windows Mobile operating system.

This platform currently occupies approximately 10% of the global smartphone OS market [12], and it's expected to shrink in the years to follow [12]. Since one of the purposes of my work was to make the software more commercially attractive, in the beginning of the development, I was also considering other platforms, such as the Apple's iPhone, Google Android, or Symbian. I decided to remain on the .NETCF and Windows Mobile, because some work has already been done before by David Stach and I assumed, in any case, it would not be impossible or just unacceptably complicated to rewrite the program for any other platform.

2.2 Development Tools

Because of the previous work of David Stach and the decision to remain on Microsoft Windows Mobile and the .NET Compact Framework, the used programming language for the user interface and a minority of image processing functions was the `C#` programming language, the majority of image processing code is written in `C++` and is included int the PMLibrary.dll.

There were three main reasons for splitting the project to two programming languages and two software platforms. One reason is that the `C#` programming language along with the .NET Framework is marketed by Microsoft as, and also widely believed to be a tool for rapid development of software. The second reason for the split is the widely believed rich offer of features in Microsoft's development tools, which are supposed to create a comfortable development environment. The last reason for splitting the project was a fear of slow performance of the .NET Compact Framework and the `C#` for running computer vision, and in general high-demanding algorithms. Due to this reason, the vast majority of the image processing functions is written as a native code, not as the user interface, which is running in the .NET's Common Language Runtime virtual machine.

2.2.1 Microsoft Development Tools

Development of an application in `C#` under the .NET Compact Framework is practically possible only in the Microsoft's Visual Studio with Microsoft Device Emulator or a real device with the Windows Mobile operating system installed. Possibilities for development of an application or a library in `C++` are more diverse, there is a lot of integrated development environments (IDEs), but since the `C#` part of the project was tied to the Visual Studio, I decided also in this case for this IDE. Most of the work was done in Microsoft Visual Studio 2008 Professional, with Microsoft Windows Mobile in version 6.1 Professional, running in Microsoft Device Emulator, or on a real device, and with the .NET Compact Framework in version 3.5 . The real device was mostly a HTC Touch HD borrowed from doc. Matas, connected via a USB cable.

Processor	Qualcomm® MSM 7201A™ 528 MHz
ROM	512 MB
RAM	288 MB
Display	3.8-inch TFT-LCD flat touch-sensitive screen with 480 x 800 WVGA resolution
Camera	Main camera: 5 megapixel color camera with auto focus
Operating System	Windows Mobile® 6.1 Professional

Table 2.1: HTC Touch HD - Technical details

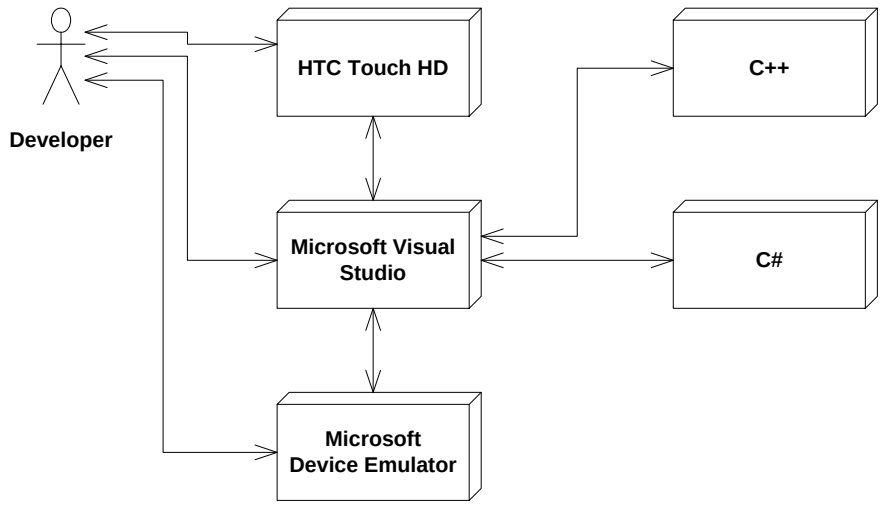


Figure 2.1: A typical development setup

Chapter 3

Image Recognition

There are two main branches of object recognition algorithms, one based on region-based segmentation, the another based on edge-based segmentation. The first method considered was based on region-based segmentation, more specifically on Watershed segmentation, which is in theory a quite straightforward concept. But because the development of this kind of algorithms is a complex task, with many of the early methods resulting in either slow or inaccurate execution [14], the implemented algorithm was a very simple one due to constrained computational power of smartphones, and in the end performed well in strict laboratory conditions, but poorly in real life setups. The program for example required the object to highly contrast with it's background, to have only one color, and even just very ordinary reflections on the object's surface were causing the segmentation to fail. Therefore another approach was necessary. The decision was made in favor of edge-based recognition algorithms.

3.1 Edge Detection

Edge detection is the first step of the implemented image recognition algorithm. Edges are pixels where the image color intensity function (brightness) changes abruptly [14]. In other words, an edge is a property attached to an individual pixel and is calculated from the image function behavior in a neighborhood of that pixel. It is a vector variable with two components, magnitude and direction. The edge magnitude is the magnitude of the gradient, and the edge direction ϕ is rotated with respect to the gradient direction ψ by -90°. The gradient direction gives the direction of maximum growth of the function, e.g., from black $f(i, j) = 0$ to white $f(i, j) = 255$ [14].

Sharp changes in image brightness are interesting for many reasons. Firstly, object boundaries often generate sharp changes in brightness - a light object may lie on a dark background, or a light object may lie on a dark background. Secondly, reflectance changes often generate sharp changes in brightness which can be quite distinctive - zebras have stripes and leopards have spots. Cast shadows can also generate sharp changes in brightness. Finally, sharp changes in surface orientation are often associated with sharp changes in image brightness [5].

(a) The original scene

(b) The same image with edges detected

Figure 3.1: An example of edge detection [15]

Edge detectors are a collection of very important local image pre-processing methods used to locate these changes in the intensity function. In these detectors, there are individual gradient operators that examine pixel small local neighborhoods. These operators are in fact convolutions, and can be expressed by convolution masks. Operators which are able to detect edge direction are represented by a collection of masks, each corresponding to a certain direction [14]. In the beginning the Sobel operator was considered, but the tests revealed the Prewitt operator is giving much better results, therefore a 5x5 Prewitt operator is used. Here the gradient is estimated in eight (for a 3 x 3 convolution mask) possible directions, and the convolution result of greatest magnitude indicates the gradient direction. More information about the operators can be found in [14].

$$h_1 = \begin{bmatrix} -2 & -2 & -2 & -2 & -2 \\ -1 & -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix} \tag{3.1}$$

The 5x5 Prewitt operator for one direction

$$h_1 = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{3.2}$$

The 3x3 Sobel operator for one direction

3.2 The Hough Transform

If an image consists of objects with known shape and size, segmentation can be viewed as a problem of finding this object within an image. One very effective method that can solve this problem is the Hough transform, which can even be used successfully in segmentation of overlapping or semi-occluded objects [14] .

The input to the transform is typically an image of detected edges in some original image. These points of interest have a value other than the value of the scene background, e.g. the points of interest have value 1, the background pixels have value 0. The basis for the Hough transform is the knowledge of the shape of the object we are looking for, and it's parametrization. For example a 2D line can be characterized by an equation $0 = ax + by + c$, where x and y are coordinates of points lying on the line, and a, b and c are the line parameters. The principle of the Hough transform is then *moving* the shape across the input image, so that the shape's *defining point* will be in each step at a position of a different interest point (those with value 1), so that it will *walk* through all of them. A *defining point* can be e.g. in case of circles their center, in case of lines a point lying on them. When the shape is at an interest point position, locations of points of the shape are computed for several pre-defined combinations of it's parameter values. In case of lines defined as above, it would be parameters a, b and c. In theory there are in general infinitely many combinations of values, therefore in real implementations of this algorithm, finite sets of value combinations are used. The number of all interest points lying at the same position as the points of the shape is then stored for every parameter setting. These numbers are in each step summed with previous numbers belonging to the same parameter setting, regardless of the *defining point* positions, and altogether create a frequency map of parameter settings, which is the final transformed image. For the line example, the map would have 3 dimensions corresponding to the 3 parameters a, b and c. The local maximums in this image (frequency map) are then those parameter settings which cover the most interest points, which means the shapes with these parameters are the best approximations of the shapes *behind* the interest point (edge) images. This also means the edge points of an object don't have to create a solid path, they may be interrupted, and yet, the transform is able to find the underlying shape.

A Hough transform of the image in Figure 3.1a with line parametrization $xcos\varphi + ysin\varphi = r$ is shown in Figure 3.2. A Hough transform of the same image, with the same line parametrization, but with considering edge directions by accepting just edge pixels with an edge angle in some certain interval, is shown in Figure 3.3. The edge directions are obtained from the edge detection described in section 3.1 by applying the Prewitt operator in certain directions. The result of applying this constraint is a transformed image with more distinct local maximums and therefore further processing is easier in this case.

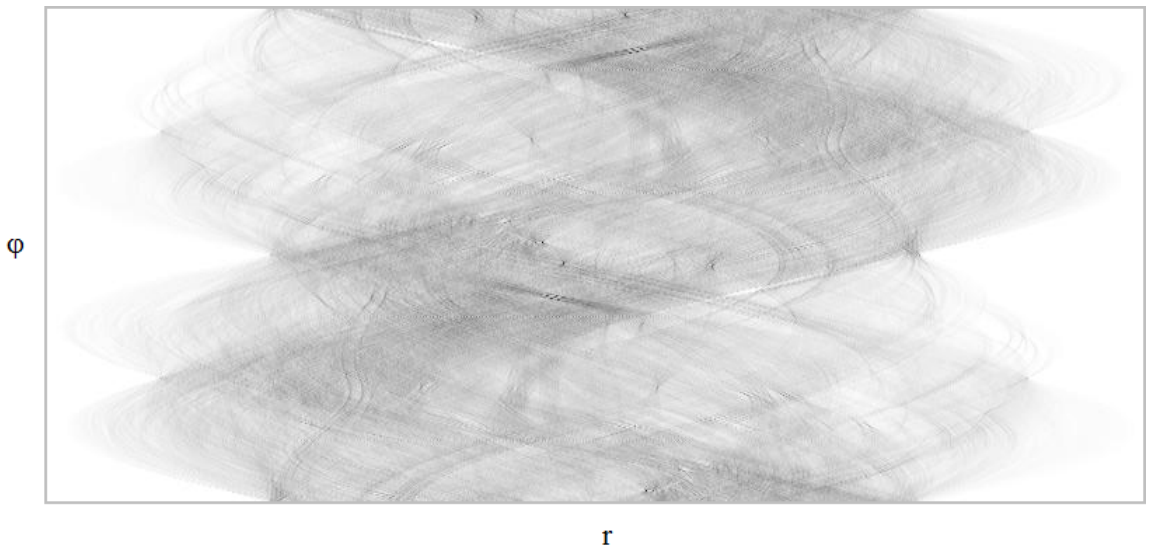


Figure 3.2: A Hough transform of the image in Figure 3.1a[15]

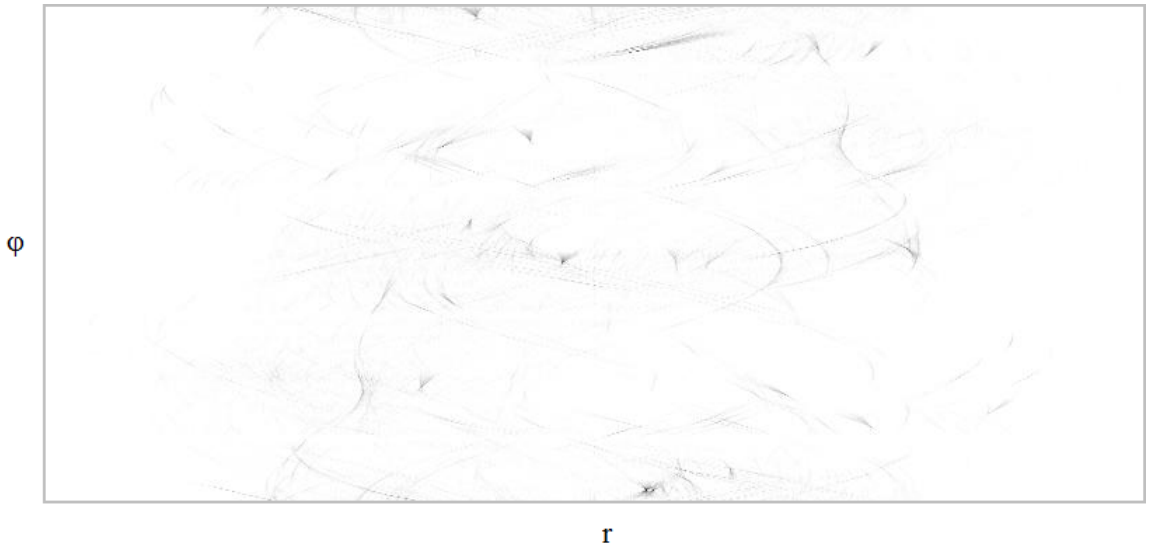


Figure 3.3: An edge direction aware Hough transform of the image in Figure 3.1a[15]

3.3 Shape Recognition

In this step of a rectangular object recognition is the recognition of the rectangle itself. The input here is the output of the Hough transform - the line parameters with the best edge approximations. A set of rules is applied to these data and the most promising (possibly deformed) rectangles defined by their edge points are returned.

The lines have to fulfill these requirements to be marked as lines of the object we are looking for:

1. A line has to cover at least some number of edge points
2. A line may have at most a certain amount of perpendicular edges in it's neighborhood
3. Opposite lines of a convex quadrangle have to be parallel, with some diversion tolerance
4. Opposite lines of a convex quadrangle have to have opposite edge directions
5. Opposite lines of a convex quadrangle have to have some minimal distance between them
6. The other sides of a convex quadrangle have to be perpendicular to the previously found two lines, with some diversion tolerance
7. These lines have to fulfill the same requirements as the previously found ones
8. A valid convex quadrangle is that, which has some certain side ratio

The purpose of the first two constraints is to eliminate inferior lines and therefore speed-up the computation. The next five constraints are there to find potential convex quadrangles. The opposite lines have to have opposite edge direction, because one line has to lay on a transition form the background to the object, and the another has to lay on a transition from the object to the background. The last constraint is there to eliminate the most improbable and therefore inferior quadrangles. The rectangle corner points are then computed as intersections of those side lines.

Figure 3.4: The recognized object with other edge lines[15]

3.4 Homography

Homography is any mapping $\mathcal{P}^d \to \mathcal{P}^d$ that is linear in the embedding space $\mathcal{R}^{d+1}$. It's also known as collineation or projective transformation [14].

In case of this thesis, homography is used to transform raw images supplied by the image inputs (e.g. a camera) to bird view, so that it is possible to undertake measurements even when the measurement plane is not perpendicular to the camera axis.

