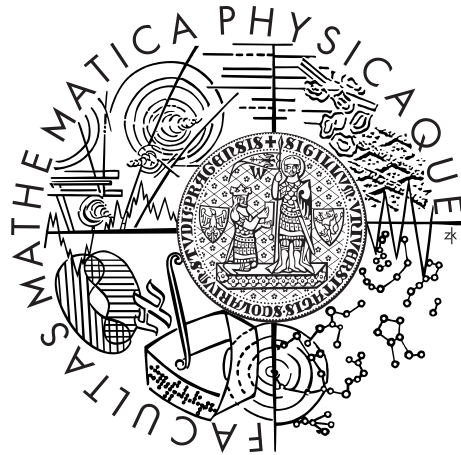


Charles University in Prague  
Faculty of Mathematics and Physics

## BACHELOR THESIS



Viktorie Vášová

## 3D Computer Vision on the Android Platform

Name of the department or institute

Supervisor of the bachelor thesis: Mgr. Lukáš Mach

Study programme: General Computer Science

Prague 2013

Dedication.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date .....

signature of the author

Název práce: 3D počítačové vidění pro platformu Android

Autor: Viktorie Vášová

Katedra: Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: Mgr. Lukáš Mach

Abstrakt:

Klíčová slova: 3d počítačové vidění, problém korespondence, platforma Android

Title: 3D Computer Vision on the Android Platform

Author: Viktorie Vášová

Department: Computer Science Institute of Charles University

Supervisor: Mgr. Lukáš Mach

Abstract:

Keywords: 3D computer vision, image correspondence, Android platform

# Contents

# Introduction

The task of reconstructing 3D information from multiple 2D photos of a real-world scene has attracted a lot of research in the last two decades and in the recent years in particular. A great number of algorithms have been proposed to solve problems in this area and several main approaches emerged. The applicability of these approaches depends mainly on what kind of input we intend to feed the algorithm (an unorganized set of photos, a video stream, a pair of stereoscopic images, etc.) and also on what kind of output we expect the algorithm to produce (polygonal model, a disparity map). As a result of this progress, various real-world applications for these algorithms have appeared – e.g., several camera trackers or products like Microsoft PhotoSynth.

Simultaneously, both the general availability and the computational power of smartphones have improved significantly. Mobile phones that employ the Linux-based Android software platform are currently very popular. (*UPDATE: uvest procentualni podil na trhu, pridat citaci.*) The Android market share increased to almost 69% in the year 2012 [?]. See Figure ?? to compare it with the market share of the other mobile operating systems on the market in the recent two years. A built-in camera and a relatively fast CPUs are very common in such mobile phones.

Operating System	2012 Market Share	2011 Market Share	Year over Year Change
Android	68.8%	49.2%	104.1%
iOS	18.8%	18.8%	46.0%
BlackBerry	4.5%	10.3%	-36.4%
Windows Phone	2.5%	1.8%	98.9%
Symbian	3.3%	16.5%	-70.7%
Others	2.1%	3.3%	-7.4%
Total	100.0%	100.0%	46.1%

Figure 1: Top five smartphone operating systems and their market share in 2011 and 2012.

The goal of this work is to explore ways how to connect these two phenomena. Our aim is to create an Android application that takes a set of photos using the phone's internal camera, applies a series of Computer Vision algorithms to reconstruct the depth information, and visualizes the result using 3D graphics. Due to the inherent ambiguity of the problem, it is inevitable that our approach will be limited to particular types of scenes, for example sets of photos of highly textured surfaces.

(*UPDATE: nasledujici vety by se mely primo odkazovat na sekce.*) The first part of this work analyses the problem, describes available software and gives an overview of programming libraries and languages that were used (see chapter ??). Secondly, we focus on the theoretical foundations of 3D reconstruction from image data (see chapter ?? and ??). Then we depict the basics of android development (chapter ??). The next section is devoted to the implementation

of our application. Finally, we evaluate and benchmark the resulting application (chapter ??). (*TODO: tady bychom určite chteli explicitneji napsat, ze vysledkem budou i nejake datasety, ale to se zvladne az to budeme mit napsane.*)

# 1. Overview

In this chapter, we give an overview of software solutions providing functionality related to our area, namely the analysis of depth information and 3D reconstruction from photos and videos. Section ?? discusses software packages currently available to end-users, while Section ?? describes software libraries implementing relevant Computer Vision algorithms.

## 1.1 Available Software Packages

### 1.1.1 Matchmoving Software

Matchmoving software in the film industry represents one of the earliest widely adopted commercial applications of algorithms that extract 3D information from 2D (video) imagery. In such a software, accurate 3D information about the scene is only a secondary product and the user is mainly interested in obtaining information about the position and orientation the camera had at the time of capture of the individual video frames. The knowledge of these parameters allows artists to add special effects and/or other synthetic elements to the video footage.

Although matchmoving (also called camera tracking) can be achieved using many different techniques, the prevailing method detects easily definable elements – such as corners – in a frame of the video and tracks their movement on the subsequent frames. The camera parameters are then calculated from the 2D movement of the tracks. In Computer Vision this approach is called *structure from motion*, since the structure of the scene (for example, the trajectory of the camera) is determined by the apparent movement of the tracks on the 2D frames. The 3D positions of the scene-points corresponding to the detected corners can also be calculated, giving the user a very rough point cloud reconstruction of the scene.

Examples of widely used matchmoving applications include 2d3's Boujou (*UPDATE: link.*)<sup>1</sup> and Autodesk Matchmover (*UPDATE: link.*)<sup>2</sup>. The open-source libmv project (*TODO: link.*) aims to add matchmoving capabilities to the Blender 3D modelling application (*UPDATE: link.*)<sup>3</sup>.

### 1.1.2 Microsoft PhotoSynth

Microsoft PhotoSynth, based on a research by Snavely, et al. (*UPDATE: citation.*) [?], has been publicly released in 2008. The software solution processes a set of unorganized pictures of a single scene and subsequently generates its 3D point cloud reconstruction. The main purpose of the reconstruction is to allow the user to navigate between the photos in a novel way, which respects the physical proximity of the cameras taking the individual photos. Perhaps most notably, the technology has been employed by the BBC and CNN. Recently, the possibil-

---

<sup>1</sup><http://www.2d3.com>

<sup>2</sup><http://www.autodesk.com>

<sup>3</sup><http://www.blender.org>



ity to generate 360° panoramas and to upload the input photos from a Windows 8 mobile phone has been added.

Microsoft PhotoSynth is a closed-source application with most of the computation running on Microsoft's servers. (*UPDATE: pridat jak photosynth postupuje, včetne odkazu na Snavelyho clanky.*) *Microsoft Photosynth extends the system called Photo Tourism developed by Noah Snavely and Steve Seitz [?] [?]. To obtain the accurate information about positions of cameras it uses transform invariant SIFT algorithm to detect interest points. For each pair of images the keypoints are matched using the approximate nearest neighbour package and by using RANSAC [?] the fundamental matrix is estimated.*

### 1.1.3 Autodesk 123D

Autodesk 123D is a bundle of several applications. One of them is 123D Catch, which creates a 3D model from a set of photos taken from different viewpoints. The software is compatible with other Autodesk 123D applications, making it a viable solution for 3D artists who want to include real-world objects in their scenes. The program is available for the Windows, Mac OSX, and iOS platforms. To achieve good results, it is necessary to follow detailed instructions when taking the photos. A failed reconstruction typically occurs when the photos are blurred, do not have solid background, or in the case of insufficient amount of photos.

## 1.2 Available Libraries

We now briefly introduce the main Computer Vision or Computer Graphics libraries that provide implementation of the algorithms necessary to build our software.

