Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS

Viktorie Vášová

3D Computer Vision on the Android Platform

Computer Science Institute of Charles University

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 Prague, date 20 July 2013 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

$\partial I_{\bar{1}}$

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 have 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 type 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 [18].

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. The Android market share increased to almost 69% in the year 2012 [9] (Figure 1). A built–in camera, a relatively fast CPU and various sensors such as the accelerometer are very common in such mobile phones, making it possible to develop a wider range of applications.

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 the ways of connecting these two phenomena. Our aim is to create an Android application that takes a pair 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. One of our main goals is to investigate how well the solving of such a computation-intesive problem can be done within the limits of a Java-based environment running on a mobile phone or a tablet computer.

Outline of the thesis

The first part of this work analyzes the problem, describes currently available software packages solving related problems, and gives an overview of some of the programming libraries employed in this work (Chapter 1). In Chapter 2, we introduce the mathematical concepts fundamental to the area of computer vision. This part also fixes the notation used in the remaining parts of the thesis. In Chapter 3, we rigorously discuss the problem of 3D reconstruction from images. Furthermore, we describe several algorithms from the literature that have been successfully applied to this problem. Then, we elaborate on the basics of Android development (Chapter 4). The following section is devoted to the implementation of our application. Finally, we evaluate and benchmark the resulting application (Chapter 6).

1. Overview

In this chapter, we give an overview of software solutions providing functionality related to our area – namely, the analysis of depth information from photos and videos. Section 1.1 discusses software packages currently available to the end-users, while Section 1.2 describes software libraries implementing relevant algorithms. Section 1.3 is devoted specifically to our solution. There, we discuss what type of input the software processes and what kind of output can the user expect as a result.

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 extracting 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 these *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 video 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[1] and Autodesk Matchmover[2]. The opensource libmv project[3] aims to add matchmoving capabilities to the Blender 3D modelling application[4].

1.1.2 Microsoft PhotoSynth

Microsoft PhotoSynth, based on a research by Snavely, et al. [18], 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 that respects the physical proximity of the cameras taking the individual photos. Perhaps most notably, the technology has

[1] `http://www.2d3.com`
[2] `http://www.autodesk.com`
[3] `http://code.google.com/p/libmv/`
[4] `http://www.blender.org`

been employed at various times by the BBC and CNN [3]. Recently, the possibility 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. It extends a system called *Photo Tourism* developed by Snavely, Seitz and Szeliski [16, 17]. To obtain accurate information about the positions of the cameras it applies the SIFT algorithm (discussed in Section **??**) to detect points of interest.These are then matched across images using an implementation of an approximate nearest neighbour algorithm and filtered using the RANSAC paradigm [5]. The theory presented in the classical monograph [7] is then applied to obtain external and internal camera calibration (if there is relevant EXIF information associated with the photo, then the software uses this to obtain an initial guess of the internal camera calibration parameters).

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 implementations of some of the algorithms necessary to build our software.

1.2.1 OpenCV

OpenCV[5] 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++. This led to the introduction of a new object oriented API.

The library provides interfaces for C, C++, Python, and Java and supports the Windows, Linux, Mac OS, iOS, and Android platforms. The Android version of the library, called *OpenCV4Android*, provides an access to the OpenCV methods using JNI bindings. The same approach has been used by the competing *JavaCV* library[6], which actually provides access to a wide range of computer graphics libraries (e.g., FFmpeg, OpenKinect, ...).

[5] `http://opencv.org`

[6] `https://code.google.com/p/javacv/`

Let us now compare three variants of the same example code for OpenCV, OpenCV4Android and JavaCV to illustrate the difference between these interfaces. A typical C++ code to detect strong corners on an image using the OpenCV function `goodFeaturesToTrack` would be:

```
std::vector<Point2f> corners;
goodFeaturesToTrack(img, corners, maxCount, qualityLevel, minDistance,
    mask, blockSize, useHarrisDetector, k);
```

An OpenCV4Android version of the same code reads:

```
MatOfPoint corners = new MatOfPoint();
Imgproc.goodFeaturesToTrack(img, corners, maxCount, qualityLevel,
    minDistance, mask, blockSize, useHarrisDetector, k);
```

The JavaCV version of this would be:

```
CvPoint2D32f corners = new CvPoint2D32f(maxCount);
int[] count = { maxCount };
cvGoodFeaturesToTrack(img, eig, temp, corners,
  count, qualityLevel, minDistance, mask,
  blockSize, useHarrisDetector);
```

Here, note that the JavaCV binding is derived from the older pure-C interface of OpenCV. For this reason, the function still expects the parameters `eig` and `temp` even though they are actually ignored by the current version of the library. The way in which the `maxCount` parameter is passed to the function is another relict from the old versions of OpenCV, where it was necessary to pass it using a pointer to get back the length of the array `corners` dynamically allocated inside the `cvGoodFeaturesToTrack`.

