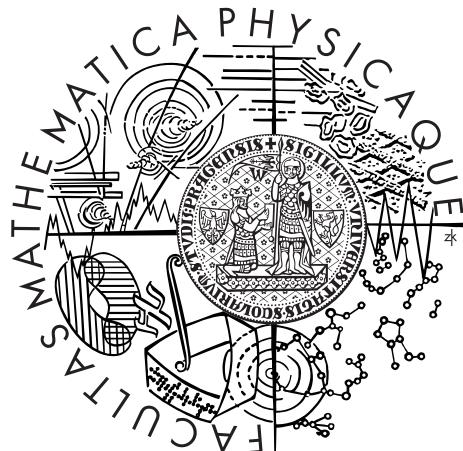


Charles University in Prague
Faculty of Mathematics and Physics

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

Contents

Introduction	3
1 Overview	5
1.1 Available software packages	5
1.1.1 Matchmoving software	5
1.1.2 Microsoft PhotoSynth	5
1.1.3 Autodesk 123D	6
1.2 Available libraries	6
1.2.1 OpenCV	6
1.2.2 OpenGL	8
1.3 Problem statement	8
2 Basic Notions	9
2.1 Convolution	9
2.2 Image derivatives	9
2.2.1 Gaussian image derivatives and scale-space	9
2.2.2 Laplacian	10
2.2.3 Hessian matrix	10
2.3 Basic image similarity metrics	11
2.4 Projective geometry	12
2.4.1 Epipolar geometry	13
2.5 Integral images	14
3 Image Processing Algorithms	15
3.1 Scale-Invariant Feature Transform	15
3.2 Speeded-Up Robust Features	17
3.3 SIFT vs. SURF	18
3.4 Lucas-Kanade algorithm for optical flow	19
4 Developing for Android	20
4.1 Introduction to the Android development	20
4.2 Application components	22
4.3 The Activity Lifecycle	23
4.4 User Interface	24
4.5 Sensors	25
4.6 Data Storage	25
4.7 An Example	26
5 Implementation	30
5.1 Implementation outline	30

6 Evaluation	35
6.1 Results	35
Conclusion	38
6.2 Future work	38
Bibliography	39

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 [20].

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-intensive 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 5).

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 Boujou¹ and Autodesk Matchmover². The opensource libmv project³ aims to add matchmoving capabilities to the Blender 3D modelling application⁴.

1.1.2 Microsoft PhotoSynth

Microsoft PhotoSynth, based on a research by Snavely, et al. [20], has been publicly released in 2008. The software solution processes a set of unorganized

¹<http://www.2d3.com>

²<http://www.autodesk.com>

³<http://code.google.com/p/libmv/>

⁴<http://www.blender.org>

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 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 [18, 19]. 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

OpenCV⁵ 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

⁵<http://opencv.org>

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* library⁶, which actually provides access to a wide range of computer graphics libraries (e.g., FFmpeg, OpenKinect, ...).

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

⁶<https://code.google.com/p/javacv/>

of the library, we have decided to choose this version for the final version of our project.

1.2.2 OpenGL

OpenGL⁷ 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.

1.3 Problem statement

Our application's objective is to provide the user the possibility 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 foresee 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.

⁷<http://www.opengl.org/>

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 referred to the monographs [1, 7] for further details.

2.1 Convolution

Convolution is an operation on two functions often encountered 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 measure spatial 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

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

where I is the input image and (x, y) are coordinates of the pixel. 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 [15] 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 $f : \mathbf{R}^n \rightarrow \mathbf{R}$ 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.

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 measure 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 measure 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 measure 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. The *camera centre* is positioned at the origin of the coordinate frame, while the image plane is in front of the camera. The line perpendicular to the *image plane* going through the camera 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 \mathbf{P} . Note that the overall scale of the matrix does not affect the result. Thus, the matrices $k \cdot \mathbf{P}, k \neq 0$ 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., barrel distortion) cannot be modeled by a projective camera.

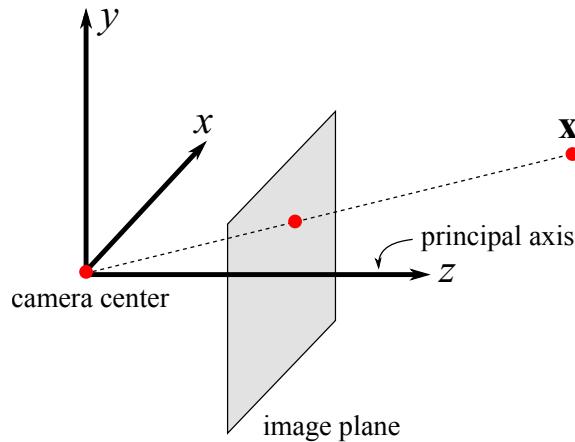


Figure 2.1: Perspective projection of a point \mathbf{x} on the image plane of a camera with camera center in the coordinate system's origin.

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

Definition 7. *The fundamental matrix \mathbf{F} for a pair of stereo images is a 3×3 matrix which satisfies*

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

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 \mathbf{F} . Because \mathbf{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 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 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).