### 1.2.1 OpenCV

OpenCV (*UPDATE: link.*)<sup>4</sup> is a cross-platform library originally developed by Intel. It provides an implementation of several hundreds of Computer Vision related algorithms, including, e.g., camera calibration routines, image segmentation algorithms, clustering algorithms, and linear algebra solvers. The library was originally written in pure C. However, in the recent years the development shifted towards C++. The library provides interfaces for C, C++, Python and Java and supports the Windows, Linux, Mac OS, iOS and Android platforms.

### 1.2.2 OpenGL

(*TODO: mozna.*) OpenGL is an application programming interface for developing 2D and 3D graphics applications. It is an open cross-platform environment providing mainly rendering and visualization functions. In 2012 was released first version of OpenGL ES, a subset of OpenGL for embedded systems including mobile phones and consoles.

---

<sup>4</sup><http://opencv.org>

## 2. Basic Notions

To make this work more self-contained, we briefly introduce basic notions and concepts used in the later chapters. Details can be found in (*UPDATE: doplnit odkazy na nejaka skripta, klidne nekolik; jednim z nich by mel byt Hartley, Zisserman: Multiple View Geometry.*) [?].

(*TODO: Co budeme potrebovat zavest.:*) (*UPDATE: nasledujici podkapi- tolky podle tohoto seznamu.*)

- convolution
- image derivatives
- gaussian image derivatives
- sum of absolute differences
- entropy, mutual information
- hessian, laplacian
- homogeneous coordinates
- camera model (mozna do sekce o MVG?)
- fundamental matrix (mozna do sekce o MVG?)

### 2.1 Convolution

*Convolution* is an operation on two functions used in signal processing in general. In image processing *convolution* performs applying some filter (kernel) over an image. The output of such *convolution* can be blurred or sharpened image for example.

**Definition 1.** *The convolution of functions  $f$  and  $g$  is an operation defined as:*

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t - \tau)d\tau.$$

*In image processing  $f$  would represent the image function and  $g$  would be the filter applied on it.*

### 2.2 Image derivatives

*Image derivatives* are local derivatives of an image function. They are proper way how to express changes in an image. By a change in an image we assume an acute change of a pixel intensity that indicates presence of an edge. After

derivation the change transforms to maxima of the image function. That is one of the methods how edges can be detected. In discrete space we can approximate derivatives by local differences as:

$$\begin{aligned}\frac{\partial I}{\partial x}(x, y) &= I(x + 1, y) - I(x - 1, y) \\ \frac{\partial I}{\partial y}(x, y) &= I(x, y + 1) - I(x, y - 1),\end{aligned}$$

where  $I$  is the input image and  $(x, y)$  are coordinates of the pixel.

## 2.3 Gaussian image derivatives

(*UPDATE: .) k teto podkapitolce jsem cerpala z nejake prezentace, co jsem nasla na internetu. nejsem si jista, jestli jsem to spravne pochopila.*

In image processing and Computer Vision scale-space is a set of down-sampled and smoothed images. Formally, it is a group of derived signals  $L(x, y; t)$  defined as *convolution* of image function  $f(x, y)$  and Gaussian kernel  $g(x, y; t)$  so that

$$L(x, y; t) = g(x, y; t) * f(x, y),$$

where  $t$  is the particular scale.

To this scale-space representation we can apply local derivatives at any scale. Equivalently, scale-space derivatives can be computed by convolving the original image  $f$  with Gaussian derivative operators which are derivatives of the Gaussian function. For this reason they are often also referred to as *Gaussian derivatives*.

## 2.4 Sum of absolute differences

*Sum of absolute differences* describes the difference of intensities of particular areas of two images. In the context of image processing it is used to measure disparity of the areas and by minimizing the difference to find the best matching points. By computation of *sum of absolute differences* we can estimate the disparity map.

## 2.5 Entropy

In computer science, *entropy* (sometimes also called *Shannon entropy*) is a measurement of the uncertainty in the image values. *Entropy* can be also perceived as a measure of histogram dispersion.

**Definition 2.** *The entropy  $H$  of a random variable  $X$  and a probability  $P(X)$  is defined as*

$$H(X) = E[-\ln(P(X))] = - \sum_k P(k) \log P(k),$$

where  $E$  is expected value operator and  $k$  is a possible value of the  $X$  variable.

## 2.6 Mutual information

*Mutual information* measures the mutual dependence of two variables and says how much one random variable tells us about another.

**Definition 3.** *The mutual Information of two variables  $I_1$  and  $I_2$  is a measure of uncertainty (entropy  $H$ ) in a probability density function:*

$$MI_{I_1, I_2} = H_{I_1} + H_{I_2} - H_{I_1, I_2}.$$

We can also perceive *mutual information* as a reduction in uncertainty about one random variable given knowledge of another. High mutual information indicates a large reduction in uncertainty; vice versa low mutual information indicates a small reduction. In the context of image processing high mutual information demonstrate high entropy and a white noise in an image.

## 2.7 Laplacian

In section ?? we mentioned *image derivatives* that are useful for detection of high variation of intensity. After derivation they occurred as maxima of the image function. If we go even further and take the second derivative we realise that the intensity changes transform to zero. Considering  $n$ -dimensional space of the function, we get  $n$  second partial derivatives. If we sum them we get the *Laplacian*.

**Definition 4.** *The Laplacian in  $n$ -dimensional space is the divergence of a gradient of function  $f$ :*

$$\nabla^2 f = \sum_{i=0}^n \frac{\partial^2 f}{\partial x_i^2}.$$

*In image processing we usually consider 2-dimensional space, thus the Laplacian is*

$$\nabla^2 f(x, y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}.$$

## 2.8 Homogeneous coordinates

In Euclidean coordinate system infinity does not exist. However, many geometric concepts are simplified when infinity is included. An example of such a form of geometry that includes infinity is *projective geometry*.

Suppose we have a point  $(x, y)$  in Euclidean plane. This point in projective geometry is expressed by extending it to the triplet value of  $(x, y, 1)$ . Generally, a  $n$ -dimensional point in Euclidean space is represented as a point in a  $(n+1)$ -dimensional projective space. The point  $(kx, ky, k)$  is certainly the same point as  $(x, y, 1)$  for any  $k \neq 0$ . These are called *homogenous coordinates* of the point. To get back the Euclidean coordinates we simply divide  $x$  and  $y$  by  $k$ . We can notice that any of  $(x, y)$  from Euclidean space corresponds to the point  $(x, y, 0)$ , because the operations  $x/0$  and  $y/0$  are undefined. This is how we get the point in infinity; points in infinity are those points which have zero as the last coordinate.

## 2.9 Projection camera model

*Projection camera model* represents a model of central perspective projection. It maps points from 3D (world) to 2D (image).

In Figure ?? we can see the camera geometry. The *camera centre* is placed at the coordinate origin. We should notice that the image plane is in front of camera. The line perpendicular to the *image plane* that originates in the centre is called *principal axis* of the camera and its intersection with the image plane is *principal point*.

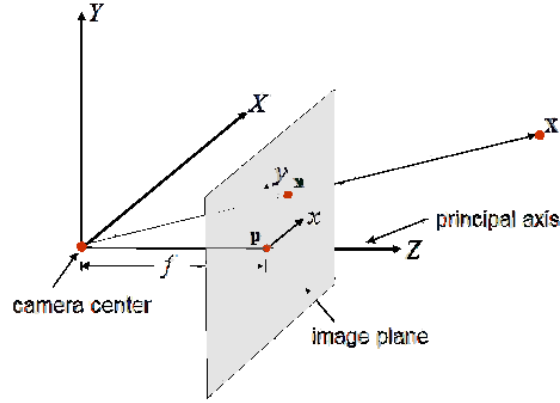


Figure 2.1: (*TODO: tento obrazek jsem stahla z googlu, mel by se nahradit.*)

## 2.10 Fundamental matrix

In Computer Vision *fundamental matrix* represents the relation of points between two stereo images. Suppose we have two stereo images and we choose a point  $\mathbf{x}$  in the first one. The knowledge of its position constraints the position of the corresponding one in the second image. The corresponding point denoted by  $\mathbf{x}'$  lies on the line defined by  $\mathbf{F}\mathbf{x}$  where  $\mathbf{F}$  is the fundamental matrix. This line is called *epipolar line*.

