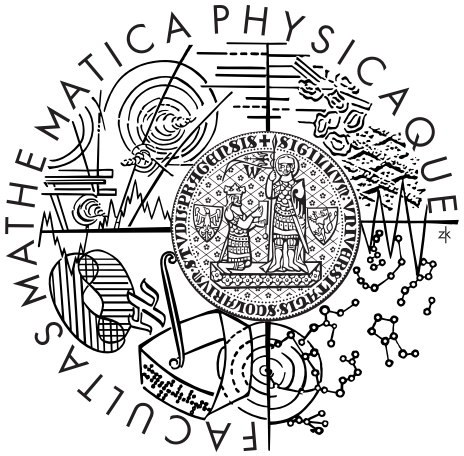
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Viktorie Vášová

3D Computer Vision on the Android Platform

Name of the department or institute

Supervisor of the bachelor thesis: Mgr. Lukáš Mach

Study programme: General Computer Science

Prague 2013

Dedication.

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

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

In ..... date ..... signature of the author

Název práce: 3D počítačové vidění pro platformu Android

Autor: Viktorie Vášová

Katedra: Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: Mgr. Lukáš Mach

Abstrakt:

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

Title: 3D Computer Vision on the Android Platform

Author: Viktorie Vášová

Department: Computer Science Institute of Charles University

Supervisor: Mgr. Lukáš Mach

Abstract:

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

Contents

Introduction **2**

1 Overview **3**
1.1 Currently Available Software . . . 3
1.2 Currently Available Libraries . . . 3

2 Related Work **5**
2.1 Interest Point Detection . . . 5
2.2 Interest Point Description . . . 5

3 Used Mathematical Terminology **6**
3.1 Hessian matrix . . . 6
3.2 Integral images . . . 6

4 Image processing algorithms **7**
4.1 The SIFT algorithm . . . 7
4.2 The SURF algorithm . . . 9
4.3 SIFT vs. SURF . . . 10
4.4 Maximum Flow Graph Cut Algorithm . . . 10

5 Developing for Android **12**
5.1 Introduction to Android development . . . 12
5.2 Application Components . . . 14
5.3 The Activity Lifecycle . . . 15
5.4 User Interface . . . 16
5.5 Sensors . . . 16
5.6 Data Storage . . . 16

6 OpenCV **17**
6.1 The Origin of OpenCV . . . 17
6.2 Structure of OpenCV . . . 18
6.3 OpenCV4Android . . . 19

7 Implementation **20**
7.1 . . . 20
7.2 . . . 20

Conclusion **21**

Bibliography **22**

List of Tables **23**

List of Abbreviations **24**

Attachments **25**

Introduction

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

Simultaneously, both the general availability and the computational strength of smartphones have improved rapidly. Especially mobile phones that employ the Linux-based Android software platform are currently very popular. (*TODO: uvest procentualni podil na trhu, pridat citaci.*) A built–in camera and a relatively fast CPUs are very common in such mobile phones.

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

(*TODO: nasledujici vety by se mely primo odkazovat na sekce.*) The first part of this work analyses the problem, describes available software and gives an overview of programming libraries and languages that were used. Secondly, we focus on the theoretical foundations of 3D reconstruction from image data. The next section is devoted to the implementation of our application. Finally, we evaluate and benchmark the resulting application. (*TODO: tady bychom urcite chteli explicitneji napsat, ze vysledkem budou i nejake datasety, ale to se zvladne az to budeme mit napsane.*)

1. Overview

Many years of researches resulted in several applications. In this part we will give an overview of available software dealing with the analysis of depth information and 3D reconstruction. In the second part of this chapter, available programming languages and libraries considered for our work, are discussed.

1.1 Currently Available Software

One of the first applications that were used to create a 3D model from a set of pictures of an object was Photosynth designed by Microsoft in cooperation with University of Washington. The algorithm is based on pattern recognition and generates a 3D model of a photographed object including the point cloud. After releasing the application there were available only projects generated by Microsoft or BBC and later a cooperation with NASA was started. Until two years later the version for public was released so users could upload own images to create a 3D model.