If we consider a 2 dimensional space of images taken by a smartphone camera, the basic problem to be solved when looking for a homography is to find a vector $\vec{a}$:

$$\vec{a} = [a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33}] \tag{3.3}$$

by solving the equation:

$$\begin{bmatrix} u_1^1 & u_2^1 & 1 & 0 & 0 & 0 & -x_1^1u_1^1 & -x_1^1u_2^1 & -x_1^1 \\ 0 & 0 & 0 & u_1^1 & u_2^1 & 1 & -x_2^1u_1^1 & -x_2^1u_2^1 & -x_2^1 \\ u_1^2 & u_2^2 & 1 & 0 & 0 & 0 & -x_1^2u_1^2 & -x_1^2u_2^2 & -x_1^2 \\ 0 & 0 & 0 & u_1^2 & u_2^2 & 1 & -x_2^2u_1^2 & -x_2^2u_2^2 & -x_2^2 \\ u_1^3 & u_2^3 & 1 & 0 & 0 & 0 & -x_1^3u_1^3 & -x_1^3u_2^3 & -x_1^3 \\ 0 & 0 & 0 & u_1^3 & u_2^3 & 1 & -x_2^3u_1^3 & -x_2^3u_2^3 & -x_2^3 \\ u_1^4 & u_2^4 & 1 & 0 & 0 & 0 & -x_1^4u_1^4 & -x_1^4u_2^4 & -x_1^4 \\ 0 & 0 & 0 & u_1^4 & u_2^4 & 1 & -x_2^4u_1^4 & -x_2^4u_2^4 & -x_2^4 \end{bmatrix} \vec{a}^{\mathsf{T}} = 0, \tag{3.4}$$

where $x_{1,2}^i$ and $u_{1,2}^i$ are coordinates of corresponding n points ($i \in \langle 1; n \rangle$) in the images. The homography matrix is then:

$$\mathbb{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}. \tag{3.5}$$

Transformation of a point with coordinates $u_{1,2}$ in one image to coordinates $x_{1,2}$ in another image is then:

$$\vec{x}^{\mathsf{T}} = \mathbb{A}\vec{u}^{\mathsf{T}}. \tag{3.6}$$

The opposite transformation of a point, from coordinates $x_{1,2}$ in the second image to coordinates $u_{1,2}$ in the first image is then:

$$\vec{u}^{\mathsf{T}} = \mathbb{A}^{-1}\vec{x}^{\mathsf{T}}, \tag{3.7}$$

while

$$\vec{x} = [x_1x_3, x_2x_3, x_3] \qquad \vec{u} = [u_1u_3, u_2u_3, u_3]. \tag{3.8}$$

[7]

3.4.1 Estimation

Homography can be computed by solving the equation (3.4) e.g. by the Gaussian elimination. For this, at least 4 point correspondences have to be known. In case of this work, those are the 4 corner points of the known object in the raw and bird view image.

3.4.2 Image Transform

Raw image can be transformed to bird view according to formula 3.7, where the $x_{1,2}$ are the points of the bird view image, and the $u_{1,2}$ are the points of the raw image. Points in the bird view image corresponding to transformed points with coordinates not in the raw image, are supplied by a default background color value, e.g. black.

3.5 The 4-Point Algorithm

Camera pose is determined by the 4-Point algorithm first introduced by Lihong Zhi and Jianliang Tang in [18]. The algorithm has several advantages: it's linear, stable, in general provides a unique solution and at last, but not at least, it's easy to implement.

3.5.1 Geometry of Camera Pose From Four Points

Before any pose computation can be done, according to [6] in any case these preconditions have to be necessarily satisfied:

- calibrated camera
- model with feature-points
- corresponding points on the screen (image-plane)

In the camera pose determination we want to obtain a rotation matrix and a translation vector that minimize the geometric reprojection error:

$$\sum_i \|\mathbb{R}\vec{p_i} + \vec{t} - \lambda_i \vec{q_i}\|/\lambda_i \to min., \tag{3.9}$$

where $\vec{p_i}$ is a set of world points, $\vec{q_i}$ is a set of normalized image points, $\mathbb{R}$ is a rotation matrix, $\vec{t}$ is a translation vector and λ_i models the depth of a point from the given view [3].

For this we will have to know at least 4 points as stated in [13]. This number of points solves the ambiguity present when solving the problem with less points. According to [16] for example, 3 points generate up to four possible solutions, which makes them insufficient for general pose estimation. Therefore, in the following text I'm considering just the case of camera pose determination from 4 points.

Let the P be the calibrated camera center, A, B, C, and D the control points. Let $p = 2cos\angle(BPC)$, $q = 2cos\angle(APC)$, $r = 2cos\angle(APB)$, $s = 2cos\angle(CPE)$, $t = 2cos\angle(APE)$, $u = 2cos\angle(BPE)$. The *4-point pose estimation equation system* is then [18]:

$$\begin{array}{rcrcrcrcl}
X_1^2 & + & X_2^2 & - & X_1X_2r & - & |AB|^2 & = & 0 \\
X_1^2 & + & X_3^2 & - & X_1X_3q & - & |AC|^2 & = & 0 \\
X_2^2 & + & X_3^2 & - & X_2X_3p & - & |BC|^2 & = & 0 \\
X_1^2 & + & X_4^2 & - & X_1X_4s & - & |AE|^2 & = & 0 \\
X_4^2 & + & X_3^2 & - & X_3X_4t & - & |CE|^2 & = & 0 \\
X_2^2 & + & X_4^2 & - & X_2X_4u & - & |BE|^2 & = & 0
\end{array} \tag{3.10}$$

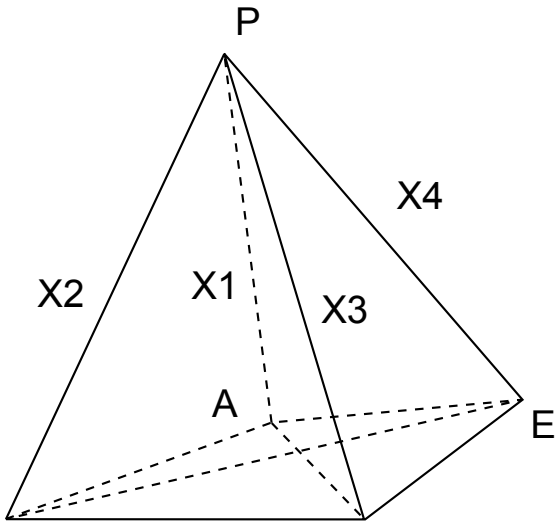


Figure 3.5: The 4-point pose estimation problem [18]

The computed camera-point distances X_i are used to estimate coordinates of the 3D reference points in camera-centered 3D frame: $\vec{P}_i = \vec{X}_i \mathbb{K}^{-1} u_i$. Then the absolute orientation is computed. The translation and scale is determined directly after determining the rotation [18].

Chapter 4

Application Overview

From the user's point of view, the application is split to 3 sections, the upper section, the middle section, and the lower section. In figures below you can see the application both after start (4.1a) and in use (4.1b) running in the Windows Device Emulator as it would look in a real device. Images in the following sections will show just the screen, and not the whole device.

(a) After start

(b) In use

Figure 4.1: The new PocketMeter

The upper section's purpose is mainly to display instructions for the user and the measurement results. It also contains two general purpose buttons for zooming the image in (Figure 4.2k) and out (Figure 4.2j).

The middle section serves as the main part of the application where most of the work is done. It displays either an image with measurements, or a panel with various controls.

The lower section harbors several buttons, by which the application is controlled. Detailed descriptions of these buttons are in the following sections.

4.1 Buttons

Here is a general description of all buttons with icons in the application:

(a) (b) (c) (d) (e) (f) (g) (h) (i) (j) (k) (l) (m)

Figure 4.2: Icon buttons

a. Take a picture with the smartphone's camera
b. Run an image recognition
c. Reset all measurements and return zoom to the default setting
d. Run length measurement
e. Run angle measurement
f. Run area measurement
g. Load or save a measurement or an image
h. Display application settings
i. Quit the application
j. Zoom out
k. Zoom in
l. Erase one vertex in a measurement
m. Erase a whole measurement

4.2 Global screen

The global screen is the *root* of all user interaction with the application. From here, users can take a photo, load and save measurements, run recognition, switch to various measurement modes, change application settings, navigate in the image, reset the measurement or exit the application.

Figure 4.3: The global screen

4.3 Navigation

Users can navigate in an image any time it's displayed. The situation overview is depicted in a navigation box in the upper left corner (Figure 4.4). For changing the window position in the image, the user can simply drag the image, or move the rectangle in the navigation box. In case of freehand input, dragging image is disabled, but navigation by the navigation box remains.

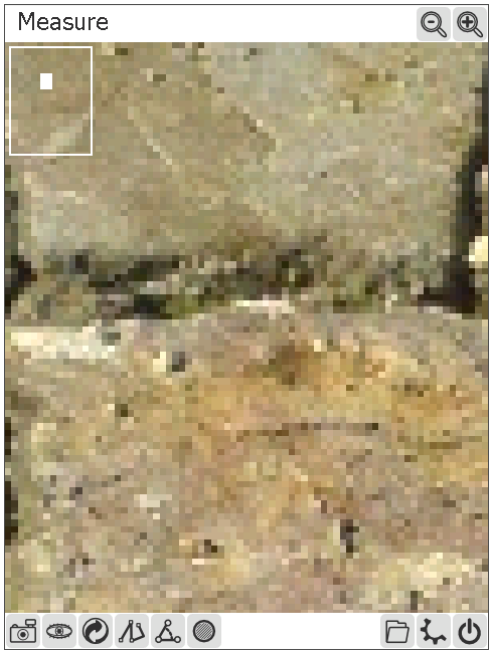
Figure 4.4: The navigation box

For zooming in and out, the application offers two buttons in it's upper section (Figures 4.2k and 4.2j). Both of these buttons work in a double regime: if a button is hold just

shortly (up to 1 second), the image is zoomed in or out just over one step, if it's hold over a longer period of time, in case of zoom out, the whole image is fitted to the screen, in case of zoom in, the user is asked to mark a rectangle to which the application should zoom, by marking it's two corners laying on the same diagonal line. The step size can be configured in the application settings shown in section 4.8.

(a) Waiting for the rectangle

(b) Zooming in



(c) After zooming in

Figure 4.5: Zoom in: before, during and after

4.4 Recognition

Users can choose one of two recognition modes, automatic or manual. In the automatic mode the application tries to find the known object automatically. In the manual mode, users are asked to mark 4 corners of the known object. If the user places a corner marker to a wrong place, he/she can still move it to the right location. After these markers are at the right places, the input can be confirmed by touching the screen once more as if placing another marker. The automatic mode falls to manual mode, if the program is unable to find the known object in the loaded image. There are in total 3 predefined known objects: a credit card, an A4 paper, and a 100CZK banknote. Beside these, users are also able to input the object dimensions directly (Section 4.8).

4.5 Scene Views

After the known object is recognized, the user is able to switch between two scene views: a raw view and a *bird* view. The raw view shows the scene as it was taken by the camera. The bird view shows the image transformed to a perspective where the measurement plane is parallel to the camera plane, or in other words, the measurement plane is perpendicular to the camera's main axis. Both views are equal by means of available application functionality, i.e. users are able to undertake measurements, move and zoom the image, etc. in both of them. Switching between the two views can be done in the application settings (Section 4.8).

In the application settings, users can also display a scene grid. Cell distances can be configured there as well - the distance is equal to a unit length of the selected grid scale unit (Section 4.8). Every fifth line is thicker than the rest of the grid.

(a) After automatic recognition

(b) During manual recognition

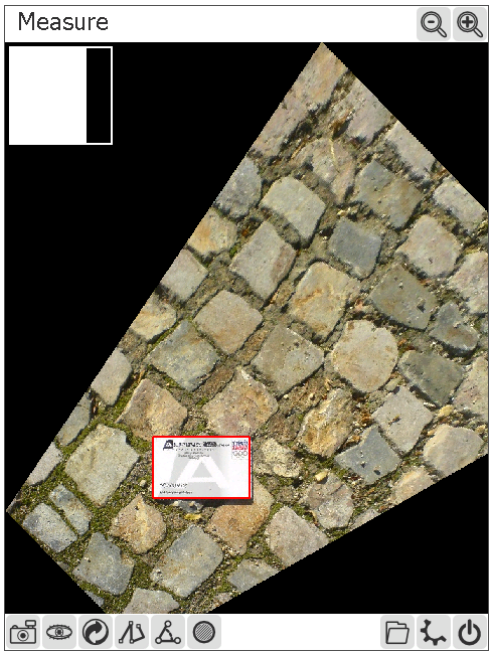
(c) After manual recognition

Figure 4.6: The two recognition modes

4.6 Measurements

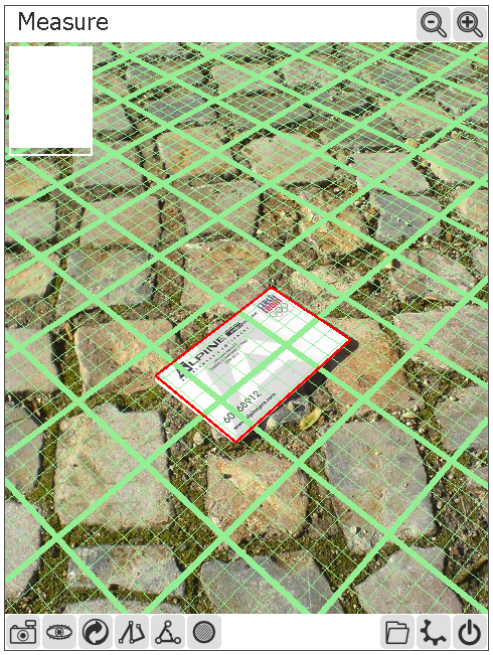
Users are able to measure 3 quantities in total: distance, angle and area, all in one plane. The camera and measurement planes are not expected to be parallel, an angle between them is considered. In all three measurements, users are able to move and zoom the image, and input, move and erase measurement points. In case of length and area, users can choose between input by vertices, or a freehand input (can be combined). Users can also change units of measurements. Both of these options can be found on the Settings

(a) Raw

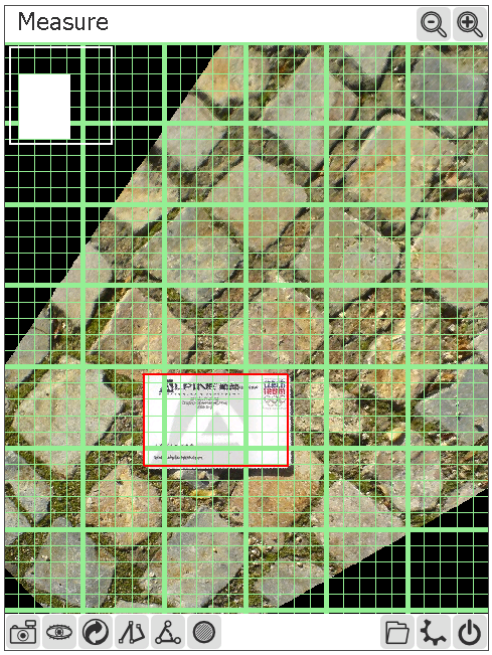


(b) Bird

Figure 4.7: The two scene views



(a) Raw



(b) Bird

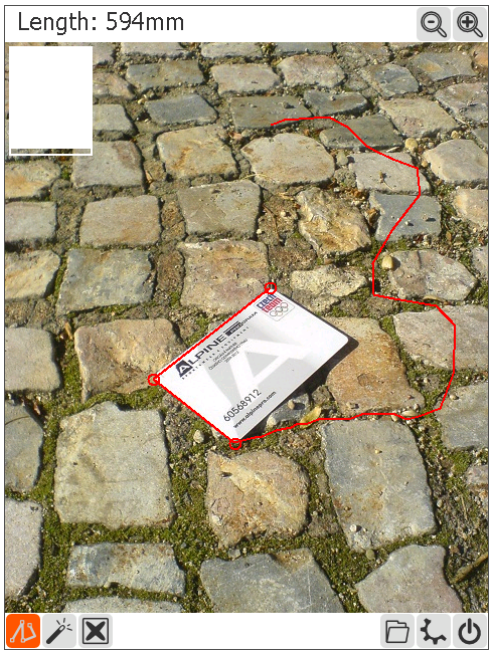
Figure 4.8: The two scene views with the grid displayed

panel 4.8. In case of incorrectly placed vertex, the user is able to either erase the vertex by pushing the appropriate button (Figure 4.2l), erase the whole active measurement (Figure 4.2m), or drag the vertex to a new location.