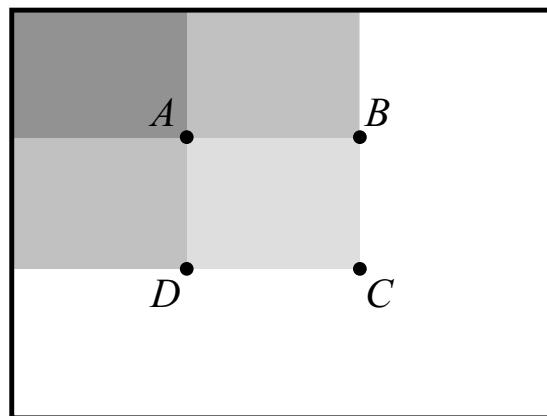


Figure 2.2: The sum of any rectangular region can be calculated by only three additions: $C - D - B + A$.

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 [15] 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 [15], building on a previous work by Lindeberg [14]. 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 important 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 in 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 precision. 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 Hessian-matrix approximation and the computation of Haar wavelet responses. Similarly to SIFT, features 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 instead employed to decrease the computation time.

The SURF feature detection is again based on detecting blob structures in the image. Similarly to the SIFT algorithm, it estimates the Hessian matrix of the convolution of the image with the second derivative of the Gaussian. The determinant of this matrix expresses the local change around the area. The points that are simultaneously local extrema of both the determinant and the trace of the Hessian matrix are chosen as candidate keypoints. However, the convolution is a very costly in terms of calculation time. It is approximated and speeded-up using the integral images (see Section ??). The convolution is performed using a kernel that approximates the second derivative Gaussian kernel reasonably well, yet it is possible to evaluate the operation using a limited number of queries to the integral image structure. Since these queries can be fulfilled in $O(1)$ time, this results in a relatively fast runtime. Another advantage of this is that the SURF algorithm does not need to build the scale-space by explicitly blurring and

down-sampling the images.

As in the case of the SIFT algorithm, 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. Recall that the Haar wavelet responses can be calculated efficiently using integral images since they correspond to convolutions with kernels containing predominantly rectangular structures. Before the descriptor calculation, in order to achieve invariancy with respect to image rotation, it is necessary to determine the local orientation of the feature in a manner similar to the SIFT algorithm.

The SURF descriptor describes the surrounding area of the feature. This area is divided to 4×4 subareas. To each of these subareas, the Haar wavelet responses are calculated in the x and y directions and then are described by the vector

$$v = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|),$$

where d_x and d_y are the wavelet responses in x and y directions. Concatenating this for all 4×4 subregions gives a 64-dimensional vector descriptor.

Matching the SURF descriptors between two images can be accelerated by including the trace of the Hessian (i.e., the Laplacian) to the process of finding the corresponding feature. We exploit the sign of the Laplacian, which effectively distinguishes bright blobs on dark background from the reverse situation. In the matching stage we compare only features with the same sign. 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. The work [4] highlights the fast computation of the descriptor. The main contributor enabling this speed-up is the approximation by 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.

The results of comparing and testing SURF against SIFT can be found in [13] and [10]. In the former study, the robustness of SIFT and SURF was tested under several different conditions. It has been shown that SIFT and SURF give similar results robustness to rotation, changes of illumination, and noise (the performance of both SIFT and SURF rapidly decreased on the images with “salt and pepper” noise). When testing matching photos taken from different viewpoints, the SIFT algorithm outperformed SURF.

According to the findings in [10], SIFT still outperforms SURF in scale invariance and rotational invariance. It is also less susceptible to image blur. However, it was found to be significantly slower. Furthermore, SURF gives better results under illumination changes. This shows that none of these detectors/descriptors outperforms the other in all situations and the choice of the method depends on the intended application.

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 x is an image position, with the property that the point x on the first frame corresponds to the point $x + \mathbf{d}_x$ 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 [16] 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 x that is being tracked, we consider the following equation:

$$\Delta I_1(x) \cdot \mathbf{d}_x = I_1(x) - I_2(x), \quad (3.1)$$

where $\Delta I_1(x)$ is the image gradient at 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 x^i taken from the neighbourhood of x of a fixed size. Specifically, we add the following equation for each x^i :

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

Note that the differences are read from the position x^i while the movement constrained is of the pixel 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 between Android and conventional desktop platforms rather than providing a complete programmer's manual.

4.1 Introduction to the Android development

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, 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.

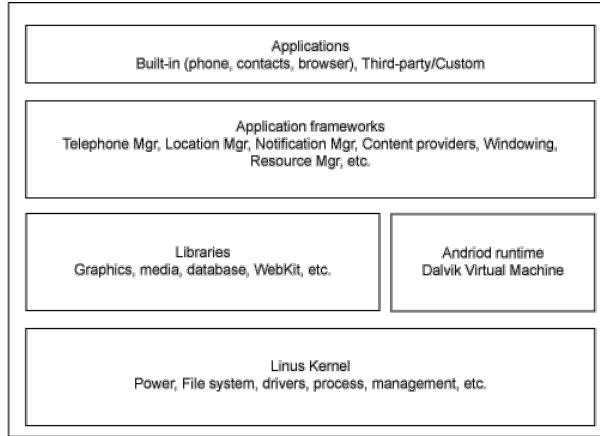


Figure 4.1: The architecture of the system.