In 2007, a year after releasing Photosynth, Google introduced Street View to extend Google Maps and Google Earth. At first, this additional application provided a panoramic views of cities in the USA, but soon it expanded to other places in the world.

Autodesk, an American corporation focused on 3D design software, released modelling application Autodesk 123D recently. There are several additional tools available. One of them is 123D Catch that creates 3D model from a set of pictures taken from different view angles. This software is compatible with Autodesk 123D application, so it is advisable idea for designers who want to work with real–world objects in the virtual scene. The program is available for these operating systems: Windows, Mac OSX and iOS. It seems that for creating such a 3D model it is necessary to follow detailed instructions how to shot the pictures. In most cases the process of building model fails because of wrong set of images. An error can occur when pictures are blurred, the background is not solid or the amount of photos is not sufficient.

If we evaluate accessibility of the software for mobile phones, Google Street View is running on every type of mobile platform without any larger errors. There is a version of Photosynth for Windows phones and iOS operating system. As we already mentioned, Autodesk developed a version for iOS as well. But apparently, we miss applications developed for Android platform.

1.2 Currently Available Libraries

The problem considered in this work is getting an image information from a set of pictures and its processing afterwards. To be able to program a software dealing with a task from the area of computer vision, it is necessary to be familiar with a library that supports work with images. That kind of functions offers OpenCV library.

OpenCV is a cross–platform library developed by Intel. It provides large scale

of functions supporting image processing; classes for segmentation and recognition, blob detection and other 2D and 3D feature toolkits are available.

For our work is important that support for C, C++, Python and the Android platform is included in the library. OpenCV4Android offers us great equipment for image processing.

2. Related Work

In this section we mention some of the approaches that have been proposed earlier. We consider algorithms for interest points detection and description especially. It gives us an overview of already existing work.

2.1 Interest Point Detection

In 1988, Chris Harris and Mike Stephens [2] published their corner detector based on combining corner and edge detector. Although this approach is not scale–invariant, it is widely used nowadays.

However, many computer vision tasks deal with the real world input data images where objects appear in different ways depending on the scale. Due to this fact, Lindberg came out with automatic scale selection for feature detection [5].

There are several other approaches that have been introduced such as edge–based region detector by Jurie and Schmid [3] or salient region detector by Kadir and Brady [4]. But apparently they are not used that much. It seems that due better results and stability, most popular are Hessian–based detectors.

Another example of Hessian–based detector is a detector published by Herbert Bay et al. in Speeded–Up Robust Features [1]. This detector is partly inspired by SIFT algorithm, but it is several times faster and more robust against the scale transformation and the image rotation. It relies on the approximation of determinant of the Hessian matrix that can be computed using integral images. Hence, the computation is very fast (see the previous chapter).

2.2 Interest Point Description

Large number of interest point descriptors were introduced. The most common one is scale–invariant feature transform [6], SIFT for short. This algorithm was published in 1999 by a Canadian computer scientist David G. Lowe. The descriptor is computed from the intensities around the key point locations where the local gradient direction of picture intensities gives description of the local key point.

3. Used Mathematical Terminology

To make this work more integrated, we will briefly remind some mathematical concepts that are used in following chapters.

3.1 Hessian matrix

Many feature detectors and interest points descriptors use the Hessian matrix. The Hessian is a square matrix of the second order partial derivatives of a function. Assuming all second partial derivatives exist, for a given function $f(x_1, x_2, \ldots, x_n)$ the Hessian looks

$$H(f) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

3.2 Integral images

Later in this work we work with the term of integral images. An integral image (also known as summed area table) allows fast and efficient computation over a rectangular area in an image. Each pixel represents the sum of all previous pixels above to the left. To be more formal, let's see the mathematical formula describing the value of a pixel, where I is an integral image and $\mathbf{x} = (x, y)^T$ is a location of a current pixel.

$$I(\mathbf{x}) = \sum_{i=0}^{i<x} \sum_{j=0}^{j<x} I(i, j) \tag{3.1}$$

An advantage of an integral image is that we are able to construct it with only one pass over the original image. Moreover, once we have calculated the integral image, it requires only three integer operations and four memory accesses to calculate the sum of intensities inside the region of any size in the picture (see figure 3.1).