(a) Length

(b) Area

Figure 4.9: Freehand measurements



(a) Length

(b) Angle

Figure 4.10: Combined vertex-freehand measurements

(a) Length

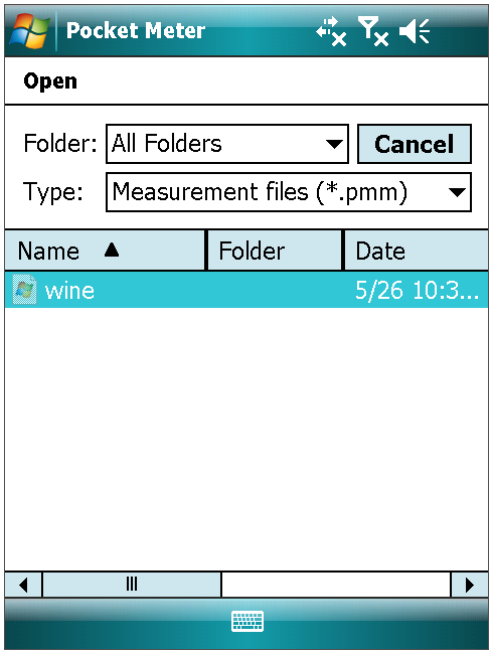
(b) Angle

(c) Area

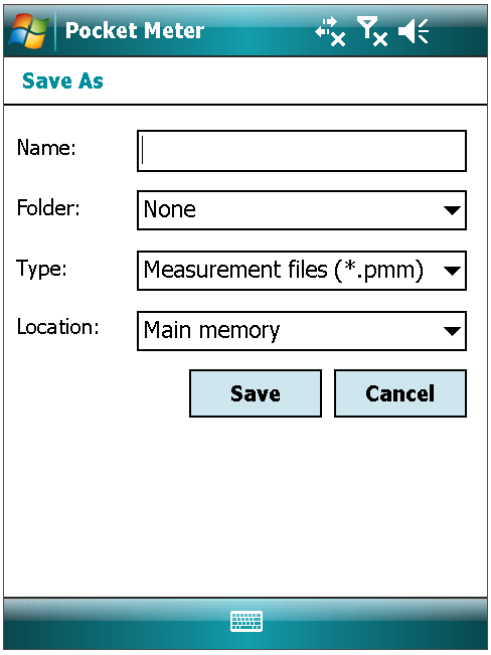
Figure 4.11: Vertex measurements

4.7 Loading and Saving Measurements and Images

Images and measurements can be loaded and saved in the File panel. The panel contains 3 buttons: “Load”, “Save As” and “Save”. The user is able to save and load either a whole measurement, or just an image. Measurements are stored as an XML file with extension .pmm (PocketMeter Measurement) and a PNG image file. Images can be stored as and loaded from PNG or JPEG files. Due to .NET Compact Framework’s restrictions, it’s possible to load and save files anywhere elsa than the \\`My Documents`\\ directory.



(a) Open



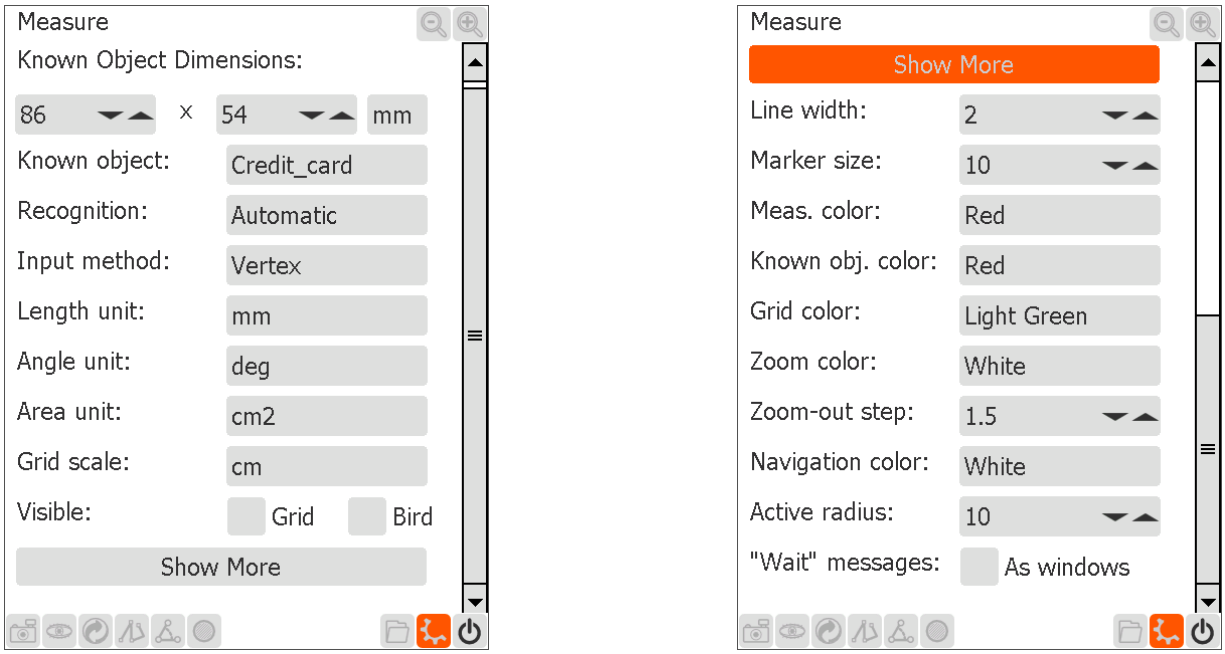
(b) Save As

(c) The File Panel

Figure 4.12: File Dialogs

4.8 Settings

The application options can be found in the Settings panel. The panel is split to two sections, the upper section contains frequently used options, the lower section, which can be displayed or hidden by pushing the *Show More* button, contains additional options.



(a) Upper section

(b) Lower section

Figure 4.13: The Settings panel

In the upper section users can set the reference object dimensions either directly, or by choosing one of the predefined objects. Then, they can set there the recognition type to automatic or manual, select input method to vertex or freehand input, units used in measurements, grid scale (note that the grid cell width and length is equal to unit length of the selected grid scale unit), and finally whether the grid should be displayed or not and whether the scene should be displayed from the raw or, from the bird perspective.

In the lower section, users can typically set appearance options, but also zoom step size and the active circle radius, which is an imaginary touch sensitive circle around every marker (Section 5.4.2).

Chapter 5

Application Architecture

From the developer's point of view, the program is split to two main parts. The first one is a so called *managed code*[17], it's written in the `C#` language running in the Microsoft .NET Common Language Runtime virtual machine, the second part is native code written in the `C++` programming language.

The native code contains image processing and object recognition functions, and it's included in a dynamically linked library (DLL) called PMLibrary.dll.

The managed code contains predominantly the graphical user interface (GUI) of the application, but also some simple graphic functions for cases when using native code would not pay off. This situation occurs, when the amount of processed data is relatively small. In this case, data transfers between managed and native code would make up most of the execution time.

5.1 General Problems and Solutions

Most of the problems in development were related to the .NET Compact Framework itself. It does not include so much functionality as the desktop .NET branch, and programs written in it are relatively slow compared to native programs. In this section I will focus on the latter problems.

5.1.1 Native and Managed Code Cooperation

Because of the poor .NET Compact Framework speed performance, the majority of the image processing functions was written in the `C++` language. Since the main part of the application is managed code written in `C#` , data transfers between these two parts have to be done. The interfacing functionality is included in a class `Lib.Gfx.TransApi` on the side of the managed code, where functions from the PMLibrary.dll library are being called.

5.1.2 GDI Graphics

Another part, where the .NET's speed performance was an issue, was the `C#` code alone. Here, functions dealing with graphics are called frequently. These functions are characterized by relatively large amounts of data which they process. As the .NET performs poor in speed, and users demand program reactions in some reasonable time, I had to look for an another solution. I solved this problem by using the Windows native graphics

API called Graphics Device Interface, or the GDI. All GDI functions are available in a Windows Mobile library called `coredll.dll`. Most notably, from this library I'm using the well-known functions `BitBlt()` and `StretchBlt()` instead of the .NET function `drawImage()`, which enabled me to significantly accelerate image drawing in the application to the present speed. With the `BitBlt()` and `StretchBlt()` functions I'm able to get almost instant responses, while the `drawImage()` function is approximately 10 times slower.

5.2 PMLibrary

This library file contains most of the image processing functions. It is mostly a work of David Stach, but I also made some contributions. The code is optimized for fast image processing on mobile devices. Before further processing, images are downsized and transformed to gray scale, which significantly reduces the amount of data and in the end, also the processing time. Another speedup is achieved by not using mathematical functions from libraries, but searching their values in pre-calculated tables. This way is for example the `sqrt()`, or the trigonometric functions done.

One of my contributions, among others, is switching the whole design from integers to real numbers. David Stach's reason for using integers was to speedup the functions, but the most significant drawback of using integers is the creation of rounding error. This was probably not a problem in case of David's work, but as I kept adding more advanced measurements, in particular cases, the accumulated error was unacceptably high, or was even causing the application to malfunction.

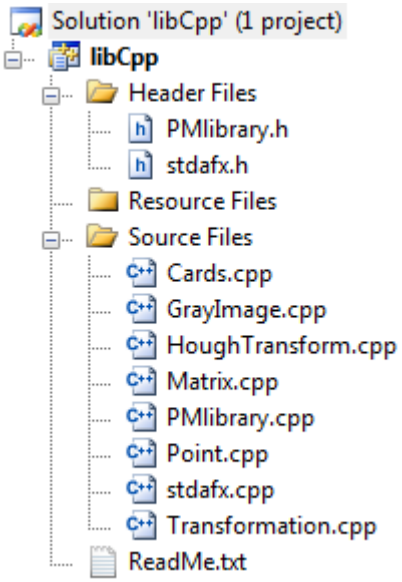


Figure 5.1: The PMLibrary project structure

5.3 BetterControls

The BetterControls is a subproject of the new PocketMeter project, which initially was not expected to exist. It emerged as the number of classes inheriting from the .NET Compact Framework's `System.Windows.Forms` package classes became significant. The reason for this is the major exclusion of methods, attributes, classes and packages in the .NET Compact Framework, which are present normally in the desktop .NET branch, or in many other frameworks. The subproject is not meant to be a replacement of the .NET Compact Framework's `System.Windows.Forms` package, but rather a supplement fit for the demands of this thesis. The components were created in a slightly universal fashion, but not entirely. Functionality which is not necessary for the project and would take a significant time to implement, was omitted.

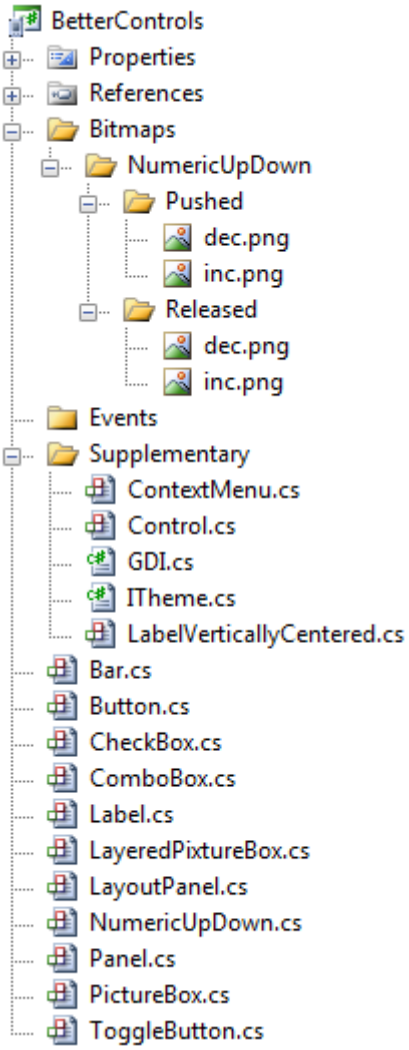


Figure 5.2: The BetterControls project structure

5.3.1 Bar

`Bar` is a simple class created by inheriting from the class `System.Windows.Forms.Panel`. It's purpose is to display a one pixel wide black border around the control. Whether a

border is displayed on a particular side is defined by four boolean properties of instances of this class.

5.3.2 Button

This class is as it's name suggests a simple button. The reason for creating it was the lack of icon support in the .NET's `System.Windows.Forms.Button` class. Basically it has two states, released and pushed.

There are 4 bitmaps in the class that have the same dimensions as the control. One bitmap stores the released button look, second bitmap stores the pushed button look, third bitmap stores the disabled button look and finally the fourth stores the actual button look. These bitmaps are recreated after every change of an instance's parameters, otherwise they remain unchanged and serve as a caches of what should be drawn in the standard method `OnPaint()`. This approach creates faster controls than if they are being drawn dynamically.

Except an icon, a `Button` instance can also display text, or have rounded edges. Basically there are 5 colors, a base color, a background released color, a background pushed color, a foreground released color, and a foreground pushed color. Base color lays on the lowest layer which cannot be rounded, back colors lay on the middle layer which can be rounded, and finally foreground colors are the colors of the possible button text. These properties can also be controlled via the instance's public parameters.

(a) Released (b) Pushed (c) Disabled

Figure 5.3: A `Button` with icon

5.3.3 ToggleButton

The `ToggleButton` class inherits from the class `Button`, and does nothing else than dividing the pushed-to-click frequency by two, i.e. it remains pushed after a first click, and is released after another, second click. This component is completely missing in the .NET Compact Framework.

5.3.4 CheckBox

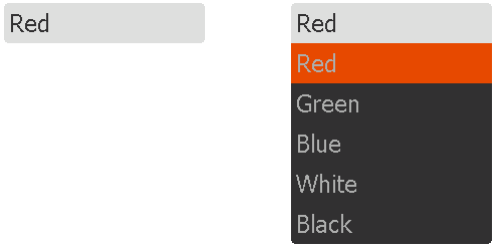
The `CheckBox` class differs from the `CheckBox` in the .NET framework by the ability to define it's colors. These color are then loaded from the application specific theme, instead of the system theme.