Android applications are written in the Java programming language and run 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 necessary permissions. An example of such permission is the ability to use the phone's camera or to access the Internet.

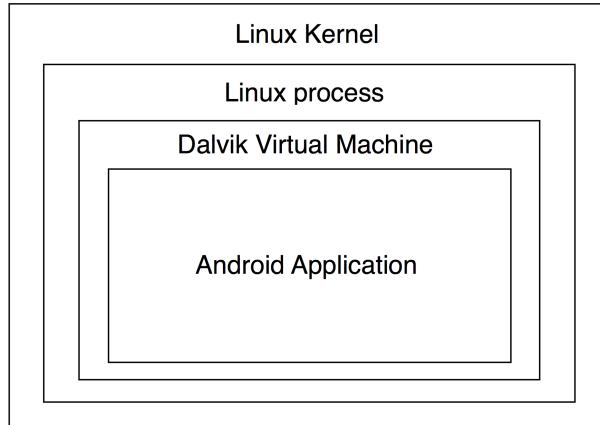


Figure 4.2: A typical Android application runs within an instance of the Dalvik Virtual Machine.

The tools required for the development of an Android application are the Android Software Development Kit (SDK)¹ and a Java IDE, such as Eclipse² or the Android Studio³ based on IntelliJ IDEA. The Android SDK contains an Eclipse plugin supporting Android development, documentation, sample code,

¹<http://developer.android.com/sdk/>

²<http://www.eclipse.org>

³<http://developer.android.com/sdk/installing/studio.html>

and various tools, such as the Android Debug Bridge used by the programmer to communicate with an application running on a device or an emulator.

4.2 Application components

Compared to the situation on the majority of conventional platforms, Android interacts with the running applications in a much closer manner. Actually, the system expects a certain specific architectural structure of the application expressed using concepts from object oriented programming. A typical Android application is decomposed into several classes, each being a descendant of a particular class provided by the SDK. Through this, requirements on what and how the application's class has to implement are imposed. This approach allows, e.g., sophisticated interaction between different applications running on the same phone. For example, one application can use a list of contacts maintained by another application without the developers of the latter having to set up any protocols or data exports – they only need to obey the software development principles of the platform. Thanks to this, the user data are seamlessly protected from an unauthorized access by the system through the principle of least privilege. In the above-mentioned example, the access to the list of contacts would not be granted unless it is listed in the accessing application's manifest file and this requirement has been authorized by the user. Since these properties of the platform are rather unique, we devote this section to discuss them.

An Android application is built from several components typically belonging to one of the following groups:

- activities,
- services,
- content providers, etc.

An activity 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. Most applications consist of more than one activity with one being designated as the main one. 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. From the developer's perspective, activities are represented by classes descending from the Activity class defined in the SDK.

A service is a component used to perform operations in the background. For example, if the application needs to run a long-term computation, the user can switch to another application with the computation continues.

A content provider manages sharing data across applications. When the app is storing data, for example in the file system or an SQLite database, the content

provider interface can be used to access or even modify the data from other applications.

The platform utilizes additional classes of components besides the above-mentioned ones (e.g., broadcast receivers). However, we omitted them for the sake of brevity.

The interaction between various components and activities is facilitated by *intents*. An intent is a message to the system to invoke new activity, service or broadcast. The reader is referred to the Android SDK documentation for more information about activities, services, content providers, intents, etc.

4.3 The Activity Lifecycle

During its life cycle, an application switches between different application states. Compared to desktop platforms, the programmer has only limited control over these state transitions. On the desktop computer, the developer has a certain level of control over, e.g., minimizing or closing a window of an application or quitting the software. This cannot be affected on the discussed platform. The life cycle is a collection of functions the operating system executes on the application during its runtime.

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 main activity class's *onCreate()* method is called and eventually the application transitions into the running state. When in a running state, the application is actually on the screen visible to the user and handling all user interactions such as typing or touching the screen. An activity in this state has the highest priority for memory allocation in the activity stack. It is 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 Android does not destroy every activity when it gets to the background, because it is probable that it is going to be used again.