**Definition 5.** *The fundamental matrix  $\mathbf{F}$  for a pair of stereo images is a  $3 \times 3$  matrix which satisfies*

$$\mathbf{x}'^T \mathbf{F} \mathbf{x} = 0$$

*for corresponding points  $\mathbf{x}$  and  $\mathbf{x}'$ .*

## 2.11 Hessian matrix

The Hessian matrix describes a second-order behaviour of a function around a particular point.

**Definition 6.** The Hessian matrix of a function  $f : \mathbf{R}^n \rightarrow \mathbf{R}$  at  $\mathbf{x} \in \mathbf{R}^n$  is a matrix  $H_f(\mathbf{x})$  of the second order partial derivatives of  $f$  evaluated at  $\mathbf{x}$ :

$$H_f(\mathbf{x}) := \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}.$$

If any of the partial derivatives on the right-hand side are undefined, we say that the Hessian matrix is also undefined.

Within the context of image processing, the function  $f$  typically corresponds to the input image and derivatives are replaced either by differences between the intensity levels of neighbouring pixels or Gaussian derivatives.

The Hessian matrix forms a basis for a basic feature detector (called Hessian detector), which selects the image positions  $\mathbf{x}$  locally maximizing  $\det(H(I, \mathbf{x}))$ , where  $I$  is the input image. This leads to a detection of corners and features forming highly textured areas of size roughly comparable to the variance of the Gaussian kernel used to compute the Gaussian derivatives.

## 2.12 Integral images

An integral image, also known as *summed area table*, allows fast and efficient computation of a sum of image intensity values inside an arbitrary rectangular area. A pixel of an integral image represents the sum of all of the original image's pixels that lie to the left and above the considered position:

$$K_I(\mathbf{x}) := \sum_{i \leq x} \sum_{j \leq y} I(i, j),$$

where  $K$  is the resulting integral image,  $I$  is the input image, and  $\mathbf{x} = (x, y)^T$  is a location of a pixel.

An advantage of the integral image is that we are able to compute it using only one pass through the original image. Moreover, once we have calculated the integral image, only three integer operations and four memory accesses are required to calculate the sum of the original intensities inside any rectangular region (see Figure ??).

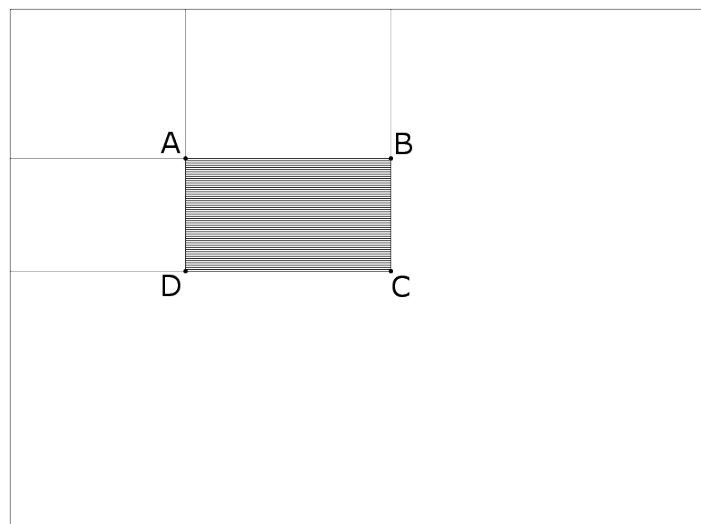


Figure 2.2: The sum of any rectangular region can be calculated by only three additions: (*UPDATE: napsat konkretni vzorec.*)  $\sum = C - D - B + A$ .

## 3. Image Processing Algorithms

In this chapter, we introduce image processing algorithms often applied in software solutions with functionality similar to those discussed in Chapter ???. Applications reconstructing depth information from photographs or video sequences typically have at their core an algorithm that detects and tracks image primitives (such as corners, blobs, lines, individual pixels, etc.) across several photos/frames. This knowledge allows one to estimate the camera parameters, which in turn makes it possible to compute the spacial parameters of the tracked primitives.

The approaches to computing correspondences between pairs of photos can be divided into *sparse correspondence* and *dense correspondence* problems. In the former, a relatively conservative subset of only the most stable *features* are considered and paired. In the latter, the objective is to obtain a correspondence in the second image for (almost) every pixel of the first image.

We describe two techniques solving the sparse correspondence problem – the SIFT [?] and SURF [?] feature detectors/descriptors – and then introduce a max-flow algorithm [?] for the dense correspondence problem. (*UPDATE: potrebujeme urcite pridat citace.*) /nevim, jestli to je takto dostacujici/

Generally, sparse feature matching algorithms first extract features from each photo. If the feature detection is repeatable, most of the features should be detected in all images where the corresponding scene element is visible. Then, for each feature a *descriptor* is generated, typically a high-dimensional vector. This descriptor is constructed in a way ensuring invariance to various image transformations. For instance, the SIFT descriptor is invariant to translation, rotation, scaling, and image noise. Thus, applying any of those transformations to the image should leave the descriptors unaffected, making it possible to match image features using the Euclidean metric. The SIFT descriptor is also partially invariant to affine transformations and illumination changes, enabling us to perform matching of images capturing the scene slightly different viewpoints.

### 3.1 Scale-invariant feature transform

Scale-invariant feature transform (SIFT) was introduced by David Lowe in [?], building on a previous work by Lindeberg [?]. The paper describes both a feature detector and descriptor. To this day, SIFT remains a popular choice for applications such as panorama stitching and object detection.

The features are detected from a grayscale version of the input photo and tend to match centers of blob-like image structures. (*TODO: neni tu urcita nesourodost v dulezitosti te prvni informace (grayscale obrazek) a druhe informace (to, co to detekuje)? pripada mi, ze to druhe je mnohem “globalnejsi” informace.*) Each feature is assigned orientation and scale, which is used to establish a reference frame for the construction of the descriptor. The SIFT feature descriptor is a 128-dimensional vector determined by the gradients around the descriptor. Since the computation of the descriptor is performed using the abovementioned local reference frame, the resulting descriptor is invariant to transformations that affect this reference frame in a covariant manner. We now proceed to describe both the detection and descriptor construction steps in more detail.



The feature detection starts by creating the scale-space of the input image, which is a series of images constructed by blurring the original one using Gaussian filters of increasing size. (The process is performed on a downsampled image for larger kernel sizes to increase computational efficiency.) After constructing the scale-space, we subtract neighbouring images of the scale-space to obtain a series of differences of Gaussians (DoG) images. A DoG image is thus equal to:

$$D_{\sigma_1, \sigma_2}(x, y) = G_{\sigma_1} * I(x, y) - G_{\sigma_2} * I(x, y),$$

where  $I(x, y)$  is the original image,  $*$  presents convolution,  $G_{\sigma_i}$  is a Gaussian kernel, and  $\sigma_i$  is its variance. Hence

$$D_{\sigma_1, \sigma_2}(x, y) = \left[ \left( \frac{1}{\sqrt{2\pi\sigma_1^2}} \cdot e^{-\frac{x^2+y^2}{2\sigma_1^2}} \right) - \left( \frac{1}{\sqrt{2\pi\sigma_2^2}} \cdot e^{-\frac{x^2+y^2}{2\sigma_2^2}} \right) \right] * I(x, y).$$

(*TODO: nasledujici vete bohuzel vubec nerozumim.*) These convolved images are grouped by octave, where an octave corresponds to doubling value of scale  $\sigma$ .