5.3.5 ComboBox

The `ComboBox` class is a replacement for the .NET's `ComboBox`, but it doesn't inherit anything from it, it's built from scratch. Basically it contains the same functionality as

the .NET's `ComboBox`, but additionally developers are able to define their own colors and therefore use an application theme, and unlike the .NET's `ComboBox`, it's menu is able to hold also items other than string. This means a developer are able create for example a color picker combo box, where items would be nothing but color.

The menu of this control is a single class in the `BetterControls` project. It's located in the package `Supplementary` and it's name is `ContextMenu`. More about this component can be found in Section 5.3.9.1.



(a) Closed (b) Opened

Figure 5.4: A `ComboBox`

5.3.6 LayeredPictureBox

`LayeredPictureBox` is a class meant as a replacement of the .NET's `PictureBox` class. The initial reason for creating this class was lack of a display component with support of layers in the .NET Compact Framework. The other reason, which emerged later in development was the need for a component for fast image displaying. Drawing images with the .NET's methods showed up to be very slow, therefore drawing graphics with the Windows native API, the GDI, became necessary (Section 5.1.2).

The `LayeredPictureBox` contains two layers, one layer where a background image is located, and an upper layer, where shapes, such as lines, circles, or rectangles are being drawn. These shapes are stored in form of a list, in which the particular shape instances are objects. The final image is generated as a composite of these two layers, where the background image is drawn first, and then the shapes from the list are drawn onto it. This composite bitmap is then displayed in the `OnPaint()` method.

This approach has several advantages, most importantly, it's possible to erase a particular shape instance from the final image. The shape has to be deleted from the shape list first, then the whole component is redrawn in the `OnPaint()` method. The result is the same image as before, but without the shape instance.

Because it's possible to control redrawing of the component, it's also possible to erase several shapes in a batch and when all operations are done, redraw the final image. This way, erasing several shape instances is made a very fast operation, because the whole image doesn't need to be recreated every time a shape is erased.

Another improvement which makes the component even faster is the so called *fast mode*. In this mode, an external entity is allowed to draw to the component's graphics directly, instead of storing the drawings to the component's background bitmap. Afterwards, when the fast mode is not necessary, the final background image should be stored to the component's background bitmap by calling an appropriate method, otherwise all changes would be lost.

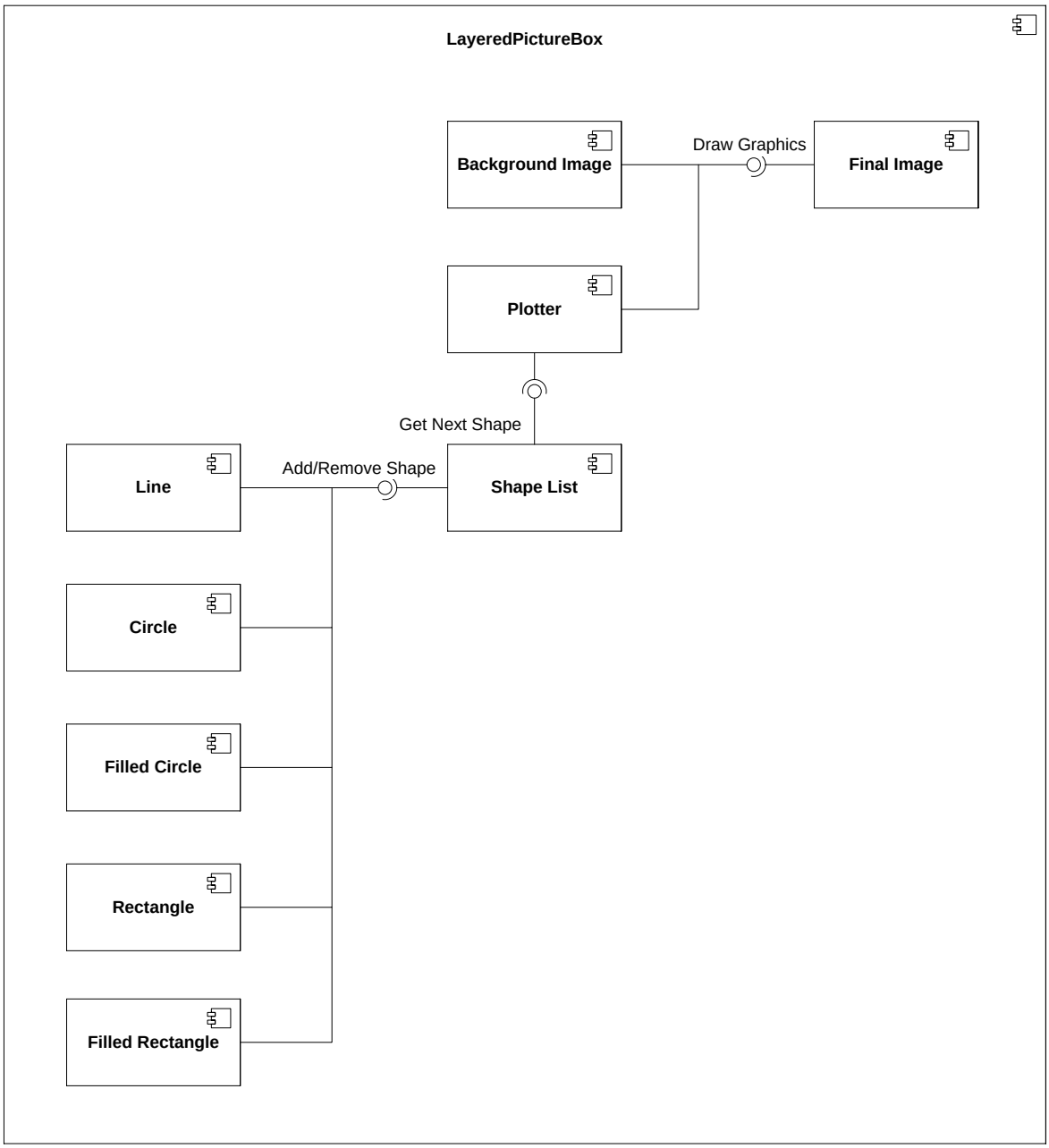


Figure 5.5: The `LayeredPictureBox` basic principle

5.3.7 LayoutPanel

`LayoutPanel` is an extension of the .NET's `System.Windows.Forms.Panel` class. It adds a few simple layout management features, as the .NET Compact Framework doesn't contain any layout manager at all. The class contains a *flow* layout manager and a design without a layout manager (*none*).

A flow layout manager arranges components in a directional flow, much like lines of text in a paragraph. It arranges components horizontally until no more components fit on the same line, then it places them on another line.[9]

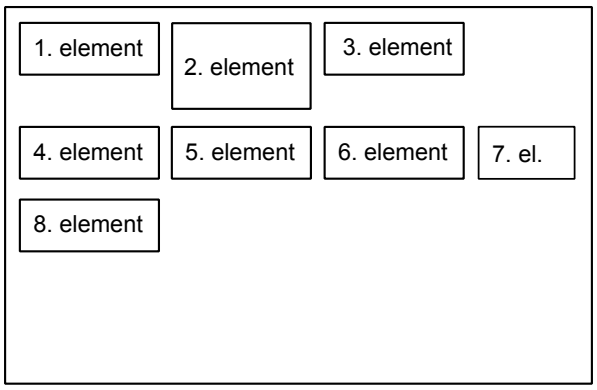


Figure 5.6: Flow layout

5.3.8 NumericUpDown

`NumericUpDown` is a replacement of the .NET's `System.Windows.Forms.NumericUpDown` class. It was build from scratch and does not inherit anything from the .NET class. Basically it does the same, but also adds a possibility of changing it's appearance. Except the component's colors, it's possible also to set it's rounding and padding width.

Figure 5.7: A `NumericUpDown`

5.3.9 Supplementary Classes

The supplementary classes in the BetterControls package are a collection of both classes inheriting from the .NET's `System.Windows.Forms.Control` class and regular general purpose classes. The classes inheriting from the `Control` class are never used alone, but embedded inside some other components.

5.3.9.1 ContextMenu

`ContextMenu` menu is used in the `ComboBox` (Section 5.3.5) class as one of it's crucial components. It's similar to the .NET's `ContextMenu`, except that it's able to display also other item types than strings, and it's appearance is customizable. In the `ComboBox` class, the `ContextMenu` instance is added as a child of the `ComboBox`'s parent, so that it can be displayed next to the `ComboBox` instance component. The `ComboBox` also listens to every mouse event in the application, and if such event occurs, the `ContextMenu` instance is closed.

Figure 5.8: A `ContextMenu`

5.3.9.2 LabelVerticallyCentered

`LabelVerticallyCentered` is, as it's name suggests, a class similar to the the .NET's class `Label`, except that the displayed text is vertically centered. This is useful in bars, or explanatory labels attached to some other GUI component, and it's in fact used almost everywhere a text is displayed in the application, most frequently in the Settings panel (Section 4.8).

5.3.9.3 GDI

`GDI` is a class which is an interface between the `coredll.dll` library and the BetterControl's `C#` classes. Here all necessary GDI functions are imported, and moreover it contains also some supplementary methods, which should simplify use of the GDI functions by the rest of the `C#` code.

5.4 PocketMeter

In this section some of the application's user interface most interesting algorithms are described. These algorithms were implemented entirely in `C#` .

5.4.1 State Machine

The state machine of the application is represented by an instance of the class `Lib.Fcn.StateMachine` and contains 15 states in total:

State	Description
`START`	The application is started, and no image has been loaded or taken
`NONE`	The application is showing the global screen (Section 4.2), an image has been loaded or taken
`CAMERA`	The user is taking a new picture by the smartphone camera
`RECOGNITION`	The application is in manual recognition mode, where the user is asked to mark the known object in the current image
`DISTANCE_MEAS`	The user is enabled to input new length measurement vertices
`DISTANCE_ERASER`	The user is enabled to delete existing length measurement vertices
`ANGLE_MEAS`	The user is enabled to input new angle measurement vertices
`ANGLE_ERASER`	The user is enabled to delete existing angle measurement vertices
`AREA_MEAS`	The user is enabled to input new area measurement vertices
`AREA_ERASER`	The user is enabled to delete existing area measurement vertices
`MOVE`	The user is moving the image by dragging it
`MOVENAV`	The user is moving the image by moving the rectangle in the navigation box 4.4
`ZOOM`	The application is zooming in or out
`OPEN_SAVE`	The user is enabled to open a save a measurement
`SETTINGS`	The application is showing it's Settings panel

Table 5.1: The application's states

There are many possibilities and many causes of how transitions between the states may occur, therefore the figure 5.9 shows just a simplified state transition graph.

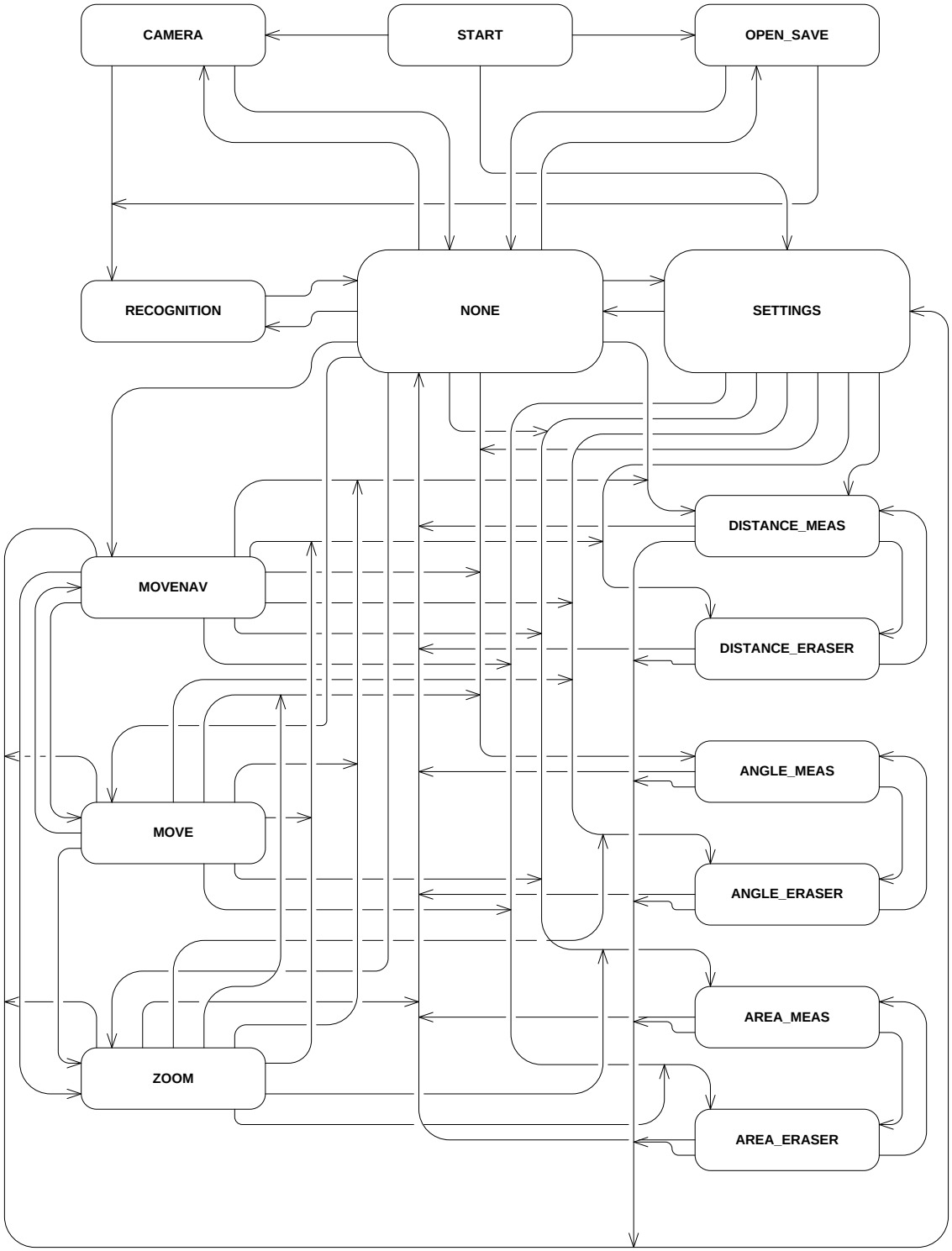


Figure 5.9: The application's state transitions

5.4.2 Stylus Interactions

If a user touches the screen in any kind of measurement mode, the application has to determine whether the user want to input a new vertex , delete or move a one already in the measurement, or move the image. Around every point there is an invisible circle, which determines, whether the user intended to pick some vertex, and whether he/she intended to move the vertex or move the image. At the moment of touch, the program determines, whether there is an existing measurement vertex inside the circle. If there is, based on the settings it erases the vertex immediately, or prepares to move it, in that case wherever the smartphone's stylus *goes*, there the vertex is moved, until the stylus leaves the screen. If there isn't any vertex in the circle at the first moment, and the application is not in any of it's erase modes, a new point may be placed at that location, or the whole image may be moved. This is determined based on whether the stylus leaves the circle before it leaves the screen. If it does, the image is moved. If it doesn't, a new vertex is created at the location at the moment when the stylus leaves the screen. Radius of the circle area is configurable in the application's Settings panel (Section 4.8).