Since OpenCV4Android is developed by the same group of developers as the original OpenCV and provides access to functions available only in newer versions of the library, we have decided to choose this version for the final version of our project.

1.2.2 OpenGL

OpenGL[7] 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. OpenGL ES, a subset of OpenGL for embedded systems including mobile phones and consoles, has been released in 2012.

[7] `http://www.opengl.org/`

1.3 Problem statement

Our application's objective is to provide the user the posibility to create a rough 3D model of a scene pictured on a pair of photos. We expect that:

- both photos are focused, reasonably sharp and not significantly over- or under-exposed,
- the viewpoints the camera had when taking the photos differ only by translation,
- the above-mentioned translation is not negligible – our software certainly cannot reconstruct the depth information from a pair of photos that are identical,
- there is some overlap between the two photos, i.e. some scene elements are visible on both photos,
- the scene on the photos is highly textured – for example, taking photos of historical houses would typically result in an appropriate input.

The output of the application is a visualization of a "depth map" showing the reconstructed depth information for the overlapping part common to both input photographs. The intended purpose of the reconstruction is purely for its visualization. However, we can forsee the extension of the software to offer additional features. For example, the ability to estimate relative dimensions of objects could be provided, although the accuracy of this would depend on how precisely the internal calibration of the phone's camera (most importantly, its focal length and sensor size) can be estimated.

2. Basic Notions

To make this work more self-contained, we briefly introduce basic notions and concepts used in the later chapters. Interested reader is refferred to the monographs [1, 7] for further details.

2.1 Convolution

Convolution is an operation on two functions often encounterred in signal processing. In image processing, convolution is typically used to apply a particular filter (kernel) to the image. For example, the output of such convolution can be a blurred image. Convolution with a Gaussian kernel is particularly common.

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 our setting, the f from the above definition represents the image function and g the filter.

2.2 Image derivatives

Image derivatives of an image function are analogous to derivatives of a real continuous function. They allow us to meassure spacial changes in an image by expressing the rate of image intensity change in a particular direction. High values often indicate an edge in the image data. They are defined as

$$1(x,y) = I(x+1,y) - I(x-1,y),$$
$$1(x,y) = I(x,y+1) - I(x,y-1),$$

where I is the input image and (x, y) are coordinates of the pixel. A generalization of this is often considered since the above discretization is not able to detect significant intensity changes spread across more than few pixels.

2.2.1 Gaussian image derivatives and scale-space

Scale-space of an image is a series of gradually more and more smoothed images obtained by convolving the original image with Gaussian kernels of increasing size. Formally, it is a series of image functions $L(x, y; t)$ defined as the convolution of the image function $f(x, y)$ and the Gaussian kernel $g(x, y; t)$, where t is the corresponding variance:

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

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*.

2.2.2 Laplacian

As already mentioned, *image derivatives* are useful for the purpose of the detection high variations of image intensity values. After taking the first derivative of the image function, points of highest intensity change are those where we have local maxima. If we go further and take the second derivative, then the corresponding values transform to zeroes. Thus, it might be important to consider the sum of derivatives in the direction of both axes to detect such structures.

Definition 2. *The Laplacian of a function with n-dimensional support is the divergence of the gradient of a 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 becomes:

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

Similar argument to the above can be used to deduce that local maxima and minima of the Laplacian function indicate the presence of a blob-like structure in the image data. The size of the structure is related to the variance of the Gaussian used in the definition of the Gaussian image derivatives of Section 2.2.1. For this reason, algorithms like SIFT [13] repeat some parts of the computation for varying values of this parameter to detect structures of all sizes.

2.2.3 Hessian matrix

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

Definition 3. *The Hessian matrix of a function $f : \mathbf{R}^n \to \mathbf{R}$ at $\mathbf{x} \in \mathbf{R}^n$ is a matrix $H_f(\mathbf{x})$ of the second order partial derivatives of f evaluated at $\mathbf{x}$:*

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

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

Note that the trace of the Hessian matrix is equal to the Laplacian at the same point $\mathbf{x}$.

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 of size roughly comparable to the variance of the Gaussian kernel used to compute the Gaussian derivatives.

2.3 Basic image similarity metrics

We now introduce two simple metrics that can be used to estimate similarity of visual contents of a pair of images. These are often used as a basis for more sophisticated image registration algorithms, e.g. [8].