The application is paused (in a paused state) when not interacting with user but is still visible on the screen. This state does not occur very often, since most

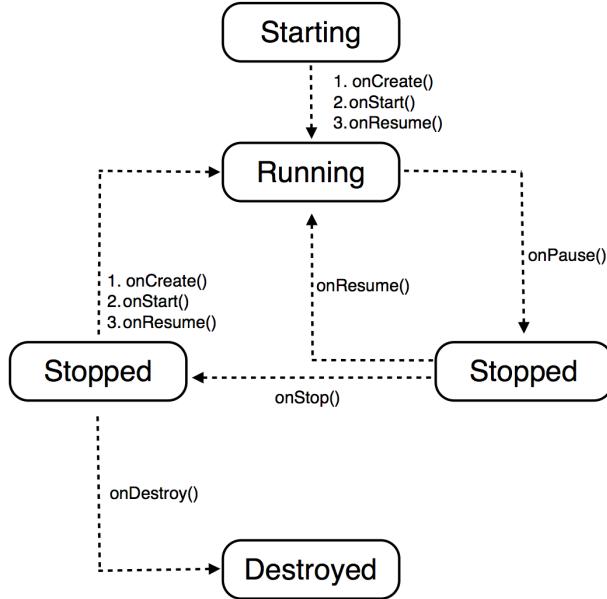


Figure 4.3: Life cycles of an application.

applications cover entire screen. However, in the cases when the application is partly visible yet not interacting with the user, the pause state is utilized.

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 and thus removed from the memory.

4.4 User Interface

All elements of the application's user interface are built using a hierarchy of *View* and *ViewGroup* objects. The layout describing the visual structure of the *Views* is declared in a separate XML file. Although not recommended, the user interface can be also declared explicitly in the source code. There are two principal layouting options: *LinearLayout* arranges the inner elements in a single row or a column, while *RelativeLayout* positions the elements in relation to their parent or each other.

With the layout defined, the developer can then utilize the UI elements offered by the SDK, such as *menus*, *action bars*, *dialogs* or *toasts*. There are several types of *menus* such as *options menu* or *context menu*. The *options menu* displays a collection of buttons when the user taps the menu button typically available on an Android device. From the Android version 3.0 higher, the options menu items are included in the action bar after touching the action overflow button. The *context menu* is a floating menu associated with a particular element in a view displayed after touching the element.

Action bars provide a way for navigating through the application's interface. 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. *Action Bars* are superior to the *options menus* since an *Action Bar* is always visible while *options menu* shows only on the user request.

A *Dialog* is a small window informing the user about some action. Unlike *Toast* it usually requires the users action before continuing 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 built-in motion, environmental and positional sensors. Motion sensors are sensors measuring the rotation along three axes and give us an opportunity to detect motion, shaking or tilting of the device. Environmental sensors are applied for measuring temperature, illumination, or humidity. Built-in magnetometers and orientation sensors provide information about the current position and/or orientation of a device and can be used in applications for GPS navigation or in mapping software. The sensor events are access using an instance of the *SensorManager* class.

4.6 Data Storage

There are several possibilities for storing application data in Android. The internal storage is preferable when we do not want other applications to access them. The read and write operations are performed using instances of *FileInputStream* and *FileOutputStream*.

However, media files such as pictures, video files or audio records are expected to be shared and accessible from other applications. In such a case, the external storage is utilized through methods of the *Environment* class. The function *getExternalStoragePublicDirectory()* depending on the supplied argument 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 directory we want to access, for example *Environment.DIRECTORY_MUSIC*.

4.7 An Example

In this section, we illustrate the concepts from the above sections on an example application. As noted in the beginning of the chapter, this is not intended to be an all-encompassing description of creating an Android application. Rather, it is supposed to illustrate crucial parts the process.

A user interface, discussed in Section 4.4, is typically specified using an XML file stored in the `/res/layout` directory. Below is a simple example of such file:

```
<?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 a *RelativeLayout* element which allows us to describe a relative position with respect to the parent. Inside the *RelativeLayout*, we find *org.opencv.android.NativeCameraView* which is an element of the OpenCV library used for camera view and another *RelativeLayout* with *Buttons*, a *ToggleButton* and a *TextView*. All of the elements have their width and height specified by using the *android:layoutwidth* and *android:layoutwidth* attributes. The value of these attributes can be *matchparent* or *wrapcontent* or a specific number of units.

It is possible to define a specific layout of an activity for a particular screen orientation through supplying an alternative XML file of the same name in the *res/layout-land* folder.

In the *src* folder, we have all the .java files representing activities or classes. Every Android application has at least one “main” activity invoked by the system when the application is launched. Due to the activity lifecycle (see Section 4.3), it is necessary to implement the following methods:

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

Usually only the basic startup actions are completed in the *onCreate()* method, for example setting up the layout or initializing some of the class variables. The implementation of the *onCreate()* method in the example below first invokes the ancestor's corresponding method, then initializes the layout using the *setContentView()* method and then performs some application specific processing.

```
@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 the user interface interactive we need to assign functions to be executed when tapping on the individual elements. This can be done either in the XML file by specifying the activity's method name in an attribute of the corresponding element:

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

Alternatively, the action can be assigned in the code:

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

5. Implementation

In this chapter, we describe how we connected the computer vision algorithms described in Chapter 3 to obtain a program solving the task of 3D reconstruction on a mobile phone. The result of this work is an Android application that allows the user to take photos with a camera, browse them and pick a pair of images to reconstruct. The reconstructed image is then visualized in 3D using OpenGL ES. The properties of the input pair of photos are described in Section 1.3.