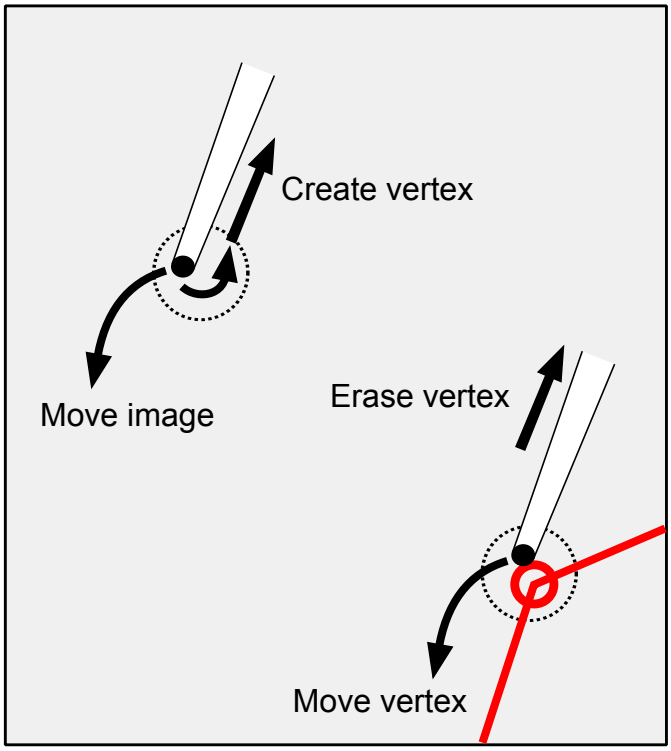


Figure 5.10: Possible stylus interactions

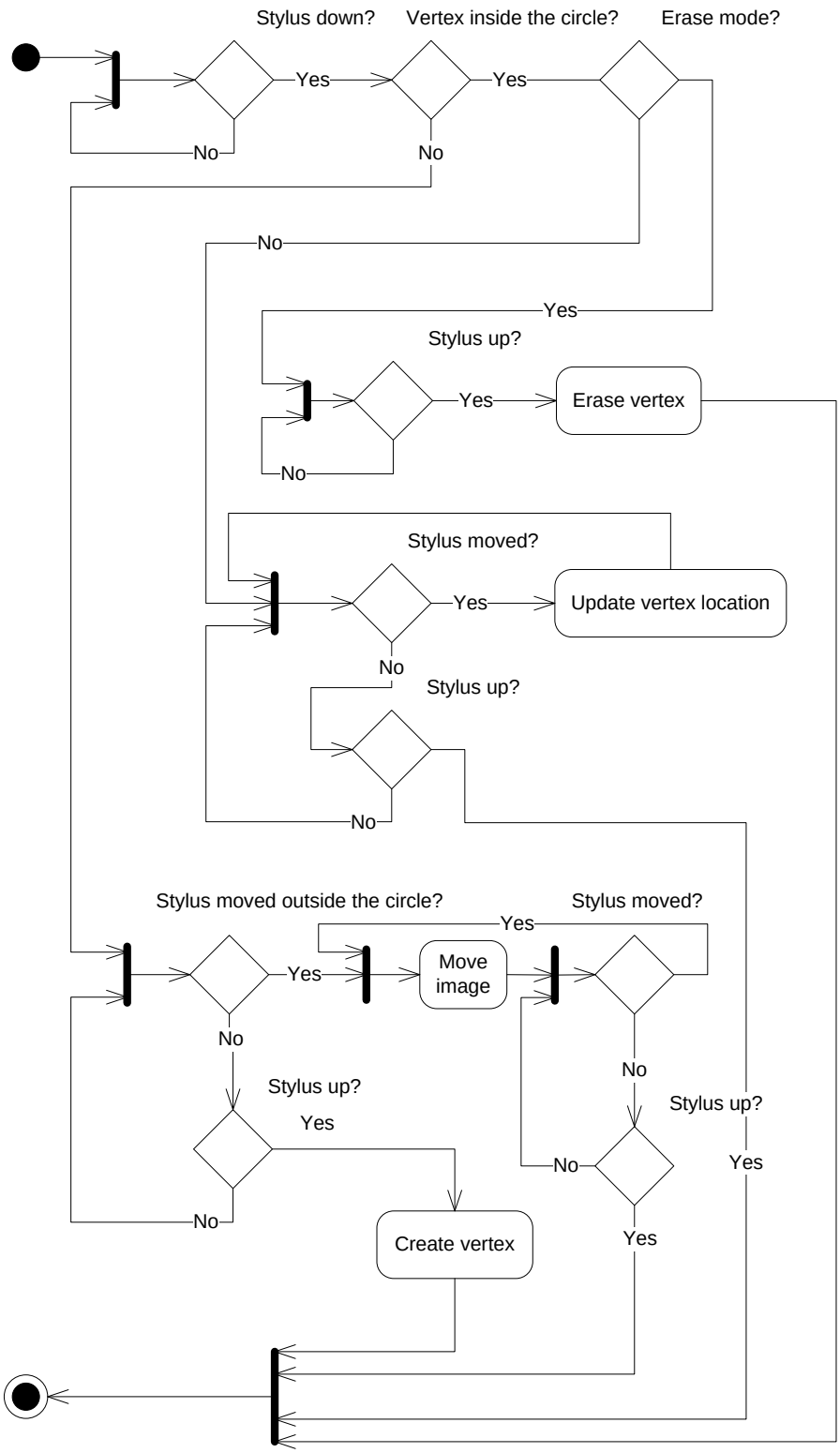


Figure 5.11: Stylus interactions algorithm

5.4.2.1 Length Measurement

When the application is in the state of measuring length, every new vertex is added as a continuation of the measurement polyline, i.e. at the end of the line. If an existing vertex is erased, a new line is created between it's neighboring points. The same applies for both the vertex and the freehand input.

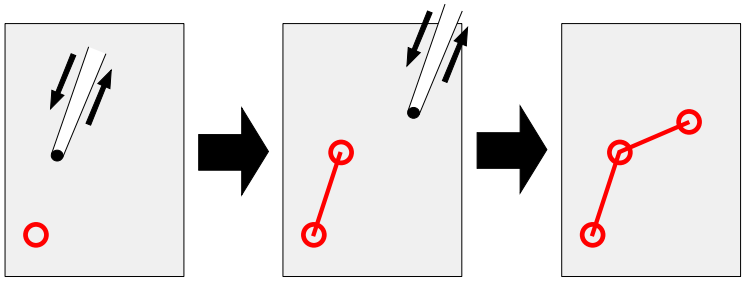


Figure 5.12: Length measurement - adding a new vertex

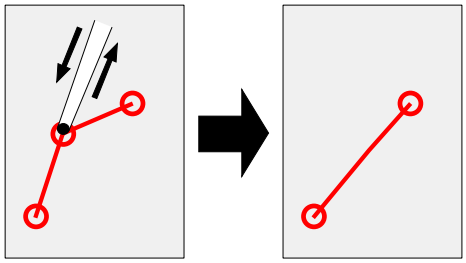


Figure 5.13: Length measurement - erasing an existing vertex

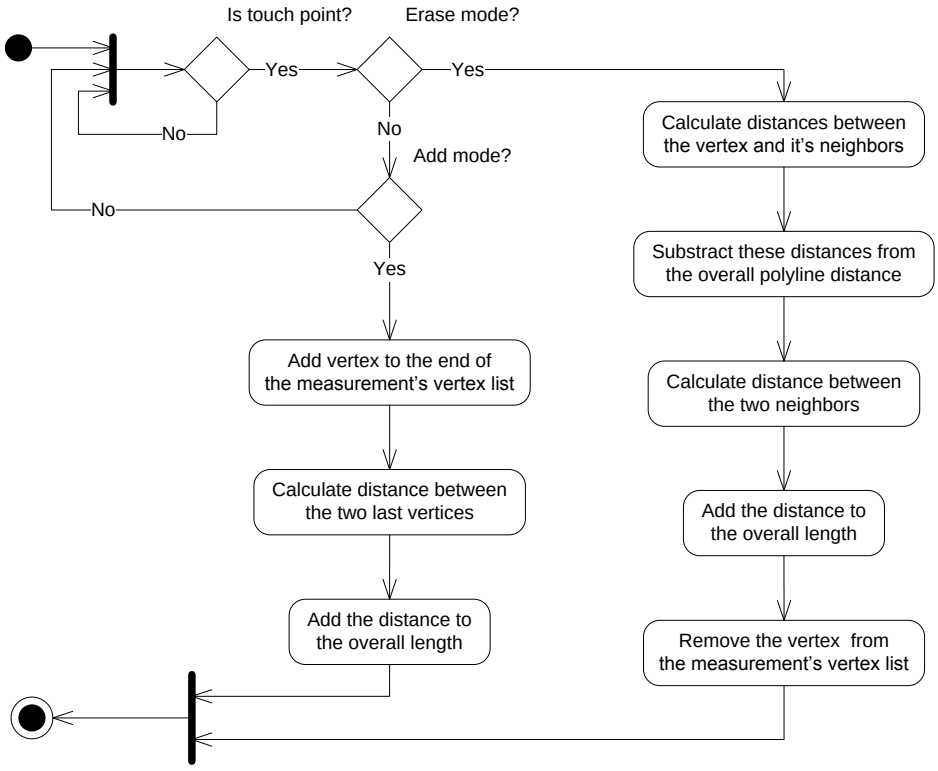


Figure 5.14: Length measurement - vertex operations

5.4.2.2 Angle Measurement

In angle measurements, 3 points can be added at most. If the number of points is lower than 3, the angle is not computed, therefore exactly 3 points are needed. If there aren't 3 vertices in the measurement, every new vertex is added to the measurement as a new point. If there are 3 vertices, every new point is added as the last vertex and the previous last vertex is erased. This means there may be a new angle, therefore it's value is updated.

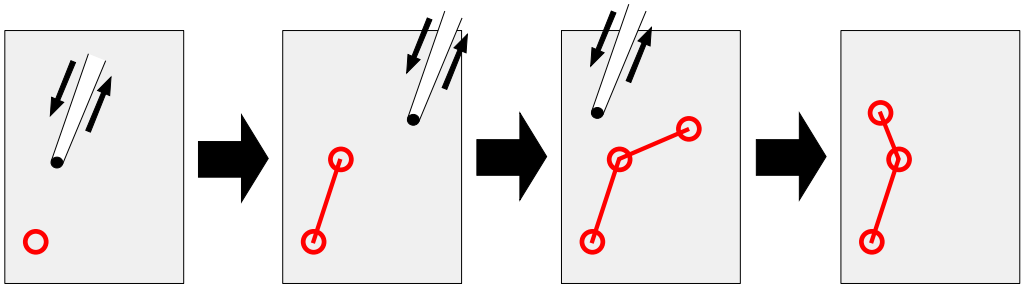


Figure 5.15: Angle measurement - adding a new vertex

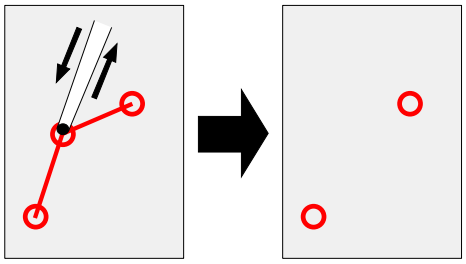


Figure 5.16: Angle measurement - erasing an existing vertex

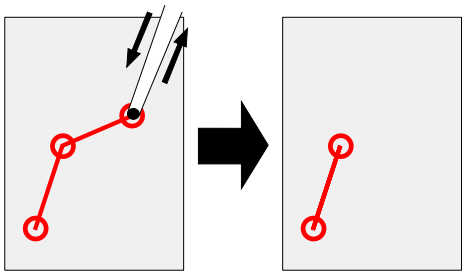


Figure 5.17: Angle measurement - erasing an existing vertex

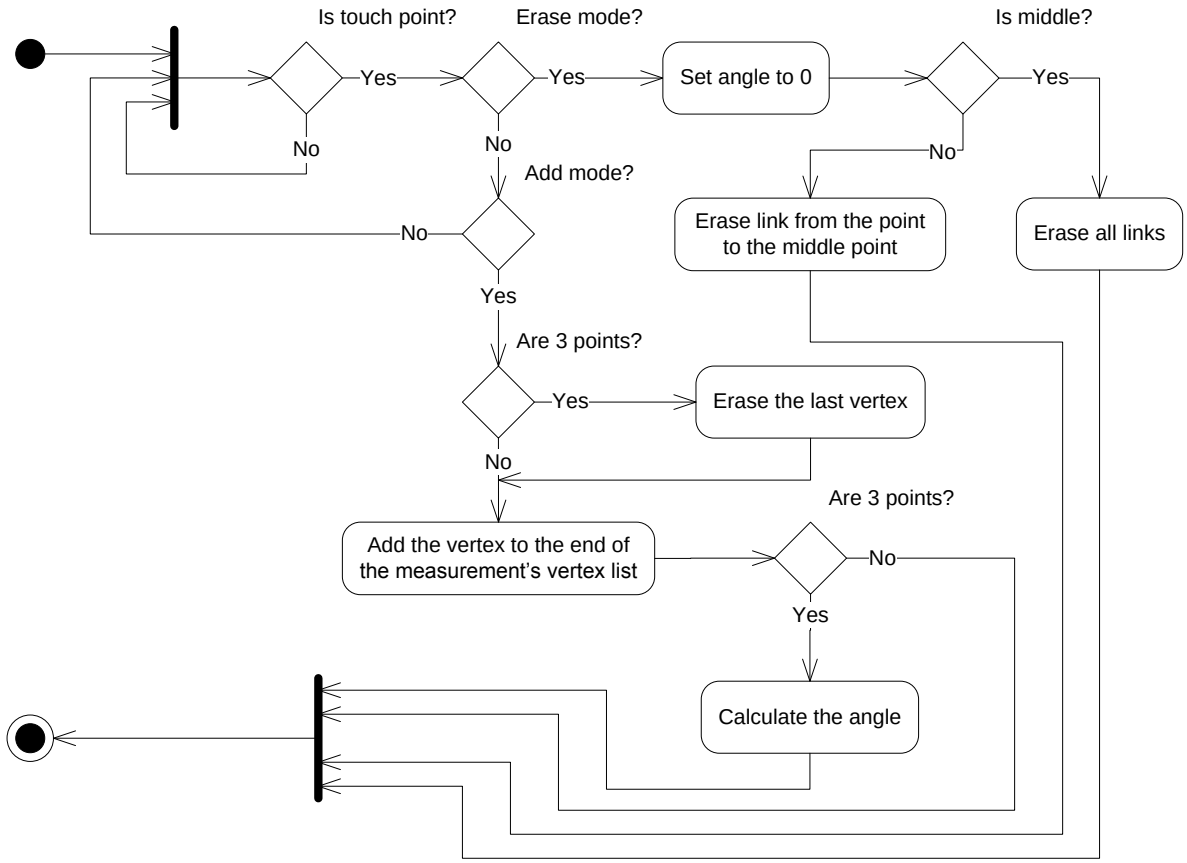


Figure 5.18: Angle measurement - vertex operations

5.4.2.3 Area Measurement

When the application is in the state of measuring area, the first two new vertices are added to the measurement as 2 new points immediately, however at least 3 points are necessary for an area to exist. Every further point therefore causes the area value to be updated. The following statements hold for both the vertex and the freehand input method, as the freehand line is in fact nothing more than a polyline.