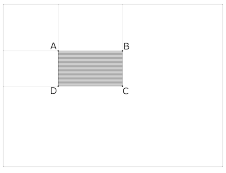


Figure 3.1: The sum of any rectangular region can be achieved by only three additions.

4. Image processing algorithms

In the previous chapter we gave an overview of related work and algorithms that were introduced. This chapter is concentrated on some of these approaches in greater detail. We will describe three techniques – SIFT and SURF algorithm for keypoints detection and then Min-cut/Max-flow graph algorithm. SIFT and SURF are the most used algorithms to detect features. Both of them are robust to scale and rotation.

Generally, feature matching algorithms first extract keypoints from a pair of photos of the same scene. If the feature detection is repeatable, most of the features should be detected in both images. Then, for each feature a descriptor is generated, typically it is a high-dimensional vector. Usually, a descriptor is affine invariant. Hence for a feature from different viewpoints should be generated the same vector. Since the descriptors are computed, we can match them between a pair of images. Due to the fact that our descriptors are vectors, we can match them according to the distance in the space, using Euclidean distance for example.

4.1 The SIFT algorithm

Scale-invariant feature transform (SIFT) was published already in 1999 by David Lowe [6]. Until now it is still very popular algorithm to generate matches for a pair of pictures. It is able to identify objects even among clutter in noisy images. SIFT feature descriptor is 128-dimensional vector invariant to scaling and orientation but only partially invariant to affine distortion and illumination changes. Hence the best results are given on the input images that do not differ much in the orientation.

The feature points are detected from the grayscale version of an input image matching centres of blob-like structures. To each feature point is assigned an information about local orientation and scale and based on these data is the descriptor constructed.

First step to reach the set of keypoints is creating a scale-space of an image. A scale-space is a series of blurred images derived from convolution of the image with Gaussian filter. The original image is blurred several times with larger and larger Gaussian kernel, then it is downsized and this process is applied again to the subsample. Since we have a scale-space, we subtract neighbouring images and like this generate a set of differences of Gaussians (DoG). A DoG image can be formulated as

$$D(x, y) = G_{\sigma_1} * f(x, y) - G_{\sigma_2} * f(x, y)$$

where $\mathbf{f}(\mathbf{x}, \mathbf{y})$ is the original image, $*$ presents convolution, $\sigma_\mathbf{i}$ is the scale and $\mathbf{G}_{\sigma_\mathbf{i}}$ is Gaussian kernel. Hence

$$D(x,y) = \left[\left(\frac{1}{\sqrt{2\pi\sigma_1^2}} \exp\left(-\frac{x^2+y^2}{2\sigma_1^2} \right) \right) - \left(\frac{1}{\sqrt{2\pi\sigma_2^2}} \exp\left(-\frac{x^2+y^2}{2\sigma_2^2} \right) \right) \right] * f(x,y)$$

These convolved images are grouped by octave, where an octave corresponds to doubling value of scale σ.

Now we can define candidate keypoints as maxima and minima of the result of difference of Gaussian that occur at multiple scales. That are exactly the areas we are looking for - spots of high contrast. This is done by comparing each pixel with its neighbouring pixels and also with corresponding pixels in neighbouring DoG image. If the pixel value is the largest or the lowest among all compared pixels, it is selected as a candidate feature point.

Usually, the detection of scale-space extrema results is a large amount of candidate keypoints. However, some of them are not stable and they need to be discarded. For each candidate we do the interpolation of nearby data to estimate its position. It is done using quadratic Taylor expansion of the Difference-of-Gaussian function around point:

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

where $\mathbf{x} = (\mathbf{x}, \mathbf{y}, \sigma)$ is the offset from the point. If the offset value is less than $\mathbf{0.03}$, it indicates low contrast and a high probability of affection by noise or of corresponding to edges and the candidate is discarded. Otherwise we keep the keypoint.