(*TODO: nekaj v tom textu budeme muset zjednotit feature/keypoint.*) We can now define candidate keypoints as local maxima and minima of  $D_{\sigma_1, \sigma_2}(x, y)$ . This is done by comparing each pixel with its neighbouring pixels and also with the corresponding pixels in neighbouring DoG images. These are exactly the areas we are looking for – spots of high contrast.

Usually, the detection of scale-space extrema results in a large number of candidate keypoints. However, some of them are local extrema created by image noise or responses of the operator along edges. Such structures are unstable and should be discarded from the set of candidate keypoints. This is performed by fitting a quadratic function to the DoG data and examining its shape. If the function is too shallow or narrow, the candidate keypoint is discarded. (*TODO: ted konkretne napsat, jak je ta kvadraticka funkce odvozena; jeji parametry by mely byt  $x, y; \sigma$  nejspis ne.*) For each candidate we perform an interpolation of nearby image data. This is done using quadratic Taylor expansion of the DoG function around the detected point:

$$D(\mathbf{x}) = D + \frac{\partial D^T}{\partial \mathbf{x}} \mathbf{x} + \frac{1}{2} \mathbf{x}^T \frac{\partial^2 D}{\partial \mathbf{x}^2} \mathbf{x}$$

where  $\mathbf{x} = (x, y, \sigma)$  is the offset from the point. If the offset value is less than **0.03**, it indicates low contrast and a high probability of affection by noise or of corresponding to edges and the candidate is discarded. Otherwise we keep the keypoint. (*TODO: az po tohle misto bychom to meli od predchoziho TODO prepsat.*)

The next step is to analyse the area around individual keypoints to generate the descriptor. Local scale of the keypoint is determined by the level of scale-space in which the keypoint was detected. To achieve invariance to rotation, local orientation must be determined as well. First, we compute a histogram of orientations of image gradients around the keypoints location. The peaks of the histogram represent dominant orientations. The orientations corresponding to the highest peak are assigned to the feature. If there are several peaks of magnitude within 80% of the highest peak, multiple keypoints are generated – one for each such orientation.

(*TODO: konstrukci deskriptoru bude nutne popsati o malinko vic.*) We create the descriptor using the gradient magnitudes of these regions. For a feature there are sixteen subregions, each with eight values of a histogram, that results in a 128-dimensional vector.

## 3.2 Speeded Up Robust Features

Another popular feature detector/descriptor is the Speeded Up Robust Features (SURF) algorithm, introduced by Herbert Bay et al. [?]. It is influenced by the SIFT algorithm described above and is based on the computation of Haar wavelet responses and Hessian-matrix approximation. Similarly to SIFT, keypoints are again detected in a scale-space to achieve invariance to scaling. In this approach, however, the scale-space is not constructed explicitly and integral images are employed instead to decrease the computation time.

The detection of interest points is performed using an approximation of the Hessian-matrix. The Hessian for a point  $\mathbf{x} = (x, y)$  in an image  $I$  at scale  $\sigma$  is defined as

$$H(\mathbf{x}, \sigma) = \begin{pmatrix} L_{xx}(\mathbf{x}, \sigma) & L_{xy}(\mathbf{x}, \sigma) \\ L_{yx}(\mathbf{x}, \sigma) & L_{yy}(\mathbf{x}, \sigma) \end{pmatrix},$$

where  $L_{xx}$  presents the convolution of the Gaussian second order derivative with the image:

$$L_{xx}(\mathbf{x}, \sigma) = \frac{\partial^2}{\partial x^2} g(\sigma) * I$$

and similarly for  $L_{xy}$  and  $L_{yy}$

$$L_{xy}(\mathbf{x}, \sigma) = \frac{\partial^2}{\partial x \partial y} g(\sigma) * I, L_{yy}(\mathbf{x}, \sigma) = \frac{\partial^2}{\partial y^2} g(\sigma) * I.$$

(*TODO: radsi bych se tomu vektoru  $x$  vyhnul a napsal  $x, y$  – kvuli tem parcialnim derivacim; hlavne dejme pozor, abychom pouzivali jednotne znaceni tech derivaci – asi bude nejlepsi nejdriv napsat podkapitolku o tom, co je Gaussovska derivace.*)

A box filter approximation of a convolution with a Gaussian-derivative kernel can be computed quickly using integral images. (*TODO: potrebujeme nekde rict, jak presne se to aproximuje, nejlip obrazkem.*) The approximations are computed for every scale of the scale-space and the points that are simultaneously local extrema of both the determinant and the trace of the Hessian matrix are chosen as candidate keypoints. The trace of Hessian matrix corresponds to the Laplacian of Gaussians (LoG) (*TODO: meli bychom rict LoG ceho.*). If we denote those approximating box filters as  $D_{xx}$ ,  $D_{yy}$  and  $D_{xy}$ , we get the determinant value

$$\det(H) = D_{xx}D_{yy} - \omega^2 D_{xy}^2.$$

(*TODO: opet musime presne definovat, co je  $D_{xx}$  a podobne.*)

The SURF descriptor is a high-dimensional vector consisting of the sum of the Haar wavelet responses around the point of interest. The vector describes the distribution of the intensity within the neighbourhood of the keypoint. (*TODO: Proc v tehle vete rikame neco, co by melo byt ekvivalentni druhe polovine predchozi vety? Obe se tvari, jako ze rikaji, cim je tvoreny ten vektor.*) At first, to

be invariant to image rotation, a dominant orientation is established for each detected keypoint. The Haar wavelet responses in  $x$  and  $y$  direction are calculated and they are summed within a sliding orientation window of size  $\frac{\pi}{3}$  afterwards. (*TODO: tady by mi nebylo jasne, co se tam s tim sliding window dela; v tuhle chvíli mam v ruce dve cisla, hodnotu derivace podle  $x$  a podle  $y$  – jak do toho vstupuje sliding window?*.) The summed responses yield a local orientation vector. (*TODO: jenom jeden vektor? Jak to, ze zahy budu vybirat ten nejdelsi z nich?*.) The longest such vector is used to establish the orientation of the feature.

Similarly to the SIFT algorithm, the local orientation and scale is used to obtain a reference frame utilised in the construction of the descriptor. The size of the inspected region around the keypoint is  $20s$ , where  $s$  is the scale where it was detected. (*TODO: nemelo by s byt nejak svazane s rozptylem  $\sigma$ ?*.) We split the region into  $4 \times 4$  regular square subregions and for each of them we compute Haar wavelet responses  $d_x$  and  $d_y$  at  $5 \times 5$  evenly spaced sample points inside. (*TODO: tady porad rikame “pocitame Haar wavelet responses” ale jde nam o to vypocitat ty derivace; ja bych to napsal tak, ze se na zacatku nadefinuji ty derivace aproximovane temi integral image a pak uz se jen pouziva dokola odkazuje se na to nejakym konkretnjsim terminem.*.) The sum of the responses for  $d_x$  and  $d_y$  and their absolute values  $|d_x|$ ,  $|d_y|$  separately results in a descriptor  $v = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|)$  for each subregion. Concatenating this for all  $4 \times 4$  subregions gives 64-dimensional vector descriptor.

Matching SURF descriptors between two images can be speeded-up by including the trace of Hessian (Laplacian) to the process of finding corresponding keypoints. We exploit the sign of Laplacian that distinguishes bright blobs on dark background from the reverse situation. In the matching stage we compare only features if they have the same sign – type of contrast. This costs no additional computation time since this quantity has already been computed at a previous stage.