Because an area is a closed structure, there isn't any end or start point, therefore a new point has to be added somewhere between the existing points and in fact *break* the structure. The edge which has to be *broken* is determined by *search for a closest line segment to a point* algorithm, where the point is the new vertex. Moreover, these points may be added only in case the edges of the final shape don't intersect each other, therefore an additional check is performed. If the new shape breaks this condition, the new vertex is not accepted and the shape won't change. If no two edges intersect each other, the new vertex is accepted and the area value is updated. The same rule is applied also when moving a vertex. If it's broken, the vertex is not moved.

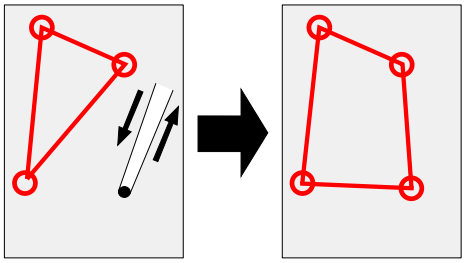


Figure 5.19: Area measurement - adding a new vertex

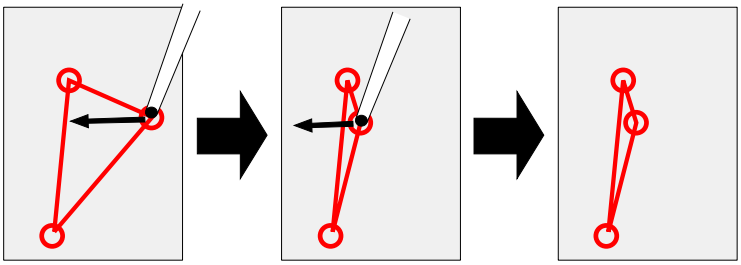


Figure 5.20: Area measurement - moving an existing vertex

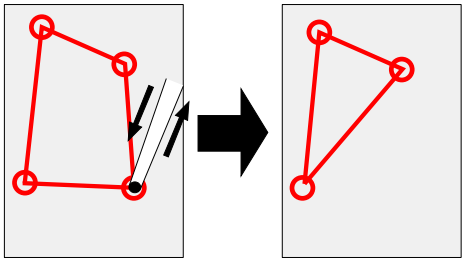


Figure 5.21: Area measurement - erasing an existing vertex

The point-to-edge distance is calculated as a point-to-line-segment distance by the use of these formulas:

$$u = \frac{(x_3 - x_1)(x_2 - x_1) + (y_3 - y_1)(y_2 - y_1)}{((x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{5.1}$$

$$d = \begin{cases} \sqrt{(x_3 - x_1)^2 + (y_3 - y_1)^2} & \text{if } u < 0 \\ \sqrt{(x_3 - y_2)^2 + (y_3 - y_2)^2} & \text{if } u > 1 \\ \sqrt{(x_3 - (x_1 + u(x_2 - x_1)))^2 + (y_3 - (y_1 + u(y_2 - y_1))^2} & \text{if } 0 \leq u \leq 1 \end{cases}, \tag{5.2}$$

where $P_1 = [x_1; y_1]$ and $P_2 = [x_2; y_2]$ are vertices of an edge, $P_3 = [x_3; y_3]$ is the third (new) point which does not lay on the same line as the vertices P_1 and P_2, and d is the distance of the point P_3 from the line segment between vertices vertices P_1 and P_2.

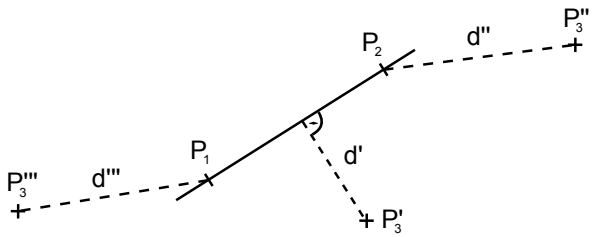


Figure 5.22: 3 variants of the point-to-line-segment distance

The area value is computed as an area of a non-intersecting (simple) polygon. The formula for this was first time described by A. M. Lopshits in 1963 [11]:

$$A = \frac{1}{2}(a_1[a_2 \sin(\theta_1) + a_3 \sin(\theta_1 + \theta_2) + \cdots + a_{n-1} \sin(\theta_1 + \theta_2 + \cdots + \theta_{n-2})] \tag{5.3}$$

$$+ a_2[a_3 \sin(\theta_2) + a_4 \sin(\theta_2 + \theta_3) + \cdots + a_{n-1} \sin(\theta_2 + \cdots + \theta_{n-2})] \tag{5.4}$$

$$+ \cdots + a_{n-2}[a_{n-1} \sin(\theta_{n-2})]), \tag{5.5}$$

where a_1,a_2, ..., a_n are the lengths of the sides, and $\theta_1, \theta_2, \ldots, \theta_n$ are the exterior angles of the polygon [10].

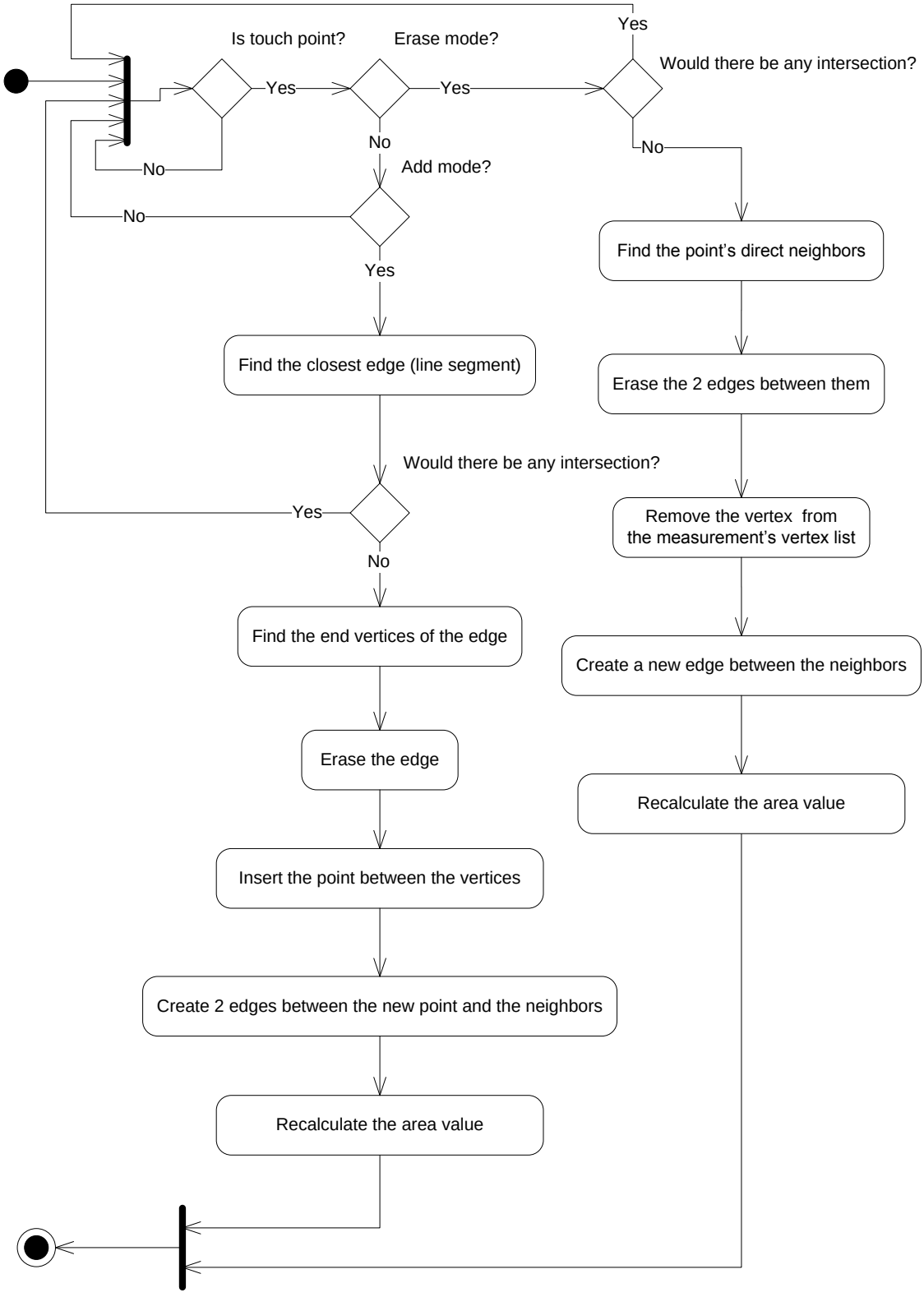


Figure 5.23: Area measurement - vertex operations

5.4.3 The .pmm File Format

PocketMeter Measurement (.pmm) files are used for storing whole measurements. Their format is XML, therefore it's possible to view and edit them in any text editor, or a specialized XML editor. An accompanying image (photo) file is necessary.

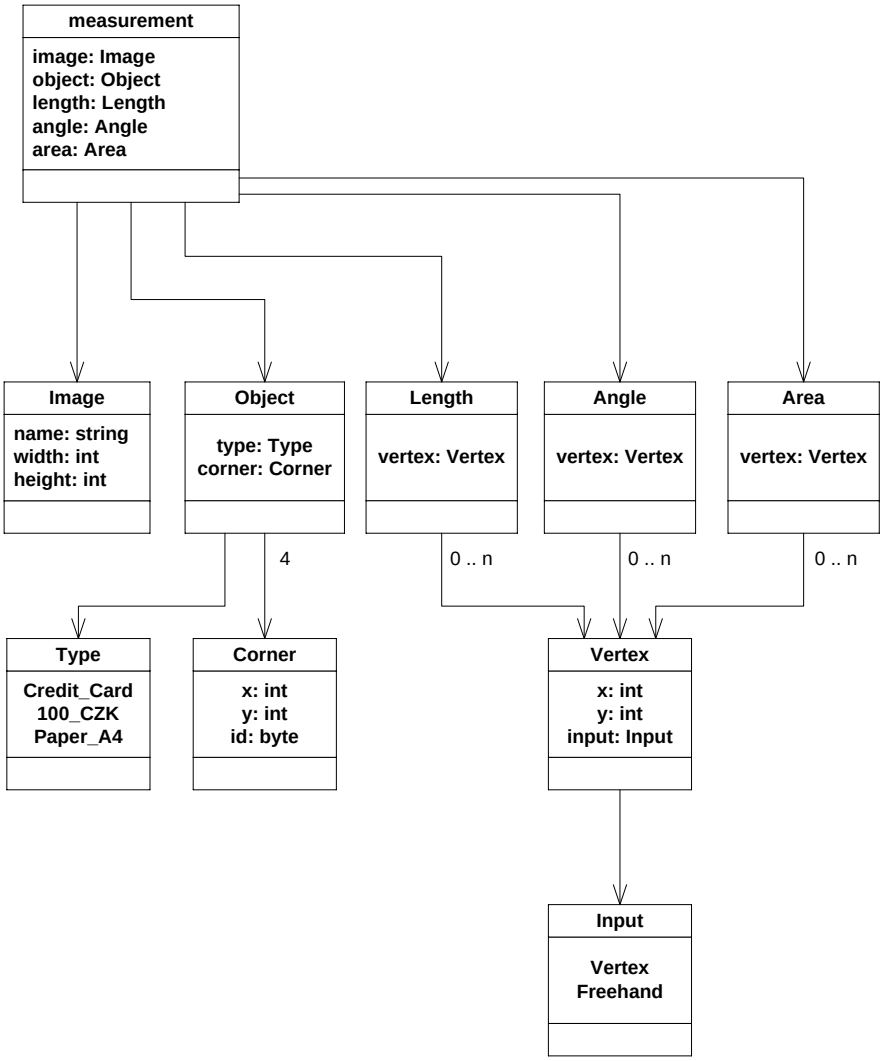


Figure 5.24: The .pmm file XML structure

5.4.4 Artwork

The application's appearance was an important part of the work, as one of the initial requirements was to make the program look more appealing.

5.4.4.1 Icons

The overall design was created by me, but the bases for button icons were created by Danny Allen [2], whom I would like to thank here. His icon set was released under the LGPL license, and the changed icon set is either distributed with the program, or available on request, as the license orders me to do.

5.4.4.2 Theme

Except the icons, almost every GUI component from the BetterControls subproject can be configured. The application's component appearances are defined in a single XML file called `pocketmeter.theme`, however this file is not accessible for ordinary users, because it's embedded in the application. It's accessible just for the application's developers.

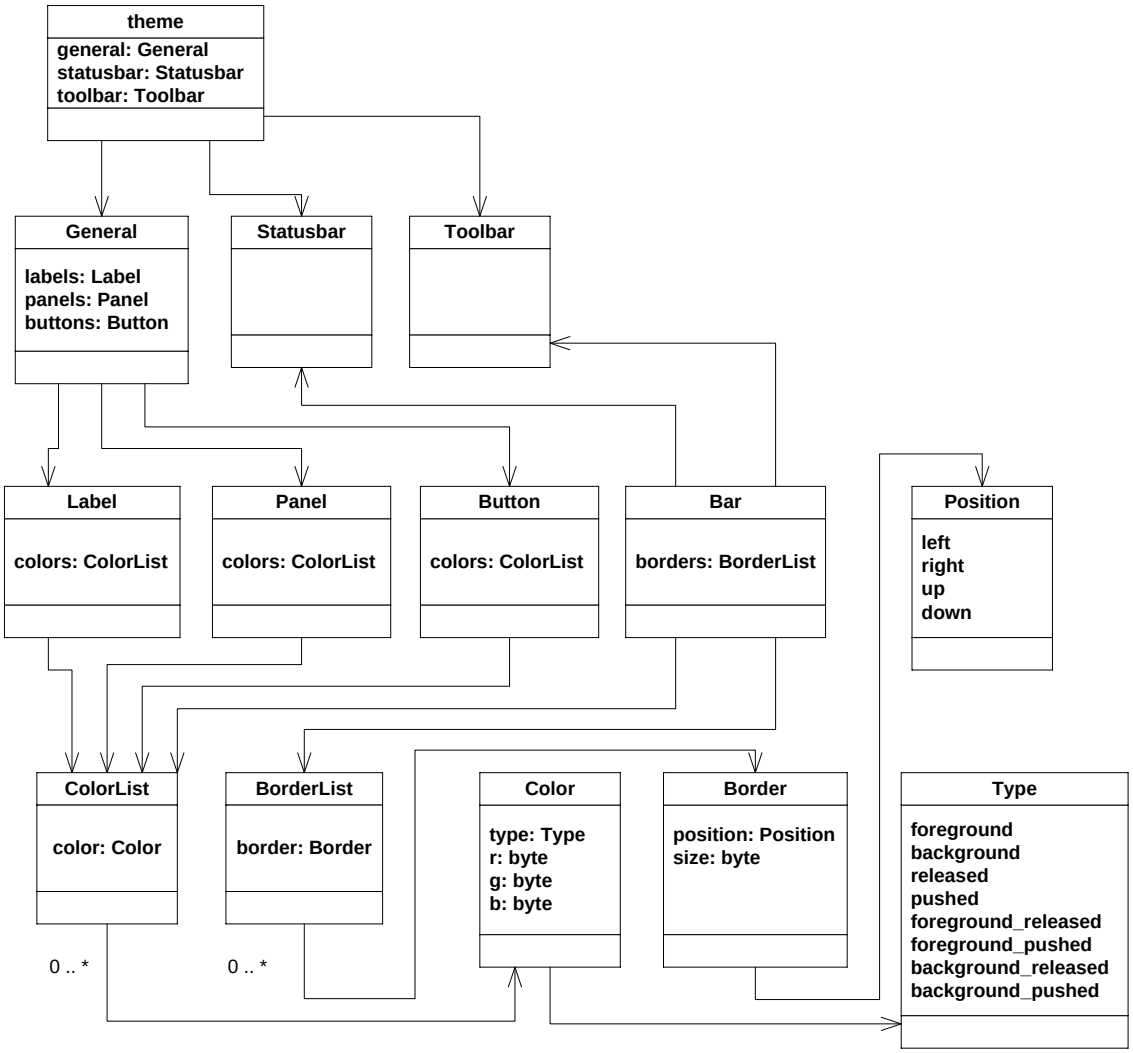


Figure 5.25: The Theme XML structure

Chapter 6

Test Results

There are several points of view of how to test the application. One point of view is a code oriented point of view, where the application's runtime and amount of allocated memory are tested. Another point of view it a function based performance point of view. Here the application is tested for it's measurement accuracy. Results of these tests can be found in the David Stach's thesis ([15]). The last point of view considered in this thesis is the everyday life application performance.