Sum of absolute differences describes the difference of intensities of corresponding areas of two images:

Definition 4. *Let $I_1(x, y)$ and $I_2(x, y)$ be two grayscale images. Then the sum of absolute differences of I_1 and I_2 is the value $\sum_{x,y} |I_1(x, y) - I_2(x, y)|$.*

Note that this value is going to be zero for a pair of identical images.

For the second metric, we need to recall the definition of *entropy* (sometimes also called *Shannon entropy*). It can be viewed as a meassure of information complexity of a random variable.

Definition 5. *The entropy H of a random variable X with probability distribution $P(X)$ is defined as*

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

where $\mathbb{E}[\cdot]$ is the expected value operator.

In computer vision, it is used to meassure the "complexity" of the image histogram when the underlying random variable is set as $X = I(\mathbf{x})$, where $\mathbf{x}$ is a uniformly randomly generated image position. A constant, uniform image attains minimum entropy while random noise maximizes it.

We can now introduce the *mutual information* of two images. Generally, it measures the mutual dependence of two variables. It expresses how much the value of one random variable predicts the value of the other.

Definition 6. *The mutual information of two variables I_1 and I_2 is defined as*

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

where H_{I_1,I_2} is the entropy $H(X)$, where $X = (I_1(\mathbf{x}), I_2(\mathbf{x}))$ for $\mathbf{x}$ uniformly random. In the context of computer vision, the vector $\mathbf{x}$ is a random image position.

We can also interpret *mutual information* as a reduction of uncertainty of one random variable given the knowledge of another. High mutual information indicates a large reduction in uncertainty and vice versa. Mutual information of two images depends only on their joint histogram. Again, this value attains zero for a pair of identical images. However, it remains zero even after, e.g., increasing the brightness of one of the images. Thus, this meassure is invariant under illumination changes, which is of great importance when matching photos taken in an uncontrolled environment.

2.4 Projective geometry

In the standard Euclidean space $\mathbf{R}^n$, the infinity does not exist. However, many geometric concepts are simplified when the notion of infinity is included. An example of such a geometry is *the projective geometry*. Projective plane $\mathbb{P}^2$, or generally projective space $\mathbb{P}^n, n \in \mathbb{N}$, is obtained by extending $\mathbf{R}^n$ by including a point at infinity for each direction.

Numerically, the points in $\mathbb{P}^n$ are represented using non-zero vectors from $\mathbf{R}^{n+1}$. Suppose we have a point (x, y) in the Euclidean plane. This point in projective geometry is expressed by the vector $(x, y, 1)$ and any of its non-zero multiples $k \cdot (x, y, 1), k \neq 0$. These are called *homogenous coordinates* of the point. In what follows, we identify the point in a projective space with the vector of its homogeneous coordinates. To get back the Euclidean coordinates we simple divide the first two coordinates by the third. We can notice that none of the points (x, y) from Euclidean space corresponds to a projective point of the form $(x, y, 0)$, because the operations $\frac{x}{0}$ and $\frac{y}{0}$ are undefined. Such non-zero vectors are used to represent the points at infinity, with each corresponding to a particular direction.

Similarly to points, lines in the projective plane can also be modelled as non-zero $(n+1)$-dimensional real vectors. A point lies on a line if the dot-product of the corresponding vectors is zero. It can be easily seen that the line $k \cdot (0, 0, 1), k \neq= 0$ contains all the points at infinity and is therefore termed *the line at infinity*. In higher dimensions, lines generalize to planes and hyper-planes. The question of representing lines in $\mathbb{P}^3$ is more subtle; we refer the reader to an overview of different approaches in [7].

From the perspective of computer vision, the most important property of projective geometry is that it allows us to express the projective camera using linear algebra. This makes it possible to build on the grounds of this well-understood area.

Projective camera represents a model of central perspective projection. It maps points from $\mathbb{P}^3$ (world) to $\mathbb{P}^2$ (image). In Figure 2.4 we can see the camera geometry. There, the *camera centre* is positioned at the coordinate origin, while the image plane is in front of the camera. The line perpendicular to the *image plane* with one end-point in the centre is called *the principal axis* of the camera and its intersection with the image plane is *the principal point*. The world point is projected by intersecting a ray going through the camera center and the world point with the image plane. This mapping from $\mathbb{P}^3$ to $\mathbb{P}^2$ can be algebraically expressed as multiplication with a real 3×4 matrix P. Note that the overall scale of the matrix does not affect the result. Thus, the matrices $k \cdot \mathsf{P}, k \neq 0$ represent the same camera.

The simple algebraic model of the action of the projective camera on the world points is the main strength of concept. However, it cannot model some properties of physical cameras applied in practice. Most importantly, non-linear distortion of the image (e.g., barell distortion) cannot be modeled by a projective camera.