The following are the main obstacles we are faced with when trying to solve this problem:

- mobile photos typically have a limited computational power,
- they have limited operational memory,
- most importantly, the optical properties of digital cameras included in a typical mobile phone are of mediocre quality at best.

The last property is particularly troublesome for us, since it implies – among other problems – that the captured image is defomed by an unknown non-linear transformation. This implies that the model of projective camera of Section 2.4 and the epipolar constrains of Section 2.4.1 are only a rough approximation. However, this significantly restricts the choice computer vision algorithms to apply. For example, algorithms for the dense reconstruction problem are often heavily dependent on having a precise model of epipolar constrains in the image pair.

We solve this problem using a combination of sparse feature matching and the classical optical flow algorithm, both described in Chapter 3. Specifically, we employ the SURF algorithm because it has better computational efficiency compared with the SIFT algorithm. Sparse feature matching can be successfully performed without any kind of epipolar constraint and thus, the innaccuracy of the camera model is not an issue. However, the resulting 3D information is only a sparse 3D point cloud reconstruction of the scene. Since this is insufficient for our purposes we then repeat optical flow calculation on pairs of neighbourhoods of corresponding features.

In XXX we discuss this method in detail. The final section is devoted to technical aspects of the implementation.

5.1 Implementation outline

In the this section we describe the way how was the task solved, what algorithms were chosen and how was the application implemented.

To solve our task, we followed these steps:

- at first we find the initial relative position of the pair of input images,

- then we detect and match SURF keypoints from which is chosen the most robust match to specify more accurate relative position,
- then we detect larger amount of SURF keypoints. The matching process uses the information of the relative position estimated in the previous step,
- at last, we get the dense correspondences by detecting even more matches using the optical flow algorithm.

Finding the initial relative position of the pair of the images is implemented by using Sum of absolute differences (described in Section ??). At first, we create a scale-space, find the overlap of the images in the lowest scale and by upscaling specify the overlapping area more accurately. In this way the registration runs in approximately two seconds for a pair of the expected input.



Figure 5.1: Left: The result of the registration where the process of upscaling was stopped when the height of the image was over 200px. Right: The division of an image into square-boxes.

The next step of the process is detection of keypoints which are detected with the SURF detector. We divide the images into square-boxes of the same size and to each square-box we assign an array of keypoints situated in it. This saves the computational time when finding the match – instead of comparing every possible correspondence between the images we compare only those which are situated in the square-boxes located in the corresponding areas. At first we find the most robust match from only the SURF keypoints where the response is higher than 4000. Based on this match we estimate the direction of the shift which gives us more accurate relative position of the input pair of images.

In the next step we match the detected SURF keypoints one more time. For each keypoint we calculate the corresponding area of the surroundings in the shape of a rectangle oriented in the direction of the shift. To avoid mismatches we reject the matches where the matched keypoints differs too much in the orientation or if there is more than one obvious potential points for the match.

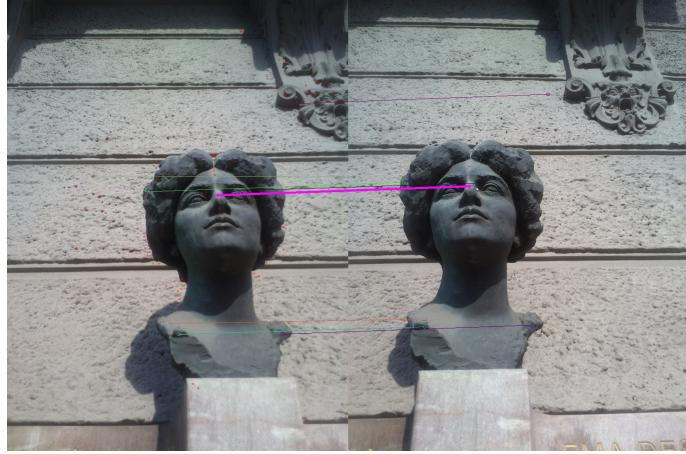


Figure 5.2: The most robust match chosen from the keypoints with response higher than 4000. According to this match the more accurate relative position of images is estimated.

At this point we have relatively robust matches for sparse correspondence. For the calculation of the depth information we need to extract more corresponding points. Assuming the difference of images is limited, we use for this purpose optical flow algorithm. For each SURF keypoint in the first image we detect corners or other features acceptable for the tracking algorithm in its 70×70 pixels area.

Calculating the optical flow we get the corresponding points in the second image. With high probability some of the results will be influenced by noise. To avoid these mismatches we calculate the variation of the distances between each match. If the variation is higher than 300 we discard all of the detected optical flow matches.

The result is visualized in OpenGL ES. Every keypoint is represented as a triangle in a space with the depth calculated from the correspondence. At first, the user can view the rough model created from the results of SURF matching. Each time some of the dense corresponding points are calculated with optical flow algorithm, the model is updated.