### 3.3 SIFT vs. SURF

Both SIFT and SURF are popular feature descriptors based on the distribution of intensity around the interest point. SURF has been shown to have similar performance to SIFT, but being much faster. (*TODO: mozna uvest jak se to chova nasich datech nebo i zminit presneji jak to je popsane v uvodnim clanku, nebo odkazat na nejakou studii porovnavajici ruzne detektory/deskripty.*.) The results of research in [?] point out the fast computation of a descriptor. SURF describes image three times faster than SIFT. (*TODO: jestli je tohle napsane v tom puvodnim clanku, tak ty predchozi vety spojme a napisme to tak, at ta informace z toho je patrna.*.) The key to the speed is using integral images. Also, the SURF descriptor is a 64-dimensional vector, enabling faster computation of Euclidean distance between the descriptors compared to the 128-dimensional SIFT descriptors. Furthermore, due to the global integration of SURF-descriptor, it stays more robust to various perturbations than the SIFT descriptor. (*TODO: potrebujeme napsat presneji nebo vynechat.*.)

SURF offers invariance to rotation and image blurring. On the other hand, SIFT is more robust under illumination changes and viewpoint changes. (*TODO: znovu neni jasne, jestli popisujeme vlastni zkusenost nebo neco z nejakeho clanku*

(*jakeho?*).)

### 3.4 Maximum Flow Graph Cut Algorithm

(*TODO: tuhle podkapitolku potrebujeme vlastne jeste napsat.*) Presumably, rectification of the images considerably simplifies the situation. Once the correspondences are solved, we obtain a disparity map and it can be exploited to reconstruct the positions and distances of the cameras.

In 1999 Sébastien Roy published an algorithm [?] to solve the stereo correspondence problem formulated as a maxim-flow problem in a graph. This approach does not use of epipolar geometry as many other do. It solves efficiently maximum-flow and gives a coherent minimum-cut that presents a disparity surface for the whole image. This way provides more accurate depth map then the classic algorithms and the result is computed in shorter time in addition.

## 4. Developing for Android

Android, Inc. was founded in October in 2003 by Andy Rubin, Rich Miner, Nick Sears, and Chris White. At first, the company planned to develop an operating system for digital cameras but ultimately decided to focus on smartphones instead. The first phone with Android operating system was sold in autumn 2008. (*UPDATE: nebylo by lepsi tuhle informaci rict v uvodu, kdyz tam zminujeme, kdy byl prni telefon uveden?—presunuto.*) It was G1 developed by HTC (also known as the HTC Dream) that was the first device running the Android operating system on the market. Majority of Android's code is released under a free license. This allows developers to modify and freely distribute the software. Due to this, Android became the most common operating system for smartphones. In this chapter we introduce the Android operating system from programmers perspective and give a brief description of how to implement an Android application.

### 4.1 Introduction to Android development

Android is a product of a group developing open standards for mobile devices led by Google. The Open Handset Alliance includes mobile operators, software providers and device and component manufacturers such as T-Mobile, HTC, Sony, and others.

The Android platform is a layered environment built upon the foundation of the Linux kernel. It includes a wide range of functions and user interface supporting views, windows, displaying boxes and lists. Also an embedded browser build upon WebKit is included. WebKit is an open source browser engine that powers Apple's Safari web browser and the latest versions of Google Chrome. Most Android-powered devices have build in sensors to measure temperature, motion and orientation such as an accelerometer, a gyroscope or a barometer. It also provides an array of connectivity options, for example WiFi, Bluetooth or cellular data. A build-in camera support is offered as well. The data storage support is provided by SQLite, a relational database management system in the form of a library implementing self-contained and server-less SQL database engine.

As already mentioned, the basic layer of the Android platform is formed by a variant of the Linux kernel. It is used for memory and process management, device drivers, and networking. Above this level are the Android native libraries, which include graphics support, media codecs, a database library (SQLite) and WebKit. They are written in C or C++ and called through a Java interface. The actual applications are running on the Dalvik Virtual Machine (DVM), which is an implementation of Java virtual machine optimized to low processing power and memory resources. The Android Runtime Layer consists of this DVM and core Java libraries. (*UPDATE: nemeli bychom kapitalizovat Android Runtime Layer?.*) The next level up is the Application Framework, which manages the basic functions of a phone. Its major part is formed by the Activity Manager (managing the life cycle of applications) and various Content Providers (managing data sharing between the applications). Other notable parts are the Telephony Manager (handling voice calls), the Location Manager (specifying location using GPS or cell tower) and the Resource Manager. The top layer of the Android architec-

ture is formed by the individual applications. Some of them are preinstalled and provided by Google and parts are third party applications created by the community. The structure of the system is illustrated in Figure ?? below. (*UPDATE: tady spis odkazat “in Figure XY”.*) Android applications are written in the Ja-

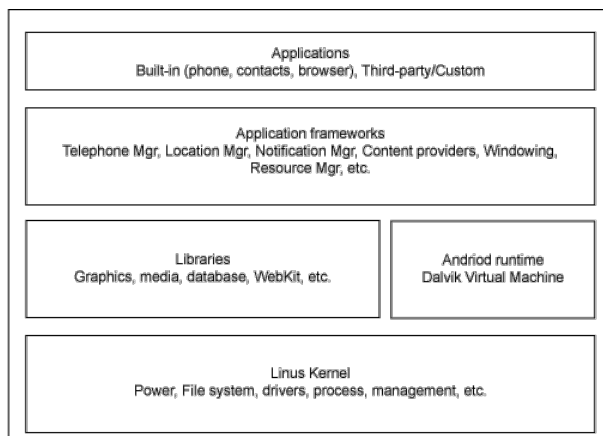


Figure 4.1: The structure of Android architecture.

va programming language and run within an instance of Dalvik Virtual Machine. (*UPDATE: Proc jsme ted od vykreslovani GUI skocili zpatky k instalovani aplikace na zarizeni? Cekal bych, ze usporadani textu bude respektovat life cycle aplikace.*) An inseparable part of the application is `AndroidManifest.xml` file containing necessary information how to properly install it to the device including permissions the application needs to run. An example of such a permission is the ability to use the phone’s camera or to access the Internet. All such requirements need to be explicitly listed in the manifest file. We concentrate more on activities and their development in the following sections.

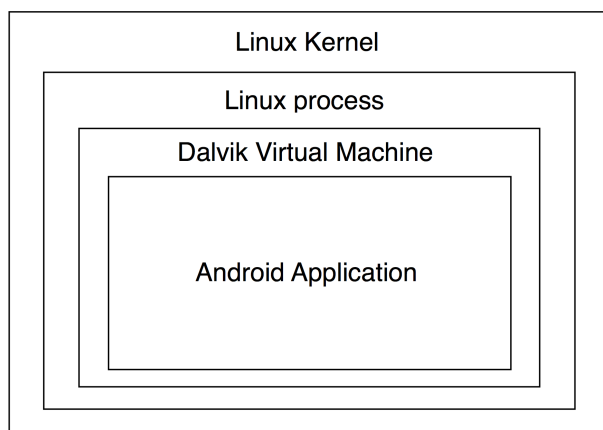


Figure 4.2: A typical Android application runs within an instance of Dalvik Virtual Machine on a Linux kernel.

The tools required for the development of an Android application are the Android Software Development Kit (SDK) and a Java IDE, such as Eclipse or IntelliJ IDEA. The Android SDK contains an Eclipse plugin supporting Android development, documentation, sample code, and various tools, such as Android Debug

Bridge used by the programmer to communicate with an application running on a device.

## 4.2 Application Components

When an application is installed to a device, it lives in its security sandbox. Android offers a secure environment where the applications are isolated and can safely run their code. To each application the Linux system assign user ID number. Android is multi-user operating system, which means that each application is a different user with a different ID. This number is known only to the system that sets permissions to all files so they can not be accessed by any other applications. Each application runs unique Linux process initiated when an application or any component of it needs to be executed. When the system needs to recover the memory or the application is not used for some time, the process shuts down. Releasing memory works on the principle “oldest first” – at first the system kills the apps that were inactive for the longest time. Each process has its own virtual machine. This provides the security and isolation between the applications.