6.1 Hardware Requirements

I've tried to run the application on several devices (tables 6.1, 6.2 and 6.3), and on all of them I was successful, although sometimes it was necessary to quit some background processes, because of the lack of free memory. Speed of run of the application was never a problem.

Processor	Qualcomm® MSM 7201A™ 528 MHz
ROM	512 MB
RAM	288 MB
Display	3.8-inch TFT-LCD touch-sensitive screen with 480 x 800 WVGA
Camera	Main camera: 5 megapixel color camera with auto focus
Operating System	Windows Mobile® 6.1 Professional

Table 6.1: HTC Touch HD - Technical details

Processor	Qualcomm® MSM 7201A™ 528 MHz
ROM	256 MB
RAM	192 MB
Display	2.8-inch TFT-LCD flat touch-sensitive screen with VGA res.
Camera	Main camera: 3.2 megapixel color camera with auto focus
Operating System	Windows Mobile® 6.1 Professional

Table 6.2: HTC Touch Diamond - Technical details

Processor	Texas Instruments® OMAP™ 850 200 MHz processor
ROM	128 MB
RAM	64 MB
Display	TFT resistive touchscreen, 65K colors, 240 x 320 pixels, 2.8 inches
Camera	2 megapixel color camera
Operating System	Windows Mobile® 5.0 PocketPC

Table 6.3: HTC P4350 - Technical details

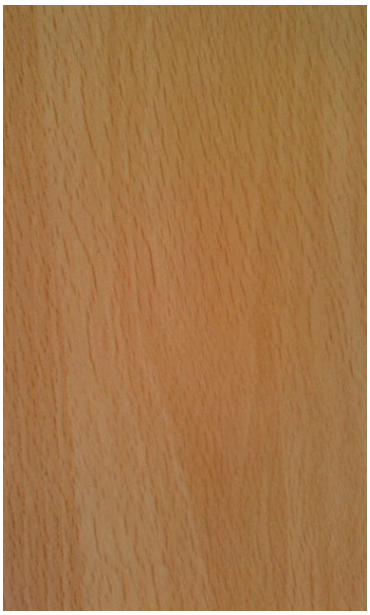
Later on, I executed the 7 following test cases to get somehow more exact results. Here I tested the peak hardware requirements. The test is split to two sections. The first section focuses on the native code performance, the second part focuses on the managed code performance. In the first part, peak amount of allocated bytes and time of recognition were tested. In the second part time of new vertex inclusion is tested for area measurement, as it's the measurement were the most computation is done upon new vertex creation. The test device was a HTC Touch HD (Table 6.1). The results can be seen in table 6.4.

6.1.1 Test Case 1-6

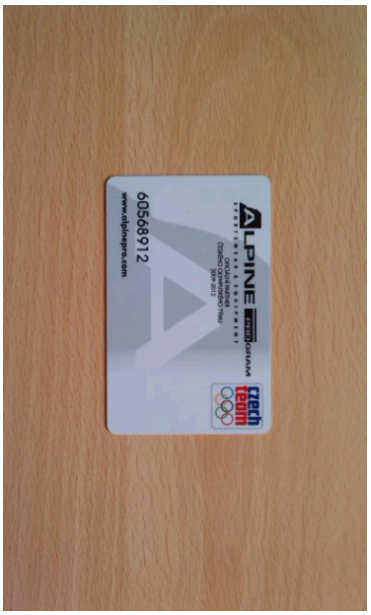
1. Start the application
2. Set automatic recognition
3. Take photo of a scene (Figure 6.1a, 6.1b, 6.1c, 6.1d, 6.1e, 6.1f)
4. Let the program recognize the known object
5. Quit the application

Scene	6.1a	6.1b	6.1c	6.1d	6.1e	6.1f
Peak allocated space [B]	1504281	1534372	2451268	1573688	1582996	1580912
Recognition time [ms]	5012 (*)	1034	2120	4302	3209	2346

Table 6.4: Native code performance, (*) - manual recognition

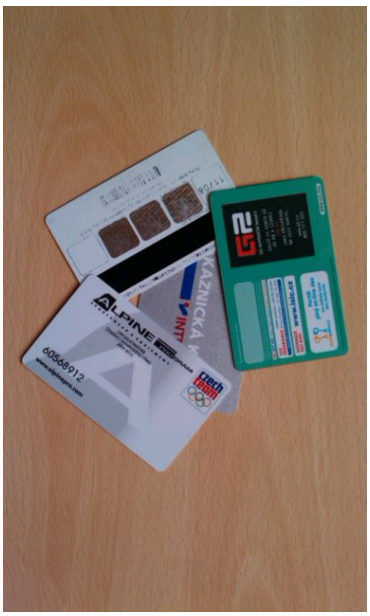


(a) No object



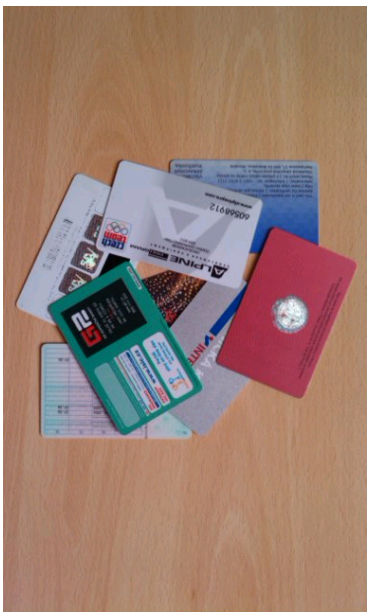
(b) 1 object

(c) 4 ordered objects



(d) 4 unordered objects

(e) 8 ordered objects



(f) 8 unordered objects

Figure 6.1: Native code test scenes

6.1.2 Test Case 7

1. Start the application
2. Set automatic recognition
3. Take a photo
4. Let the program recognize the known object
5. Switch to area measurement
6. Input 50 vertices
7. Save the measurement
8. Quit the application

The values from this measurement were unexpectedly low, inclusion of the fifth vertex took less than 1 second.

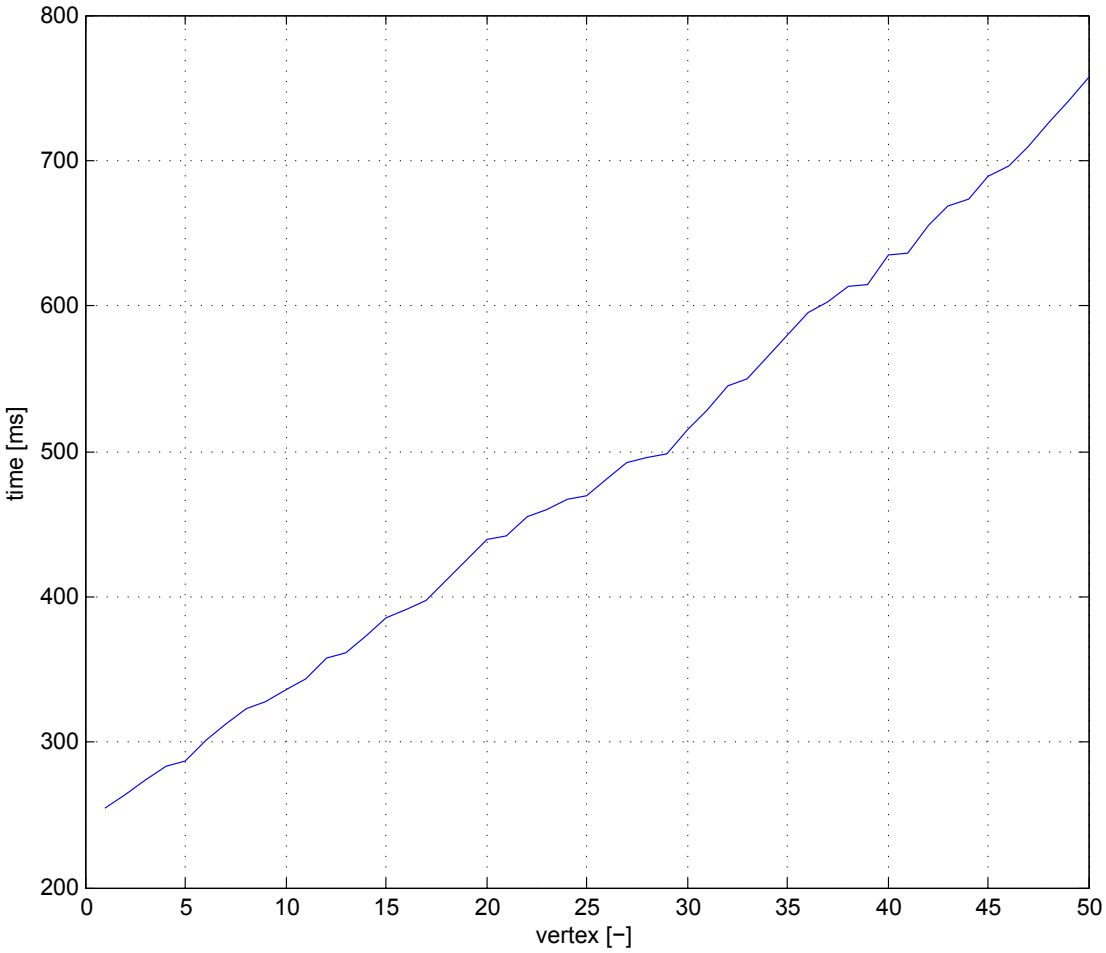


Figure 6.2: Time of inclusion vs. number of existing vertices

6.2 Everyday Life Measurements

This section is here to demonstrate use of the application for everyday life measurements. Three examples are presented, one, the measurement of a chimney contains in fact an another embedded measurement. Basically there's no need for embedded measurements, if dimensions of some object in the scene are known, but often an object which is known has inappropriate size to the unknown. The second measurement demonstrates universal use of object dimensions knowledge, when the results of the embedded measurement in the first measurement are employed to perform measurements in a different scene. The third measurement is an indoor measurement of furniture. In these measurements I assume the measured and the known objects are laying on the same plane.

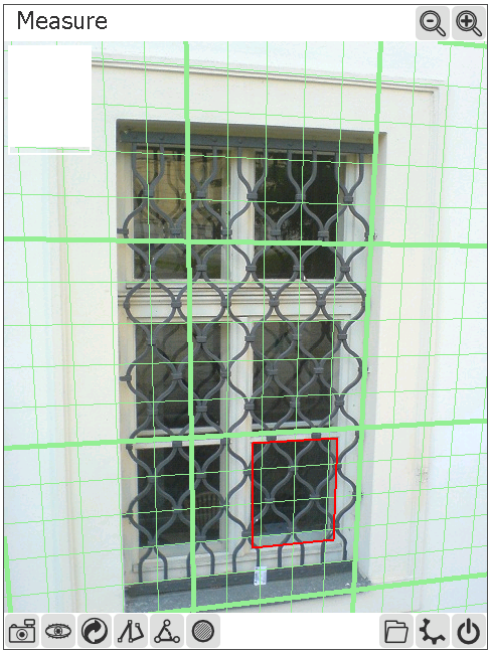
6.2.1 Chimney

In this section height a chimney is measured. An embedded measurement is performed first, as no dimensions of a object in the final scene are known.

6.2.1.1 The Embedded Measurement

Here dimensions of a window are measured. A known object here is a part of the window.

(a) The recognized object



(b) Grid displayed

(c) Window inner width

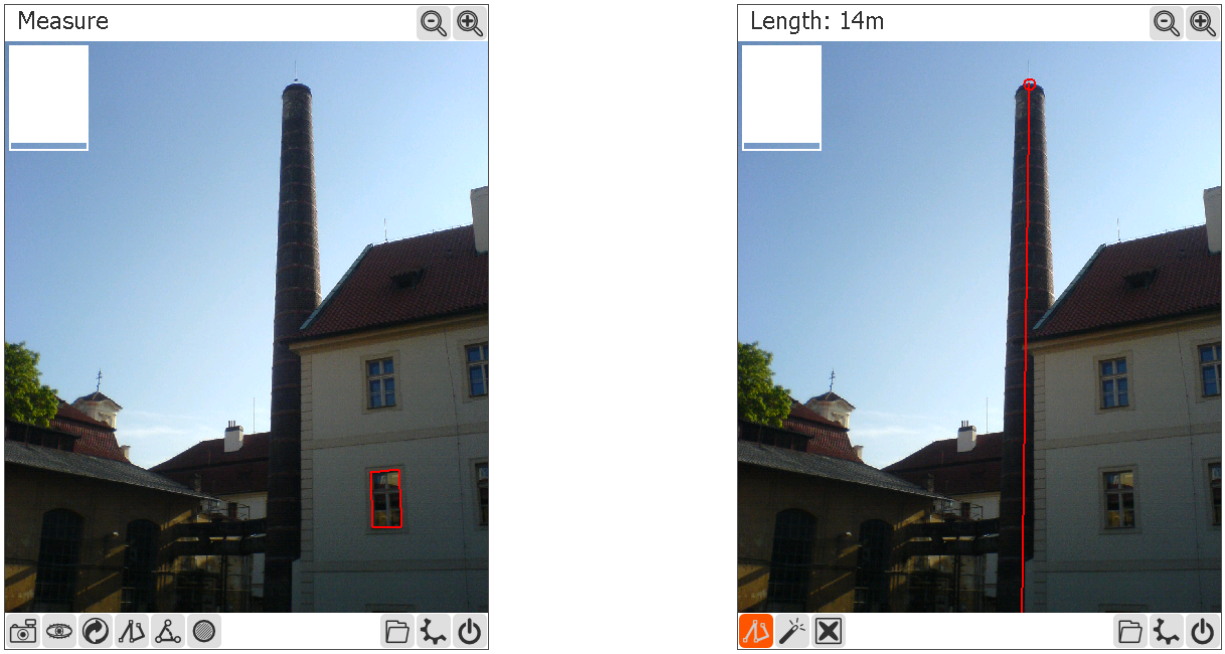
(d) Window inner height

Figure 6.3: The embedded measurement - the window scene

The obtained inner dimensions of a window are: width = 74cm, height=123cm.