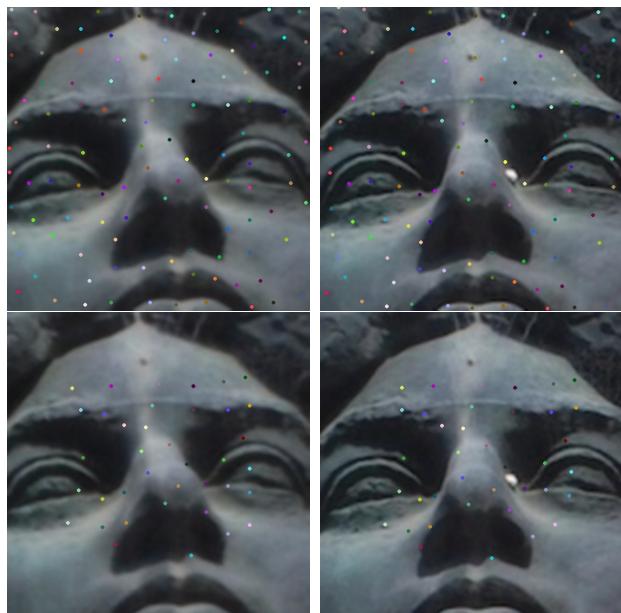


Figure 5.3: 70×70 windows of the pair of the input images with optical flow matches. In the first line all the detected correspondences are shown, in the second line are only those which were accepted after filtering.

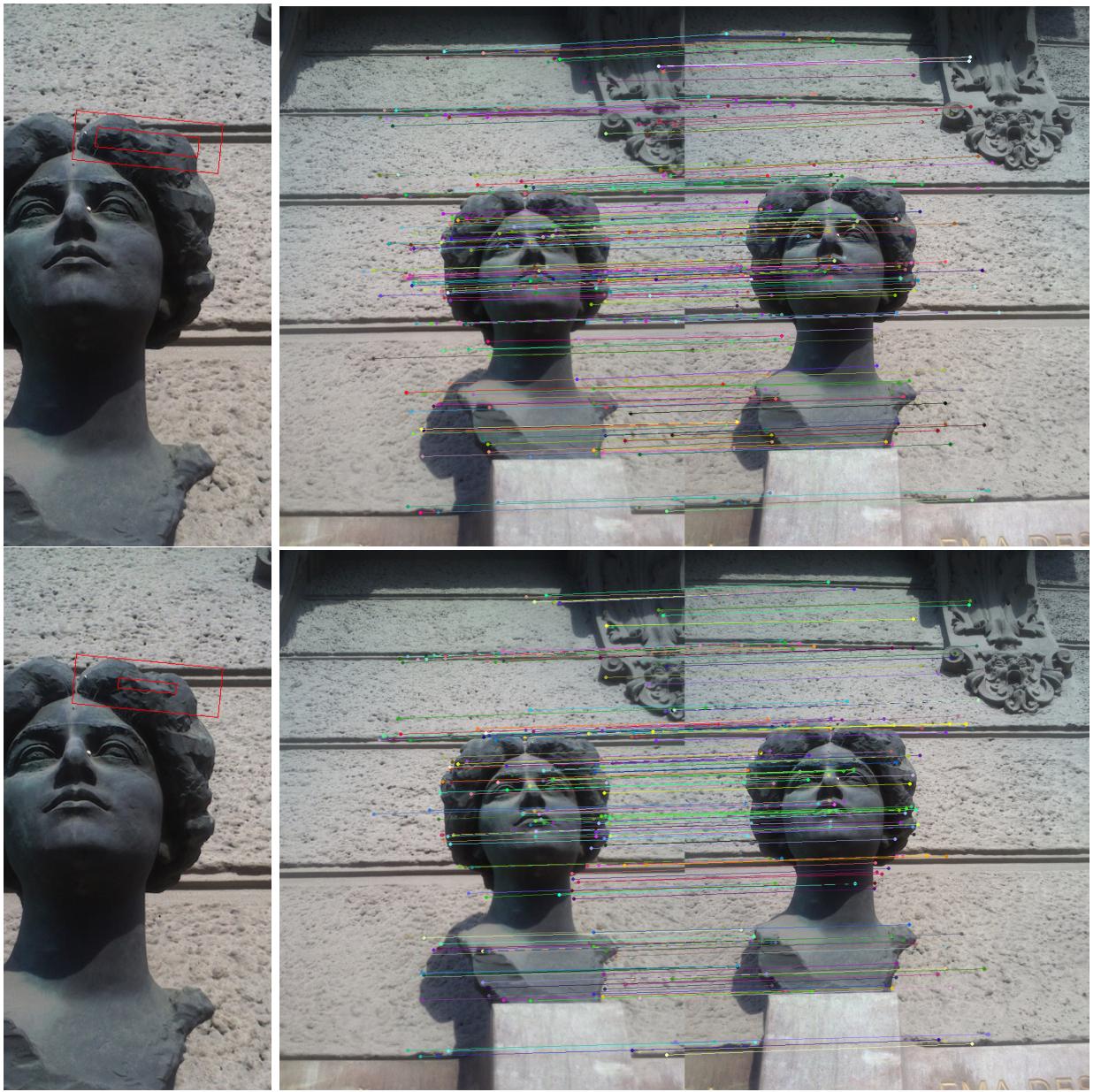


Figure 5.4: Comparison of the result of matching with different sizes of the oriented rectangles. At the top left is a sample of the oriented rectangle where the width of inner rectangle is 20% and height is 35% of the size of the large one. Next to it is the result of matching the SURF keypoints where for each keypoint in the first image is estimated this rectangle for the detection of the corresponding point in the second image. Below is the result of same matching with smaller rectangular area – the width of the inner rectangle is reduced to 10% and height to 20% of the size of the large one. We can see that in the second case there is less mismatches than in the first one.