The next step is to describe the nearby area of each keypoint to generate the descriptor. To achieve the invariance to rotation, the value of the gradient that establishes local orientation and scale of the keypoint is used. Firstly we compute gradients for pixels around the location of the keypoint. The nearby area is divided to 4x4 subregions and for each subregion is computed a histogram of 8 gradient values. The peaks of the histogram presents the dominant orientations. The orientations corresponding to the highest peak or peaks that are within 80% of the highest peak are assigned to the feature.

We create the descriptor using the gradient magnitudes of these regions. For a feature there are sixteen subregions, each with eight values of a histogram, that results in a 128-dimensional vector.

All extracted keypoints are stored in a database; the last step is matching them. Typically, we do this by taking one feature and looking for the nearest (in the meaning of Euclidian distance) vector descriptor in the second image.

4.2 The SURF algorithm

Another popular algorithm detecting features is SURF (Speeded Up Robust Features). It was presented by Herbert Bay et al. [1] in the year 2006 and it is partially inspired by SIFT described above. SURF is based on computation of Haar wavelet responses, Hessian-matrix approximation and using integral images.

To build a robust descriptor we need to detect keypoints that are invariant to scale thus we create a scale-space. That is because often is required to detect objects in different distances. A scale-space is usually implemented as an pyramid of subsequently smoothed images by Gaussian filter as was described in previous section concerned with SIFT algorithm. In this approach integral images are used, see previous chapter to get detailed information about integral images. So we do not have to iteratively apply the same filter, but we can reach the scaling by fast computing of integral image where the speed is independent to the size of filter.

The detection of interest points is done by using basic Hessian-matrix and its approximation. The Hessian for a point $\mathbf{x} = (x, y)$ in an image $\mathbf{I}$ at scale σ is defined as

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

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

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

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

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

Since the Gaussian is considered as over-rated, in this approach we use box-filters instead and obtain speeded-up variation. Box filter approximation of Gaussian can be computed very quickly using integral images. Size of a box filter correspond to the scale size of Gaussian, furthermore due to way of computing integral images, the calculation time is independent to the scale size. These approximations are computed for every scale of the scale-space and those points that simultaneously are local extrema of both the determinant and trace of the Hessian matrix are chosen as candidate keypoints. The trace of Hessian matrix is identical to the Laplacian of Gaussians (LoG). If we denote those approximating box filters as $\mathbf{D_{xx}}$, $\mathbf{D_{yy}}$ and $\mathbf{D_{xy}}$, we get the determinant value

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

SURF descriptor is a high-dimensional vector consisting of the sum of the Haar wavelet response around the point of interest. The vector describes the distribution of the intensity within the neighbourhood of the keypoint. At first, to be invariant to image rotation, for a feature dominant orientation is established. The Haar wavelet responses in $\mathbf{x}$ and $\mathbf{y}$ direction are calculated and they are summed within a sliding orientation window of size $\frac{\pi}{\mathbf{3}}$ afterwards. Again, the

responses can be computed with the aid of the integral image. Summed responses then yield a local orientation vector. The longest such vector lends the orientation of the feature.

The first step to extract the descriptor is constructing a square region around the keypoint aligned with dominant orientation. The size of the square area is **20s**, where **s** is the scale where the keypoint was detected. We split the window into 4 x 4 regular square subregions and for each of them we compute Haar wavelet responses $\mathbf{d_x}$ and $\mathbf{d_y}$ at 5 x 5 evenly spaced sample points inside. The sum of the responses for $\mathbf{d_x}$ and $\mathbf{d_y}$ and their absolute values $|\mathbf{d_x}|$, $|\mathbf{d_y}|$ separately results in a descriptor $\mathbf{v} = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|)$ for each subregion. Concatenating this for all 4 x 4 subregions gives 64-dimensional vector descriptor.