2.4.1 Epipolar geometry

Epipolar geometry expresses the geometric constraints encountered when observing a 3D scene by a pair of cameras with known parameters. Suppose we choose

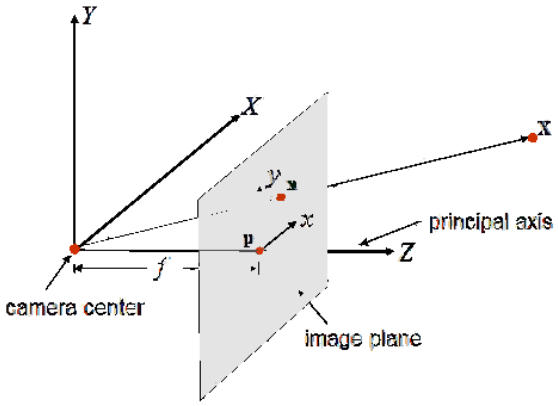

Figure 2.1: (*TODO: tento obrazek jsem stahla z googlu, mel by se nahradit.*)

a point $\mathbf{x}$ on the image of the first camera from such a pair. The knowledge of its position constraints the position of the corresponding one on the second image. This point, denoted by $\mathbf{x}'$, lies on the line $\mathsf{F}\mathbf{x}$, where F is the fundamental matrix. This line is called *the epipolar line*.

Definition 7. *The fundametal matrix* F *for a pair of stereo images is a* 3×3 *matrix which satisfies*

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

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

Fundamental matrix for a pair of photos can be estimated from the knowledge of a set of corresponding points since each correspondence is a linear constraint on the values of F. Because F has 9 real entries and is determined only up to a scaling factor, it has 8 degrees of freedom and thus at least 8 correspondences are required to determine its values. However, in our case, the matrix is going have only one degree of freedom due to the restrictions we have specified for the input photographs.

2.5 Integral images

An integral image, also known as *summed area table*, allows fast and efficient computation of a sum of image intesity 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 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 2.5).

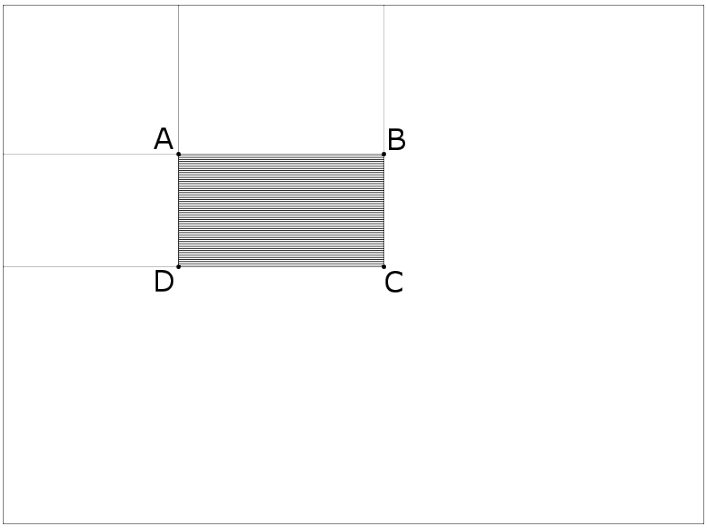

Figure 2.2: The sum of any rectangular region can be calculated by only three additions: $\sum = C - D - B + A$.

3. Image Processing Algorithms

In this chapter, we introduce the image processing algorithms often applied in software solutions with functionality similar to those discussed in Chapter 1. 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* algorithms. 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 [13] and SURF [4] feature detectors/descriptors – and then discuss the Lucas-Kanade method for computing optical flow.

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 elements are 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 Euclidean metric. The SIFT descriptor is also partially invariant to illumination changes and to affine transformations, the latter enabling us to perform matching of images capturing the scene from slightly different viewpoints.

3.1 Scale-Invariant Feature Transform

The scale-invariant feature transform (SIFT) was introduced by Lowe in [13], building on a previous work by Lindeberg [12]. 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. It is able to detect objects in images even in the presence of clutter.

Let us outline the major steps in which the algorithm proceeds in its task to repeatably generate a set of features in an image and to compute their corresponding descriptors. Recall that an imporant and desirable property of these descriptors is their invariance to image transformations like rotation, scaling or illumination changes. Firstly, candidate features are detected over all scales of the grayscale version of the input image. Locations of these features are then determined with subpixel precision. Simultaneously, features corresponding to unstable image regions are filtered out from the dataset. In the next stage, each feature is assigned its orientation and scale based on the prevailing gradient directions in the image neighbourhood. This orientation is used to establish a reference

frame for the construction of the descriptor. The last step is the feature description. The SIFT feature descriptor is a 128-dimensional vector determined by the gradients around the feature at the corresponding scale. 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 but only partially invariant to affine transformations and illumination changes. We now proceed to describe both the detection and the descriptor construction steps in more detail.