6. Evaluation

Our sampling data for testing our application were especially highly textured images of sculptures.



Figure 6.1: An example of input data.

6.1 Results

For the input data shown in Figure 6.1 the results seem satisfying. For this kind of input the SURF detector works properly. It detects sufficient amount

of keypoints and after filtering there remain enough good matches to get the information about the depth of some parts of the input images.

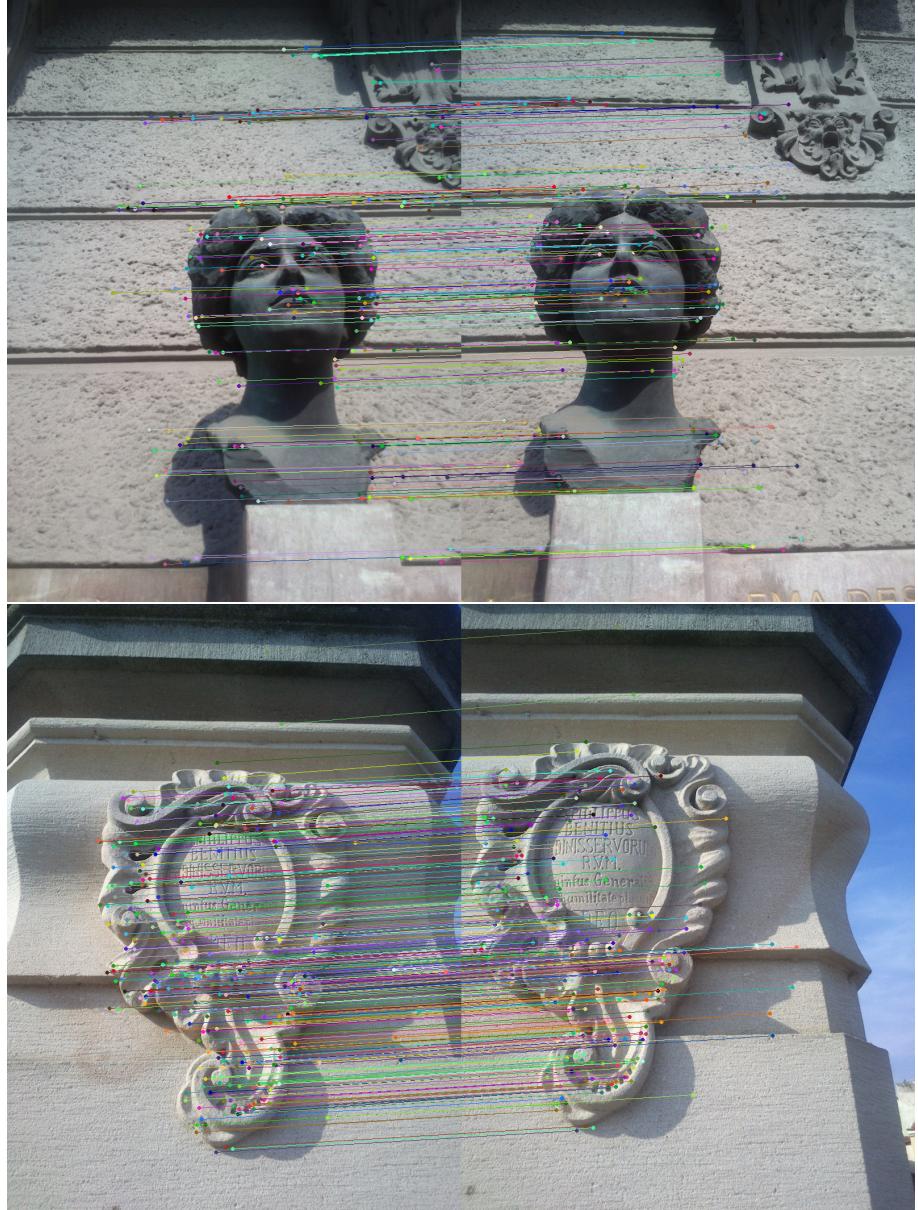


Figure 6.2: The result of SURF matching.

For the pair of the images of a bust of Ema Destinová we get a 3D model shaping the head and a part of a wall in the background. The second mentioned pair of input images gives us a result as an inclined plane in the angle of the memorial in the picture. These results are comparable with the real appearance of the scenes.

On the other hand for images with noise or images which are not highly textured, the result is usually a bunch of mismatches so the 3D model does not correspond to the reality.



Figure 6.3: The output as a 3D model of disparity. The model of Ema Destinová sculpture (left) and of a memorial (right).