The security is risen by the principle of least privilege. Due this principle an application has access only to the required components of the system. These requirements must be stated in the manifest file. This protects the whole Android system and the applications from each other.

An application has several components that are used to build it. We can divide the basic building blocks into four different groups:

- Activities
- Services
- Content providers
- Broadcast receivers

*Activity* is a screen providing the user interface. In the activity a developer place elements such as Views, Lists, Buttons, Labels etc. The layout of an activity and the widgets placed in the window are described in a separate XML file (we reveal more about user interface later). Most of applications consist of more than one bounded activities. Although the activities form one application, they are all independent from each other. If another application has a permission to do so, it can start an activity of a different app.

Usually in a application there is a one main activity that shows up to the user at first after launching it. To this one are all other activities linked. Each activity in android will be subclass of Activity class defined in Android SDK.

A *service* is a component used to perform operations in background. It can be invoked by an activity for example. If the application needs to run long-term process, user can switch to another application and our process can continue to run in background. This can be exploited to develop an application to play music or an application which needs to fetch data in the background.

A *content provider* manages sharing data across the applications. When the app is storing any data - in file system, SQLite database or any online storage,

through the content provider they can be accessed or even modified from other applications. An example when a content provider is used is an application working with the contacts database.

A *broadcast receiver* is a component broadcasted by the Android system. It usually inform applications about some basic actions such as turning the screen off, running out of battery or changes of timezone. Applications can also initiate broadcasts to let other apps know about some action, that there some data ready to use or about completed downloading for example.

From these mentioned components, activities, services and broadcasts are activated by intents. An intent is a message to the system to invoke new activity, service or a broadcast. It is a component activating mechanism in Android.

A very important part of an application is `AndroidManifest.xml` file, where are specified all permissions of the application. Each activity we create must be defined in it. Basically, every component that is used in the application must be declared in the manifest.

## 4.3 The Activity Lifecycle

Any application switches between different states of its life cycle. (*UPDATE: jak souvisi to, ze aplikace ma aktivitu s tim, ze aplikace prechazi mezi nejakyimi stavy? V dalsim se pak hovori stavech aktivitu a ne aplikaci.*) Compared to desktop platforms, the programmer has only limited control over the state transitions. (*UPDATE: napsat priklad toho, co se muze stat na Androidu a nemuze se stat na desktopu.*) On the desktop computer we can handle minimalizing or closing a window of an application or quit the entire app. This we can not affect on the Android device. The life cycle is a collection of functions operating system calls on the application while it is running.

There are five stages of the life of an application:

- the starting state,
- the running state,
- the paused state,
- the stopped state,
- and the destroyed state.

The starting state and the destroyed state are phases of the activity when it is not in the memory. To launch the app, the method `onCreate()` is called and eventually it is in a running state. When the activity is in a running state it is actually on the screen seen by the user. The activity is in foreground and handling all user interactions such as typing or touching the screen. An activity in this state has the highest priority of getting memory in the Android Activity stack to run as quickly as possible. It will be killed by the operating system only in extreme situations. This transition from starting to running state is the most expensive operation in terms of battery requirements. That is also the reason why we do not destroy every activity when it gets to the background, because



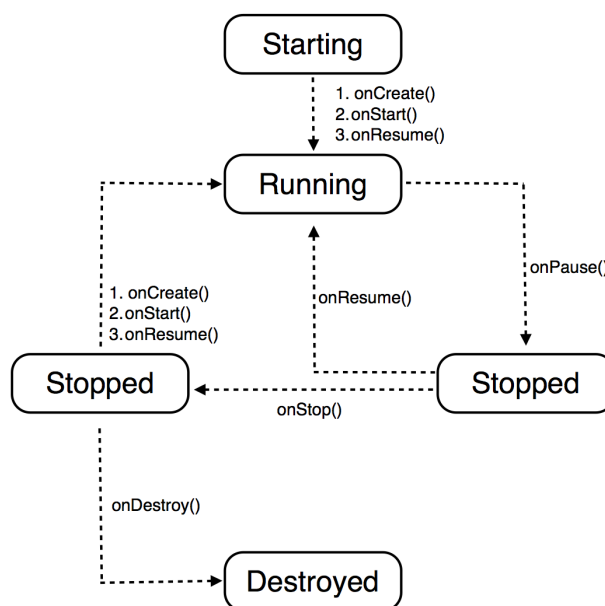


Figure 4.3: Life cycles of an application.

it is probable that it is going to be called back in a while and we don't want to create all of it again.

The application is paused (in a paused state) when it is not interacting with user at the moment, but still visible on the screen. This state does not occur that often, because most of applications cover entire screen. But when a dialog box appears or another activity is on the top of this one but not hiding it all, the underlying activity is partly visible and it is paused.

When the application is not visible, but still in memory, it is in a stopped state. Then it is easy to bring it up to the front again or it can be destroyed, removed from the memory or brought to the destroyed state.

(*UPDATE: Pribyly nasledujici sekce - User Interface, Sensors a Data Storage.*)

## 4.4 User Interface

The user interface is part and parcel of an Android application. Basically, an application can not live without the user interface.

All elements of the user interface are built using *View* or *ViewGroup* objects. By using them we also define the hierarchy of used components - a *View* can be nested within a different *View*. The layout describing the visual structure of the *Views* of the user interface is declared in separate XML file. Each *View* is represented as a XML element in it corresponding to the *View* class or subclass. The user interface can be declared also right in the code of an application. Defining it separately make the user interface more flexible to changes without any modification of the code though. Also, the structure of XML corresponds to the structure of UI and makes it more inspectional and easier to debug. In the XML file declaring layout can be only one root element. It must be a *View* or *ViewGroup* which is *LinearLayout* or *RelativeLayout* for example. *LinearLayout* is a layout that arranges the inner elements in a single row or a column depending on

the set orientation, while in *RelativeLayout* are positions of its children described in relation to parent or each other.

The developer tools for Android gives us an option of creating other parts of UI beside sketching the layout such as *Menus*, *Action Bars*, *Dialogs* or *Toasts*. There are several types of *Menus* such as *options menu* or *context menu*. The *options menu* is an user interface component commonly used in many applications collecting menu items for an activity. It appears when a user touches the menu button. From the Android version 3.0 higher, options menu items are part of the action bar after touching the action overflow button. The *context menu* is a floating menu associated with an particular element in a view that is displayed after touching the element.

*Action Bars* are a proper way how to navigate the user through the application. The *Action Bar* is located at the top of an activity and can display the activity title, icon or other actions and items. Some Android devices have a hardware action overflow button which opens the *options menu* at the bottom of an activity (as we mentioned above). *Action Bars* are superiors to *options menus* since an *Action Bar* is always visible while *options menu* shows only on the user request.

*Dialog* is a small window informing the user about some action. Unlike the *Toast* it usually requires the users action before continue by confirmation or giving some additional information. The *Toast* provides only feedback about some action in a small popup that disappears after a moment.

## 4.5 Sensors

Most Android devices have build in motion, environmental and position sensors. Motion sensors are sensors measuring rotation along three axes and give us an opportunity to detect motion, shaking or tilt. For measuring temperature, illumination, or humidity we use environmental sensors. Build-in magnetometers and orientation sensors provides information about current position of a device and can be used in application for GPS navigation or map browsers. We can access sensor events within an instance of the *SensorManager* class.

## 4.6 Data Storage

We have several possibilities how to store our application data. The way we choose depends on the needs and the level of privacy. If we use private files in the application and do not want other application to access them, we prefer using the internal storage. We read and write private data simply by using *FileOutputStream* and *FileInputStream*.