The first step in the feature detection part of the algorithm is to identify candidate locations of all scales in a highly repeatable manner. For this, Gaussian scale-space is employed. Recall its definition from Section 2.2.1:

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

where σ is the deviation of the Gaussian function, $*$ the convolution operator, $f(x, y)$ is the input image, and $g(x, y)$ the Gaussian function

$$g(x, y, \sigma) = \left(\frac{1}{\sqrt{2\pi\sigma_1^2}} \cdot e^{-\frac{x^2+y^2}{2\sigma_1^2}} \right).$$

To extract the features in the scale-space we detect the local extrema in the difference-of-Gaussian function $D(x, t, \sigma)$. The difference-of-Gaussian (DoG) is defined as the difference of two neighbouring images of the scale-space:

$$D(x, y, \sigma) = L(x, y; 2\sigma) - L(x, y; \sigma) = (g(x, y, 2\sigma) - g(x, y, \sigma)) * f(x, y).$$

Each sample point of the DoG is compared to its eight neighbours in the current scale and nine neighbours on the levels below and above. If it is larger than all of these neighbours, it is selected as a maximum; if it is smaller than all of them, it is selected as a minimum. It can be shown that the DoG approximates the Laplacian of the image at the particular location. These extrema should therefore correspond to "blobs" in the image. The application of scale-space ensures that blobs of all possible sizes are detected. These locations are taken as an initial set of candidate feature.

We need to filter this set significantly to ensure high repeatability of the resulting set of features. Usually, the detection of scale-space extrema results is a large number of candidate keypoints. However, some of them are local extrema resulting from image noise or the responses of the operator along edges. Such structures are unstable and should be discarded from the set of candidate keypoints. To reject such unstable features, we fit a 3D quadratic function to the local sample points and interpolate the location of the extremum. This is done using quadratic Taylor expansion of the function $D(x, y, \sigma)$ around the detected point. By calculating the minimum we get the position of the local extremum with a subpixel precission. Properties of this quadratic function are used to reject unstable extrema: if the function is too shallow, we discard the feature as a result of image noise; if it is too narrow, the extremum likely corresponds to an edge in the image. Otherwise we accept it as a feature.

To achieve invariance to rotation, local orientation must be determined for each feature. First, we compute a histogram of orientations of image gradients

around the its location. The peaks of the histogram represent the dominant orientations. The orientation corresponding to the highest peak is 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.

The descriptor is then constructed by considering several histograms in the four quadrants of the reference frame of the keypoint. Typically, each quadrant is divided into four subregions. In each of the resulting sixteen subregions, we construct a histogram of orientations of gradients sampled regularly in the subregion. Concatenating the values in these histograms 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 Bay et al. [4]. 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 descrease 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 σ 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}

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

(*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 – stejne jako v podkapitole 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: mozna nekde rict, jak presne se to aproximuje, nejlip obrazkem; asi by to mohlo byt v sekci o integralnim obrazu.*) 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 the Hessian matrix corresponds to the Laplacian of Gaussians (LoG) (*TODO: meli bychom rict LoG ceho.*). If we denote the results of applying the above-mentioned box filters as D_{xx}, D_{yy} and D_{xy}, we get:

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

(*TODO: porad neni uplne dobre definovno, co je to ten box filter; idealni by bylo pridat to do sekce o integralnim obrazu.*)

Similarly to SIFT, the SURF descriptor is a vector describing the distribution of intensity values within the neighbourhood of the keypoint. The difference lies mainly in the fact that the SURF descriptor is constructed using Haar wavelet responses around the point of interest. 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 chvili 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.

The local orientation and scale is again used to obtain a reference frame ensuring that the resulting descriptor is not affected by rotation and scaling of the input image. The size of the inspected region around the keypoint is set to $20s$, where s is the scale where it was detected. (*TODO: nemelo by s byt nejak svazane s rozptylem σ?.*) We split the region into 4×4 regular square subregions and for each of them we compute Haar wavelet responses d_x and d_y in a 5×5 grid. (*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 konkretnejsim terminem; navic mi neni jasne, jaky je rozdil mezi temi box filtery a Haar wavelety.*) 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 4×4 subregions gives 64-dimensional vector descriptor.