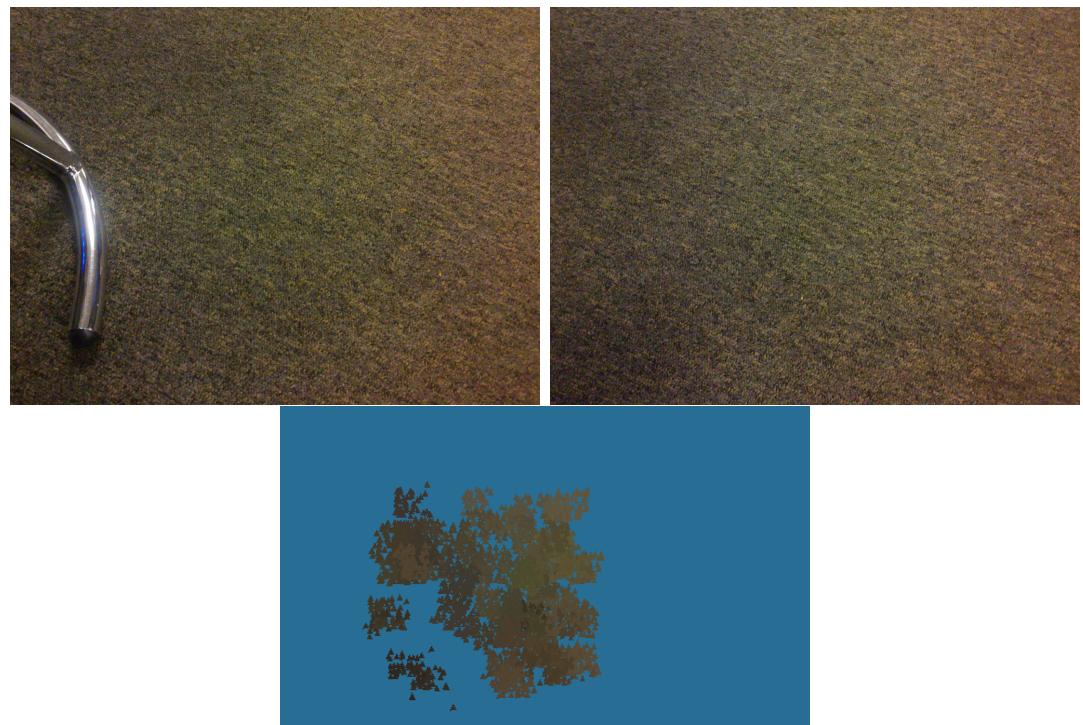


Figure 6.4: An example of an input and output of a scene that does not give a good result.

Conclusion

Our work gives satisfiable results on the highly textured input images. For the testing input data, in case of a bust of Ema Destinová we get a 3D model shaping the head and a part of a wall in the background. In the second case, the input images give us a result as an inclined plane in the angle of the memorial in the picture. These results are comparable with the real appearance of the scenes.

However, while we use a sum of absolute differences to detect the initial relative position, our approach fails on the pair of images with different illumination. Also it does not give good result on the images influenced by noise, which is very common on the images taken by mobile phones camera, especially when the conditions of the illumination are not ideal.

6.2 Future work

As we mentioned, the results on the pair of input images with different illumination does not give good results. We could solve it by using other approach e.g. the mutual information.

Our implementation could be also extended to the input of n photos. It would be necessary to find the relative position of all pairs of photos. This does not have to be calculated explicitly, but possibly we could estimate the position of two images when having the information about their position with one other image. Detection and matching of SURF keypoints would be applied only to the pairs with an overlap. Most of the implemented methods were designed to be easily extended for this purpose.

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 44th 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. ZISSEMAN. *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] L. JUAN AND O. GWUN. *A Comparison of SIFT, PCA-SIFT and SURF*. In International Journal of Image Processing (IJIP), 2009.
- [11] F. JURIE AND C. SCHMID. *Scale-invariant shape features for recognition of object categories*. In CVPR, volume II, pages 90 – 96, 2004.
- [12] T. KADIR AND M. BRADY. *Scale, saliency and image description*. IJCV, 45(2):83 – 105, 2001.
- [13] N. Y. KHAN, B. MCCANE, AND G. WYVILL. *SIFT and SURF Performance Evaluation Against Various Image Deformations on Benchmark Dataset*. On International Conference on Digital Image Computing: Techniques and Applications, 2001.
- [14] T. LINDEBERG. *Feature detection with automatic scale selection*. In International Journal of Computer Vision, 30(2):79 – 116, 1998.
- [15] D. LOWE. *Object recognition from scale-invariant features*. In CCV, 1999.

- [16] 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.
- [17] S. ROY. *Stereo Without Epipolar Lines: A Maximum-Flow Formulation*. In International Journal of Computer Vision, 1999.
- [18] N. SNAVELY; S. M. SEITZ; R. SZELISKI. *Modeling the world from Internet photo collections*. In International Journal of Computer Vision, 2007.
- [19] 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.
- [20] 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.