The media files such as pictures, video or audio records are usually expected to be shared and accessible from other applications. In that case the external storage is used. These files are readable and can be modified by user. Methods on the *Environment* class provide the access to the external storage. The function *getExternalStoragePublicDirectory()* depending on the argument given to it returns the path of the root directory where we can write our data, for example

- ../Music/

- ../Pictures/
- ../Movies/
- ../Download/
- etc.

As an argument we pass the type of the director we want to access, for example *Environment.DIRECTORYMUSIC*, *Environment.DIRECTORYPICTURES*, or *Environment.DIRECTORYMUSIC*.

## 4.7 Developing Android Application

In this section we give an outline of implementing the bases for an Android application.

When developing an Android application usually we need to define some user interface. Some of the components of the UI are described in section ?? . The layout of all components is defined in a .xml file. All .xml files are stored in */res/layout* directory.

The file consists of one root and other descendant xml elements. Each element is described by the inner attributes of it. We can see an example of such layout below.

---

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <org.opencv.android.NativeCameraView
        android:id="@+id/camera_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <RelativeLayout
        android:layout_width="200dp"
        android:layout_height="match_parent"
        android:layout_margin="10dp"
        android:layout_centerInParent="true">

        <Button
            android:id="@+id/captureButton"
            android:layout_width="90dp"
            android:layout_height="90dp"
            android:layout_alignParentBottom="true"
            android:layout_centerHorizontal="true"
            android:background="@drawable/circle_button"
            android:onClick="callTakePicture"/>
        </Button>
    </RelativeLayout>
</RelativeLayout>
```

```

<Button
    android:id="@+id/startCapturingButton"
    android:layout_width="90dp"
    android:layout_height="90dp"
    android:layout_alignParentBottom="true"
    android:layout_centerHorizontal="true"
    android:background="@drawable/circle_button"
    android:onClick="start_stopAutoCapturing"
    android:visibility="invisible"/>

<ToggleButton
    android:id="@+id/toggleAutoCaptureButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentTop="true"
    android:layout_centerHorizontal="true"
    android:textOn="AutoCapture on"
    android:textOff="AutoCapture off"
    android:onClick="autoCaptureStateChanged"/>

<TextView
    android:id="@+id/startCapturing_textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_above="@+id/captureButton"
    android:layout_centerHorizontal="true"
    android:text="@string/startcapturing_text"
    android:visibility="invisible"/>

</RelativeLayout>

</RelativeLayout>

```

---

In our example the root element is *RelativeLayout* element which allows us to describe the relative position to each other or to a parent. Inside the *RelativeLayout* we can find *org.opencv.android.NativeCameraView* which is an element of OpenCV library used for camera view and another *RelativeLayout* with *Buttons*, a *ToggleButton* and a *TextView*. All of the elements have defined the width and height by using *android:layoutwidth* and *android:layoutheight* attributes. The value of these attributes can be *matchparent* or *wrapcontent* or the specific number of dp units.

It is also possible to define different layout of one activity depending on the screen orientation. If we need to define different layout for landscape orientation, we save a xml file of the same name with new layout into *res/layout-land* folder.

In the *src* folder are stored all .java files representing activities or classes. Every Android application has at least one activity which is called by the system when the application is launched. Due to the activity lifecycle (see ??) it is necessary to implement the following methods:

- *onCreate()*,
- *onPause()*,
- *onResume()*.

The *onCreate()* method handles what happens when the activity is started. Usually only the basic startup instructions should be completed in this method, for example setting the layout or initialisation of some class variables. The particular layout into an activity is loaded by *setContentView(name-of-the-xml-file)*. We can see an example of *onCreate()* method below.

---

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_gallery);

    adapter = new Adapter(this);
    ArrayList<GridImage> db_update = new ArrayList<GridImage>();
    for (String s : files) {
        ...
    }
    ...
}
```

---

Depending on the needs of our application, we can add some code into the other overriding methods handling the activity lifecycle.

---

```
@Override
public void onPause(){
    super.onPause();
    ...
}

@Override
public void onResume(){
    super.onResume();
    ...
}
```

---

To make our user interface interactive we need to set some functions that will be executed when clicking on the elements of the user interface. This can be done either in the xml file or in the code of the activity. To demonstrate these two approaches, we added an example:

---

```
/* This is the definition of a button in the xml file.
   When clicking on the button, takePicture() method will be executed.
   The method must be defined in the .java file of the activity. */
```

```
<Button
...
    android:onClick="takePicture"
... />

/* This is the example how to define
   the method directly in the code. */

Button my_button = (Button)findViewById(R.id.button);
my_button.setOnClickListener( new View.OnClickListener() {

});
```

---

## 5. OpenCV

OpenCV (a short for Open Source Computer Vision) is an open source library covering the area of Computer Vision. It was designed mainly for real-time applications and computational efficiency. OpenCV is written in C and C++, can take advantage of multicore processors and (*TODO: abychom si touhle zminkou ale nenabehli a nekdo se neptal, jak pouzivame na tom androidu parallelismus.*) runs on various platforms including Linux, Windows, and Mac OS X. Bindings for Python, Matlab and other programming languages have been introduced as well. OpenCV contains over 500 functions spanning many areas of Computer Vision, for example medical imaging, camera calibration, robotics or stereo vision.

### 5.1 The Origin of OpenCV

OpenCV was developed by Intel and the first alpha-version was released in 1999. At first, the project was an Intel Research initiative to advance CPU-intensive applications, part of a series of projects including real-time ray tracing and 3D display walls. That time, several university groups developed open computer vision infrastructure, code that every student could reach and use to his or her own vision application. This code was adapting by the students, building on the top what was implemented before. When one of the OpenCV authors noticed this, OpenCV was conceived as a way how to create computer vision infrastructure universally available.

The team developing OpenCV included experts from Russia (e.g. Vadim Pisarevsky) and Intel's Performance Library Team. The goals supporting the reason why to concern with this research were to provide not only open but also optimized code for computer vision, disseminate knowledge about computer vision by providing easily accessible open library and to advance vision-based commercial applications.

The first 1.0 version was released in 2006. From 2008 OpenCV is supported by Willow Garage, who are notable for their Robot Operating System. As of today, the latest version of OpenCV is 2.4.5 released in the April of 2013.

(*TODO: tady ted zase rikame neco, co bych cekal bud maximalne v uvodnim odstavci.*) Since the first release, OpenCV has been used in many applications. It has even been used in sound and music recognition area where the vision techniques were applied to sound spectrogram images.

### 5.2 Structure of OpenCV

OpenCV consist of image processing functions and algorithms and because computer vision and machine learning often go hand-in-hand, OpenCV also contains Machine Learning Library(MLL). This part of OpenCV library is focused on statistical pattern recognition and clustering.

Basically, we can structure OpenCV into four components, illustrated on Figure ???. First of all, it would be the CXCore component containing the basic data structures and operations on them. The HighGUI component ensures I/O rou-

tines and functions for loading and storing images. ML is the machine learning library. Finally, the Computer Vision component containing basic image processing and vision algorithm. CXCore provides the core functionality including

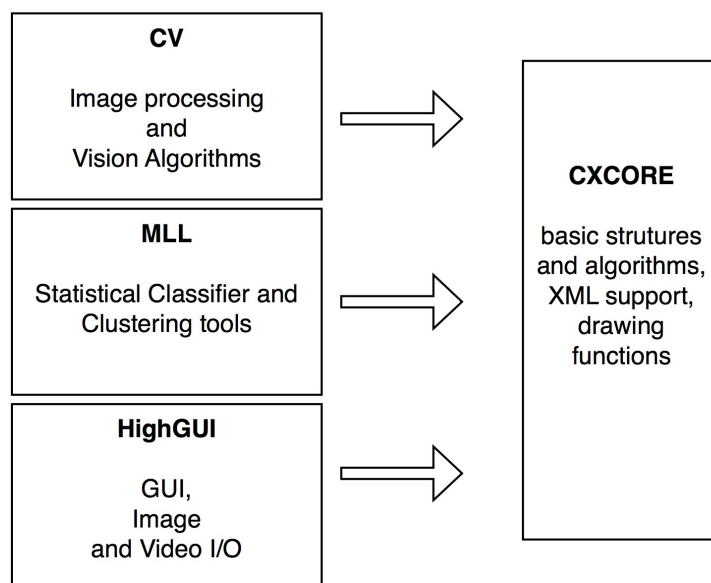


Figure 5.1: The basic structure of OpenCV library.