Matching the SURF descriptors between two images can be speeded-up by including the trace of the 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: odkaz na nejakou studii porovnavajici ruzne detektory/deskriptory.*) The work [4] highlights the fast computation of the 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 element enabling this speed-up are the approximations using integral images. Also, the SURF descriptor is a 64-dimensional vector, enabling faster computation of the Euclidean distance between the descriptors compared to the 128-dimensional SIFT

descriptors. Furthermore, due to the global integration of the SURF-descriptor, it stays more robust to various perturbations than the SIFT descriptor. (*TODO: potrebujeme napsat presneji, zejmena by se hodil odkaz na to, odkad to tvrzeni je cerpano.*)

SURF offers invariance to rotation and limited levels of 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 Lucas-Kanade algorithm for optical flow

Optical flow algorithms are typically applied in postprocessing software for movies to track an object through a video sequence. We can think of this problem as the image correspondence problem optimized for the situation where there is very little camera movement between the registered pair of photos, as is in the case with two consecutive frames of a video sequence. The output of such algorithm is an array of 2-dimensional vectors $\mathbf{d_x}$, where $\mathbf{x}$ is an image position, with the property that the point $\mathbf{x}$ on the first frame corresponds to the point $\mathbf{x} + \mathbf{d_x}$ on the second one. Optical flow is typically not constrained by epipolar geometry, since the assumption is that the video is of a dynamic scene that includes objects moving relative to the camera and each other. The algorithms for optical flow tend to work only when the differences between the images are significantly limited and the registration can be performed only on a small neighbourhood of the original position of a feature.

We now describe a classical Lucas-Kanade method [14] for calculating the optical flow between two images I_1 and I_2.

The method proceeds by specifying linear constraints on the values $\mathbf{d_x}$ and then solves the resulting over-determined system in the least squares sense. For each pixel $\mathbf{x}$ that is being tracked, we consider the following equation:

$$\Delta I_1(\mathbf{x}) \cdot \mathbf{d_x} = I_1(\mathbf{x}) - I_2(\mathbf{x}), \tag{3.1}$$

where $\Delta I_1(\mathbf{x})$ is the image gradient at $\mathbf{x}$. This is motivated by the observation that the apparent movement should be roughly in the direction of the image gradient for locations with a large jump of the intensity value between the frames. On the other hand, when the intensity of a pixel is approximately the same in both frames, we either expect no movement or just a movement perpendicular to the image gradient. However, the inclusion of just the above equations does not enforce spatial consistency of the movement in any way. This is taken into account by repeating equation (3.1) for each position $\mathbf{x}^i$ taken from the neighbourhood of $\mathbf{x}$ of a fixed size. Specifically, we add the following equation for each $\mathbf{x}^i$:

$$\Delta I(\mathbf{x}^i) \cdot \mathbf{d_x} = I_1(\mathbf{x}^i) - I_2(\mathbf{x}^i).$$

Note that the differences are read from the position $\mathbf{x}^i$ while the movement constrained is of the pixel $\mathbf{x}$.

Solving the above over-determined system using conventional methods gives us the desired values $\mathbf{d_x}$.

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. HTC Dream, the first phone with the Android operating system, was sold in autumn 2008. Majority of Android's code is released under a free license. This allows developers to modify and (with certain exceptions) freely distribute the software. Due to this, Android became the most popular operating system for smartphones. In this chapter, we introduce this system from programmer's perspective and give a brief description of how to implement an Android application. The purpose of the chapter is to highlight the main architectural differences of Android and conventional desktop platforms rather then providing a complete programer's manual.

4.1 Introduction to Android development

Android is a product of *The Open Handset Alliance*, which develops open standards for mobile devices and is led by Google. The Android platform is a layered environment built on the foundation of the Linux kernel. It includes a user interface library featuring various types of elements (views, windows, display boxes, lists, etc.), an embedded web browser, and support for OpenGL ES. Most Android-powered devices have built-in sensors to measure, e.g., motion, orientation or temperature. These include an accelerometer, a gyroscope or a barometer. It also provides an array of connectivity options, for example Wi-Fi, Bluetooth or cellular data. A built-in camera support is also included and most Android handsets indeed feature a camera. 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.

Linux kernel, the first architectural layer of the platform, is used for memory and process management, device drivers, and networking. Above this, we have the native libraries, which include graphics support, media codecs, a SQLite and WebKit. These are written in C or C++ and called through a Java interface. The actual applications are running on the Dalvik Virtual Machine (DVM), an implementation of the Java Virtual Machine optimized for low processing power and memory resources. The Android Runtime Layer consists of this DVM and the core Java libraries.

The next level is the Application Framework, which manages functions fundamental to running applications. Its major part is formed by the Activity Manager (managing the life cycle of the 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 final layer is formed by the individual applications. Some of them are preinstalled and provided by Google while the rest are third party applications created by the community. The structure of the system is illustrated in Figure 4.1 below.