6.2.1.2 The Main Measurement

Next height of the chimney is measured. Because the chimney and the window are not precisely on the same plane, to get some approximate results, an assumption that they are have to be accepted. Due to this issue, the result of this measurement contains some minor additional error.



(a) A recognized window

(b) The chimney height

Figure 6.4: The chimney scene

The obtained height of the chimney is: 14m.

6.2.2 Tree

This measurement employs results of the first measurement to measure geometric quantities in another scene. Here approximate height and width of a tree is measured. The measurement shows than the previous measurement the importance of the measured and the known object laying on the same plane, as the measured height of the tree is not the same as one would expect. Clearly the tree is not as high as the chimney, but the measurement result is the same. This happens because the tree is not on the same plane as the window, it's in front of it. Despite this inaccuracy the measurement still provides good enough estimate of the tree height for most every day life problems.

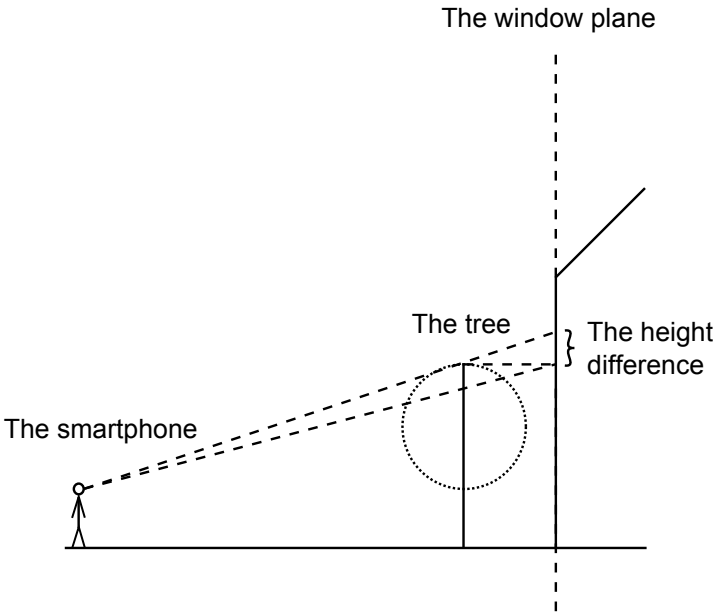
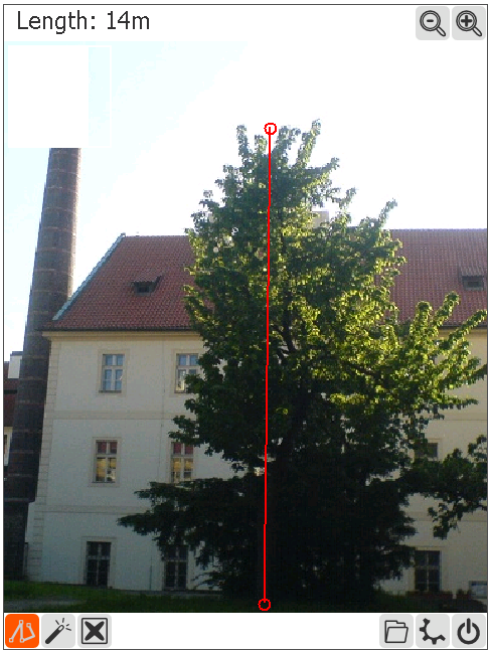


Figure 6.5: The tree scene geometry

(a) The recognized window



(b) The tree height

Figure 6.6: The tree scene

The obtained approximate height of the tree is: 14m.

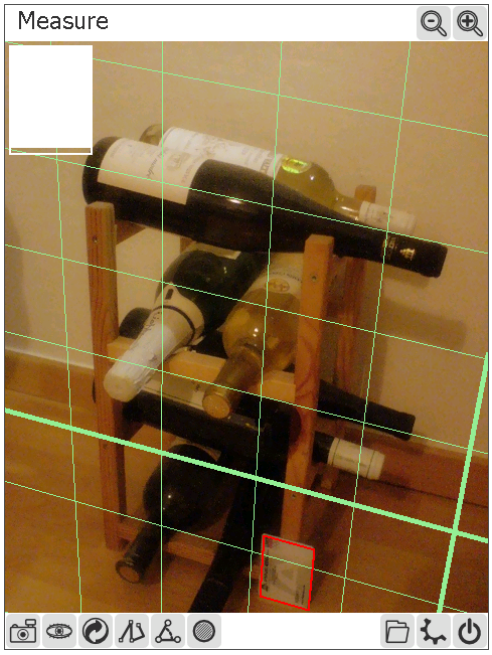
6.2.3 Wine Bottle Holder

In this measurement an indoor scene is processed. Indoor use of the application is expected to be the most frequent. In this scene I wanted to obtain dimensions of a wine bottle holder.

(a) The holder width

(b) The holder height

(c) The recognized object



(d) Grid displayed

Figure 6.7: The wine bottle holder scene

Chapter 7

Conclusion

The aim of this master's thesis was to create an application for geometric quantities measurements running on a smartphone and employing algorithms and techniques of Computer Vision. The thesis is a continuation of a work of David Stach ([15]), who developed an image recognition library for his own master's thesis. My work then consisted mainly of designing and programming the user interface, which in David's work was just a very simple one. Because the image recognition library does just image recognition, beside the user interface, I also implemented the algorithms of particular measurements, namely the measurement of a polyline length, measurement of angle and measurement of area. Because there was no time left for debugging and enhancing the image recognition algorithms, the program in some cases fails to recognize known objects. Therefore I added a possibility for the user to mark the known object by hand. Users are also able to define known object dimensions, therefore the recognition is not restricted just to the predefined templates. Concerning the recognition precision and reliability, despite the lack of time, I first attempted to enhance the library, and then create my own from scratch, but in the end I abandoned it. In my library I employed the apparently widely known open source library OpenCV originally from Intel [**?**], which enabled me to make a very fast progress, and which, if used from the beginning, would speed up the work significantly and would enable much broader possibilities of development.

An issue in which I was concerned probably the most was the application speed performance and memory usage. However, the tests (Chapter 6) showed the application performs quite well (Sections 6.1.1 and 6.1.2), which was also confirmed by usability tests on two subjects.

Personally, I expected to achieve more with this thesis, but during the development I encountered several serious problems which caused the time plan to slip.

The majority of the problems were related to the .NET Compact Framework itself. Because the framework is meant for mobile devices with very constrained resources, such as the amount of operational memory, or the computational power, it's relatively restricted compared to the desktop .NET version. Many necessary methods, and even whole classes or packages are missing. As a result I spent a lot of time figuring out workarounds for these problems and implementing classes which are present in the desktop version of the .NET Framework, or any other framework, but not in the .NET Compact Framework. This type of problems caused the BetterControls subproject to be created.

Another issue with the .NET Framework as a whole is poor speed performance of it's managed code compared to native code. Although this property was expected, it was not expected to delay the project significantly. Unfortunately performance of the

platform showed up to be insufficient for the majority of graphic functions needed in the application. Therefore as with the first .NET issue, I ended up inventing workarounds for the problems, which in some cases led to use of unmanaged native Windows API functions.

In the process of debugging I encountered another problem with the .NET Compact Framework specific to my case. At the time of the development just the Microsoft Visual Studio 2008 Professional and the Microsoft Visual Studio 2010 Ultimate were available to me, neither of which strangely contains a usable .NET Compact Framework profiler. The 2008 Professional version doesn't contain any profiler at all, which is quite unexpected regarding to it's suffix. The 2010 Ultimate version contains a profiler, but a one which, again unexpectedly, doesn't support the .NET Compact Framework. These pitfalls caused additional development delay.

I estimate that solving these problems with the .NET Compact Framework took me about 60-70% of the development time, therefore use of another framework and programming language was worth considering. Clearly in this work use of a virtualized environment did not bring many positives, use of native environment might bring higher speed of development. In the end I, as well as the image recognition library, attempted to switch to another GUI framework and another programming language. I chose the Qt framework from Nokia [**?**] and the `C++` programming language. Although the Qt framework was not initially intended for Windows Mobile, not even for mobile devices at all, the development was much faster and the interface was more responsive and enhanced than the .NET version. Moreover, thanks to the Qt, OpenCV and `C++` portability, it would be possible to compile and run the program on many platforms other than Windows Mobile, which is not the case of .NET version. Unfortunately, there was no time left to finish neither the Qt interface, nor the library, therefore I proceeded with the .NET version with the David Stach's library. Due to these reasons I have to conclude that the assumption about fast software development with `C#` and the .NET Framework given in Section 2.2 did not hold for this thesis.

These problems along with careful project planning however provide me with valuable experience which might help me to not to repeat them in the future.

Chapter 8

Future Work

As the work was delayed by the .NET issues, and as the field of Computer Vision is broad, there is a plenty of space for enhancements.

First, the user interface might be tested on usability and changed accordingly. If the application is transferred to another platform, e.g. the Apple iPhone, it can be changed radically, as new software and hardware possibilities might become available, e.g. most Windows Mobile devices nowadays don't have multi-touch screens, if such a component is available, zooming and moving images might be done in a completely new way. The user interface might become even faster, as currently some of it's operations (e.g. redrawing the image) are noticeable by the user. Also, the interface might be simplified by the use of gestures, or by a different overall architecture. Measurement results might be for example displayed together as depicted in Figure 8.1.

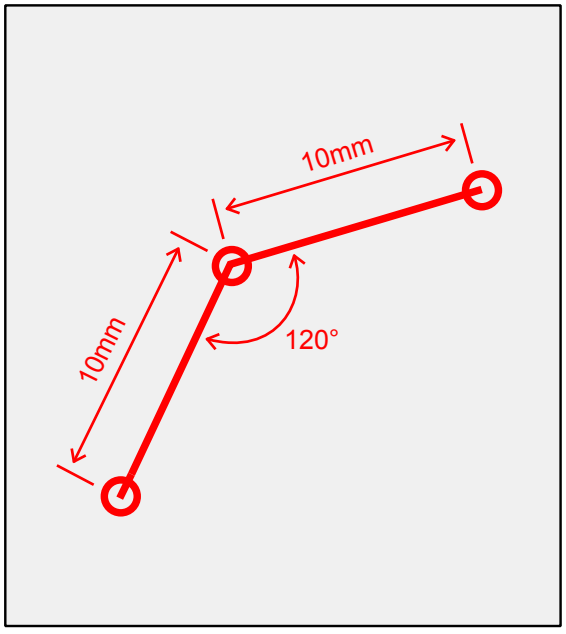


Figure 8.1: A proposed display of measurement results

The second part, where enhancements can be done, is the image recognition. Now, reference objects are sometimes not recognized precisely, or not found at all. Serious inaccuracies are caused by shadows and reflections. The recognition is based just on

edge detection and shape fitting, the *content* of the shape is completely ignored, and moreover the shape is constrained just to rectangular objects. Concerning the first issue, some probabilistic classification technique of defined image features might be considered, for example Bayesian networks or Neural networks. Both the object content recognition and the current algorithm's parameter settings might be enhanced by some optimization technique, e.g. evolutionary algorithms, or dynamic programming. These methods would then require a vast amount of annotated test images. Recognition of lines might be improved by the use of (Gaussian) image pyramid [5]. This was experimentally confirmed by the second computer vision library I made. Another possible enhancement of the application is the use of the 4 point algorithm 3.5 for camera pose estimation. With this information, distances of objects laying on the same plane as the reference object, from the smartphone can be estimated which would enable another new kinds of measurement.

Bibliography

[1] Mike Abramsky and Paul Treiber. Sizing the Global Smartphone Market, November 2008. RBC Capital Markets.

[2] Danny Allen. Monochrome icon set, 2010. URL `http://dannyallen.co.uk/downloads/icons/`.

[3] Andreas Ess, Alexander Neubeck, and Luc van Gool. Generalised linear pose estimation. In *British Machine Vision Conference (BMVC '07)*, September 2007.

[4] Brian Gammage et al. 2010 and Beyond: A New Balance, December 2009. Gartner.

[5] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, 2002.

[6] Sebastian Grembowietz. Algorithms for Augmented Reality: 3D Pose Estimation, November 2004. URL `http://home.in.tum.de/~grembowi/ar2004_05/3dPoseEstimation_presentation.pdf`. Technische Universität München.

[7] Matej Horváth. Počítačové Vidění a Virtuální Realita: DU-01, November 2009. Czech Technical University in Prague, Department of Cybernetics.

[8] William L. Hosch. Encyclopædia Brittanica: Smartphone, May 2010. URL `http://www.britannica.com/EBchecked/topic/1498102/smartphone`.

[9] Wikimedia Foundation Inc. Wikipedia: Layout manager, May 2010. URL `http://en.wikipedia.org/wiki/Layout_manager`.

[10] Wikimedia Foundation Inc. Wikipedia: Polygon, May 2010. URL `http://en.wikipedia.org/wiki/Polygon`.

[11] Abram Mironovich Lopshits. *Computation of Areas of Oriented Figures*. D C Heath and Company: Boston, MA, 1963.

[12] J. Gerry Purdy. 2010 Outlook & Forecast: Mobile & Wireless Communications, December 2009. Frost & Sullivan.

[13] Long Quan and Zhongdan Lan. Linear N-Point Camera Pose Determination. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1999.

[14] Milan Sonka, Václav Hlaváč, and Roger Boyle. *Image Processing, Analysis and Machine Vision*. CL-Engineering, 2007.

[15] David Stach. Mobilní telefon jako dálkoměr-hloubkoměr-pravítko. Master's thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2009.

[16] William J. Wolfe, Donald Mathis, Cheryl Weber Sklair, and Michael Magee. The Perspective View of Three Points. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1991.

[17] Paul Yao and David Durant. *Programming .NET Compact Framework 3.5*. Addison-Wesley, 2010.

[18] Lihong Zhi and Jianliang Tang. A Complete Linear 4-Point Algorithm for Camera Pose Determination. *MM Research Preprints*, 2002.

Appendix A

First Appendix

The first and only appendix to this work is a CD with the following content:

```
--|-- mt_matej_horvath.pdf    // the thesis
  |
  |-- src.zip                 // source codes
  |
  |-- bin.zip                 // executable application
  |
  |-- doc.zip                 // source code documentation
```