*(TODO: ten seznam prosim predelejme tak, aby byl spravne typograficky: mala pismenka a carky na konci radku krome posledniho zakonceneho teckou.):*

- Basic Structures
- Operations on Arrays
- Dynamic Structures
- Drawing Functions
- XML/YAML Persistence
- Clustering and Search in Multi-Dimensional Spaces
- Utility and System Functions and Macros.

In this part of OpenCV library we can find the basic structures that are needed for image processing. They are represented as structures in the class Basic Structures. For example **CvPoint** defines a point, **CvSize** is a set of two numbers representing size of rectangle, **CvScalar** is a container of double values, **IplImage** is a structure inherited from the Intel Image Processing Library designed for loading images, or **CvMat** is a data structure for storing a matrix.

As we can presume, the basic methods to work with these structures are implemented. We can find there functions for mathematical operations on matrices such as multiplication (`void cvMul()`), transposition (`void cvTranspose()`), or others like dividing multi-channel array into several single-channel arrays (`void cvSplit()`).

Available are also drawing functions working with matrices or images including `void cvCircle()` which as arguments takes an image, centre point of the



circle, and a colour and draws a circle in the image. Similarly `void cvEllipse()` or `void cvDrawContours()`.

The section with functions for image processing and computer vision provides these functionalities:

- Image Filtering
- Geometric Image Transformations
- Miscellaneous Image Transformations
- Histograms
- Feature Detection
- Motion Analysis and Object Tracking
- Structural Analysis and Shape Descriptors
- Planar Subdivisions
- Object Detection
- Camera Calibration and 3D Reconstruction

## 5.3 OpenCV4Android

(*UPDATE: uprava nasledujici kapitoly.*)

Since 2010 OpenCV was also ported to the Android environment, it allows to use the full power of the library in the development of mobile applications. In this part we will list the important structures and functions of OpenCV4Android library used in our implementation.

Images are commonly stored in `Mat` structure from the `org.opencv.core` package as a matrix of individual pixels of the image. For reading and writing images from saved files, we can use `imread(String filename)` and `imwrite(String filename)` functions from the `org.opencv.highgui` package.

In comparison with desktop version OpenCV4Android also includes the `opencv.android` package containing all the additional functions for the Android platform. This package provides mainly an environment for work with camera by offering classes such as `NativeCameraView.java` or `CameraBridgeViewBase.java`. It controls the frame process, camera adjustments and drawing the resulting frame to the screen.

The following packages contain basic methods used in image processing and implementations of commonly used algorithms (some of them we described in chapter ??):

- `org.opencv.imgproc`,
- `org.opencv.features2d`,
- `org.opencv.calib3d`.

In the image processing package are functions for image scaling (`Imgproc.pyrDown()`, `Imgproc.pyrUp()`), blurring image (`Imgproc.GaussianBlur()`), histogram calculation (`Imgproc.calcHist()`) or methods for converting images between different colour definitions (`Imgproc.cvtColor()`).

With an instance of `FeatureDetector.java` and `DescriptorExtractor.java` classes and calling `compute()` function we can detect keypoints in a set of images. There are several options for selecting descriptor type (such as BRIEF, BRIS, FREAK, ORB, SIFT) and algorithm for keypoint detection (BRISK, DENSE, FAST, ORB, HARRIS, STAR). All keypoints can be stored in a `KeyPoint` structure. Since the keypoints in all images are detected, we can match them using `DescriptorMatcher.java` class.

The `StereoSGBM.java` class included in `org.opencv.calib3d` package is defined for computing stereo correspondence using the semi-global block matching algorithm, which is a modification of algorithm by Heiko Hirschmüller [?].

## 6. Implementation

This chapter is devoted to implementation of our software. The result of this work is an Android application that allows to shoot pictures with a camera, browse them and pick photos for further processing. The output is a 3D model representing the disparity map which was calculated on the base on the selected input data. In the first section we enumerate the subtasks and chosen algorithms and then describe the structure of the software resources.

### 6.1 Implementation outline

To implement the Android application, several subtask must be completed. At first it is the graphical user interface and handling the camera which is managed through Android Activities and xml files defining the layout of the graphical interface.

Secondly, we deal with the calculation, which consist of image registration, keypoint detection, keypoint matching and solving the dense correspondence problem. Image registration is implemented by using Sum of absolute differences described in [1]. Keypoints are detected with the Harris edge detector and described with ORB descriptor, which is an oriented variation of BRIEF (Binary Robust Independent Elementary Features) [2]. Each extracted keypoint we try to match with one extracted keypoint in the second image but only in the corresponding area determined due the previous image registration. In this way we save calculation time since we do not have to compare every two detected keypoints.

### 6.2 Project structure

The project resources can be divided into four parts:

- Activities,
- Classes,
- Structures,
- xml files defining layout.

# Conclusion

# Bibliography

- [1] H. BAY, A. ESS, T. TUYTELAARS, AND L. VAN GOOL. *Speeded-Up Robust Features (SURF)*. In ECCV, 2006.
- [2] M. CALONDER, V. LEPETIT, M. OZUYSAL, T. TRZCINSKI, C. STRECHA, AND P. FUA. *BRIEF: Computing a Local Binary Descriptor Very Fast*. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2012.
- [3] K. G. DERPANIS. *Overview of the RANSAC Algorithm*. (*TODO: kde to bylo publikovano.*), 2010.
- [4] C. HARRIS AND M. STEPHENS. *A combined corner and edge detector*. In Proceedings of the Alvey Vision Conference, pages 147 – 151, 1988.
- [5] R. HARTLEY, A. ZISSERMAN. *Multiple View Geometry in Computer Vision*. ICambridge University Press, 2003.
- [6] H. HIRSCHMÜLLER. *Stereo Processing by Semiglobal Matching and Mutual Information*. Pattern Analysis and Machine Intelligence, 2008. IEEE Transactions on.
- [7] IDC Worldwide Mobile Phone Tracker, February 14, 2013
- [8] F. JURIE AND C. SCHMID. *Scale-invariant shape features for recognition of object categories*. In CVPR, volume II, pages 90 – 96, 2004.
- [9] T. KADIR AND M. BRADY. *Scale, saliency and image description*. IJCV, 45(2):83 – 105, 2001.
- [10] T. LINDBERG. *Feature detection with automatic scale selection*. In International Journal of Computer Vision, 30(2):79 – 116, 1998.
- [11] D. LOWE. *Object recognition from scale-invariant features*. In CCV, 1999.
- [12] S. ROY. *Stereo Without Epipolar Lines: A Maximum-Flow Formulation*. In International Journal of Computer Vision, 1999.
- [13] N. SNAVELY; S. M. SEITZ; R. SZELISKI. *Modeling the world from Internet photo collections*. In International Journal of Computer Vision, 2007.
- [14] N. SNAVELY; S. M. SEITZ; R. SZELISKI. *Photo tourism: Exploring photo collections in 3D*. ACM Transactions on Graphics (SIGGRAPH Proceedings), 25(3), 2006, 835-846.
- [15] N. SNAVELY; S. M. SEITZ; R. SZELISKI. *Skeletal graphs for efficient structure from motion*. Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on.

# List of Tables

# List of Abbreviations

# Attachments