Android applications are written in the Java programming language and run

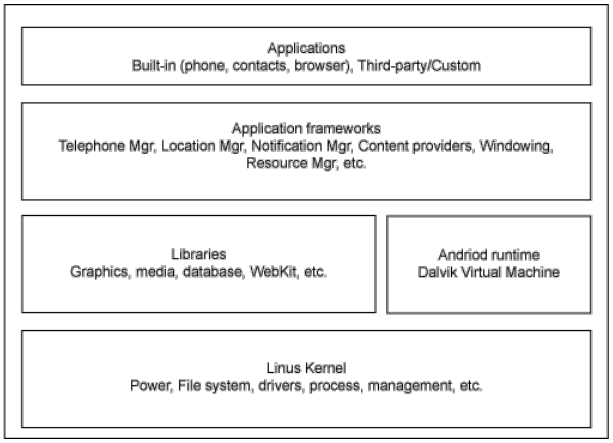

Figure 4.1: The architecture of the system.

within an instance of the Dalvik Virtual Machine. A key part of the application is the AndroidManifest.xml file containing installation meta-data, including the permissions needed to run. An example of such permission is the ability to use the phone's camera or to access the Internet. We concentrate more on activities and their development in the following sections. (*TODO: O aktivitach jsem zatim nic nerekli. Proc ted rikame ctenari, ze se na ne vic zamerime v dalsi sekci?.*)

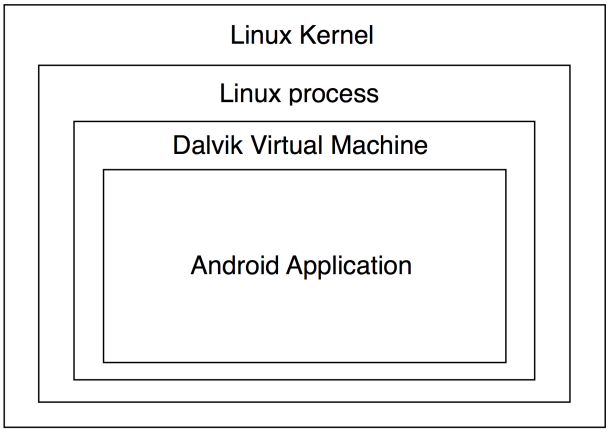

Figure 4.2: A typical Android application runs within an instance of the Dalvik Virtual Machine.

The tools required for the development of an Android application are the Android Software Development Kit (SDK) and a Java IDE, such as Eclipse[1] or the Android Studio[2] based on IntelliJ IDEA. The Android SDK contains an Eclipse plugin supporting Android developement, documentation, sample code, and various tools, such as Android Debug Bridge used by the programmer to communicate with an application running on a device.

[1] `http://www.eclipse.org`
[2] `http://developer.android.com/sdk/installing/studio.html`

4.2 Application components

When an application is installed to a device, it lives within 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 assigns a user ID number. Android is a multi-user operating system, meaning that each application corresponds to a different user with a different ID. This number is known only to the system that sets permissions to all files so they cannot be accessed by any other applications. Each application runs in a unique Linux process initiated when the application or any of its components need 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 is governed by the "oldest first" principle – the system starts by killing the apps that were inactive for the longest time.

The security is driven by the principle of least privilege and the application is allowed access only to the resources specified in the manifest file. This protects the whole Android system and the applications from each other.

A typical application is built from several components. These basic building blocks can be divided into four different groups:

- activities,
- services,
- content providers,
- broadcast receivers.

An activity is can be thought of as a UI screen providing elements such as views, lists, buttons, labels, etc. along with the implementation of their functionality. The layout of an activity and the widgets placed in the window is described in a separate XML file (we reveal more about user interface later) (*TODO: Tu zavorku rozhodne nahradme odkazem na sekci.*) Most applications consist of more than one activity. 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 aktivity s tim, ze aplikace prechazi mezi nejakymi stavy? V dalsim se pak hovori stavech aktivit 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 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

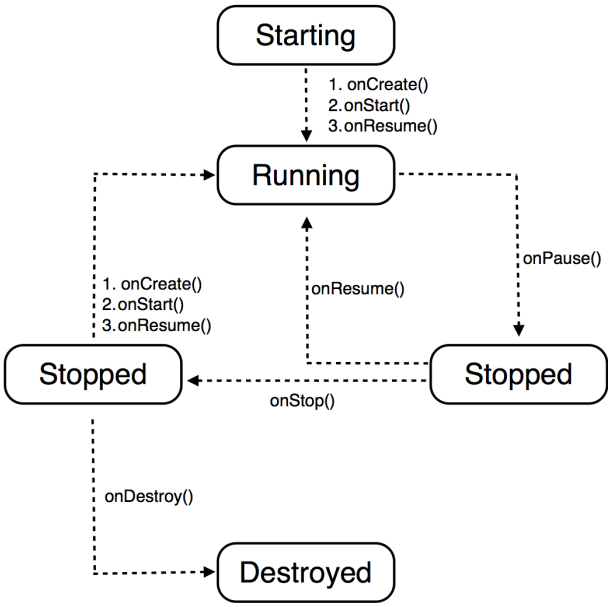

Figure 4.3: Life cycles of an application.

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 a 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 4.4. 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
                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 an height by using *android:layoutwidth* and *android:layoutwidth* 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 4.3) 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 5.2. 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

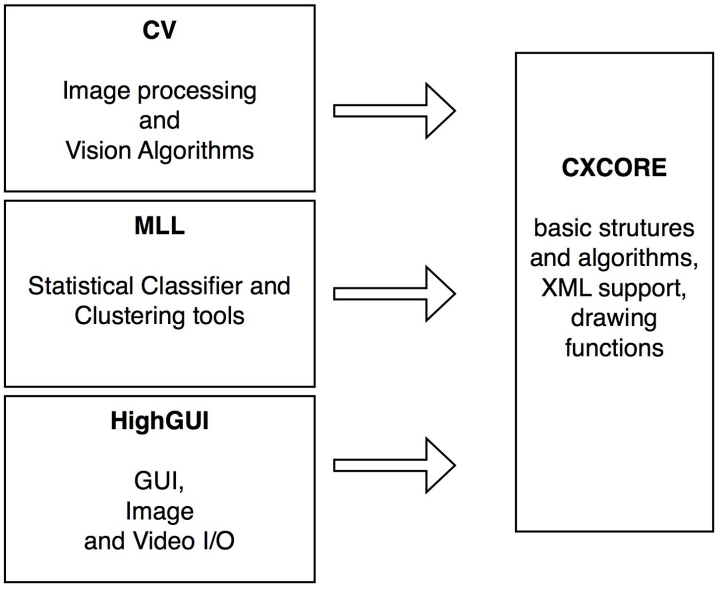

Figure 5.1: The basic structure of OpenCV library.

(*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 3):

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

In the image processing package are functions for image scaling (`Imgproc.pyrDown()`, `Imgproc.pyrUp()`), bluring 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 [8].

6. Implementation

6.1

6.2

Conclusion

Bibliography

[1] R. Szeliski. *Computer Vision: Algorithms and Applications.* Springer, 2010.

[2] Noah Snavely and Rahul Garg and Steven M. Seitz and Richard Szeliski. *Finding Paths through the World's Photos.*

[3] *The 44^{th} President Inauguration.* CNN, Inc., 2013. Accessible from `http://www.cnn.com/themoment`, retrieved on 2013-07-01.

[4] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. *Speeded–Up Robust Features (SURF).* In ECCV, 2006.

[5] M. A. Fischler, and R. C. Bolles. *Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography.* In Communications of the ACM, 24(6): 381-395, 1981.

[6] C. Harris and M. Stephens. *A combined corner and edge detector.* In Proceedings of the Alvey Vision Conference, pages 147 – 151, 1988.

[7] R. Hartley, A. Zisserman. *Multiple View Geometry in Computer Vision.* Cambridge University Press, 2003.

[8] H. Hirschmüller *Stereo Processing by Semiglobal Matching and Mutual Information* Pattern Analysis and Machine Intelligence, 2008. IEEE Transactions on.

[9] IDC Worldwide Mobile Phone Tracker, 2013. IDC, 2013.

[10] F. Jurie and C. Schmid. *Scale–invariant shape features for recognition of object categories.* In CVPR, volume II, pages 90 – 96, 2004.

[11] T. Kadir and M. Brady. *Scale, saliency and image description.* IJCV, 45(2):83 – 105, 2001.

[12] T. Lindeberg. *Feature detection with automatic scale selection.* In International Journal of Computer Vision, 30(2):79 – 116, 1998.

[13] D. Lowe. *Object recognition from scale–invariant features.* In CCV, 1999.

[14] B. D. Lucas and T. Kanade. *An iterative image registration technique with an application to stereo vision.* In Proceedings of Imaging Understanding Workshop, 121–130, 1981.

[15] S. Roy. *Stereo Without Epipolar Lines: A Maximum-Flow Formulation.* In International Journal of Computer Vision, 1999.

[16] N. Snavely; S. M. Seitz; R. Szeliski. *Modeling the world from Internet photo collections.* In International Journal of Computer Vision, 2007.

[17] 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.

[18] 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.