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

4.3 SIFT vs. SURF

Both SIFT and SURF are popular feature descriptors based on the distribution of intensity around the interest point. SIFT used to be evaluated as the most robust and distinctive descriptor for feature matching. However, SURF has been shown to have similar performance to SIFT, but being much faster. The results of research in [1] point out the fast computation of a descriptor. SURF describes image three times faster than SIFT. The key to the speed is using integral images. Also, the SURF descriptor is 64-dimensional vector in the comparison to 128 integers describing SIFT feature. Hence the SIFT matching costs more calculation time. Furthermore, due to the global integration of SURF-descriptor, it stays more robust to various perturbations than the SIFT descriptor.

SURF offers invariance to rotation, handling image blurring, but SIFT is still more robust in illumination changes and viewpoint changes.

4.4 Maximum Flow Graph Cut Algorithm

In computer vision there is a general task of stereo correspondence. It is a problem of discovering the closest match between points of two pictures typically taken from different positions. Presumably, rectification of the images considerably simplifies the situation. Sometimes this constraint can be done. Once the correspondences are solved, we obtain a disparity map and it can be exploited to reconstruct the positions and distances of the cameras.

Generally, there are two approaches to find the corresponding points. One of them is to detect features and interest points in an image and then find corresponding points in the other one. Only these high distinctiveness points are matched, hence it is called sparse stereo matching. The second way, dense stereo matching, is to match as many pixels as possible.

In 1999 Sébastien Roy published an algorithm [7] [8] to solve the stereo correspondence problem formulated as a maxim-flow problem in a graph. This approach does not use of epipolar geometry as many other do. It solves efficiently maximum-flow and gives a coherent minimum-cut that presents a disparity surface for the whole image. This way provides more accurate depth map then the classic algorithms and the result is computed in shorter time in addition.

5. Developing for Android

Android, Inc. was founded in October in 2003 in California by Andy Rubin, Rich Miner, Nick Sears and Chris White. At first, the company planned to develop an operating system for digital camera, but when they realised that the market is not large enough, they decided to change their intentions and focused on smartphone operating system. That time, the Apple's iPhone was not released yet. The rival mobile phones operating systems were Symbian and Windows Mobile.

After four years with the financial aid of Google, Android was revealed. The first phone with Android operating system was sold in autumn 2008.

Android is open source operating system and the code is released. This allows developers to modify and freely distribute the software. Due this, Android became the most expanded and used operating system for smart phones.

In this chapter we will introduce the Android operating system and give a brief description how to implement an Android application.

5.1 Introduction to Android development

Android is a product of a group developing open standards for mobile devices led by Google. The Open Handset Alliance includes mobile operators, software providers and device and component manufacturers such as T-Mobile, HTC, Sony, Wind River Systems, Sprint Nextel and others. The first handset device running the Android operating system on the market was G1 developed by HTC (also known as the HTC Dream). It was released and offered to T-Mobile costumers in 2008 and a half-year after T-Mobile USA announced it had sold one million G1s.

The Android platform is layered environment built upon a foundation of the Linux kernel. It includes wide range of functions and user interface supporting views, windows, displaying boxes and lists. Also an embedded browser build upon WebKit is included. WebKit is an open source browser engine that powers Apple's Safari web browser and the latest versions of Google Chrom. Most Android-powered devices have build in sensors to measure temperature, motion and orientation such as accelerometer, gyroscope or a barometer. It also provides an array of connectivity options, for example WiFi, Bluetooth or wireless data. A build-in camera support is offered as well. In an effort to improve graphics, Android platform supports environment for 2D and 3D graphics development including OpenGL library. The data storage is sponsored by the SQLite database, a relational database management system as a library implementing self-contained and server-less SQL database engine.

As we already mentioned, Android platform is powered by the Linux kernel. It is used for memory and process management, device drivers, and networking. From the view of point of an Android architecture, this is the first and basic layer. Upon this level are Android native libraries including graphics, media codes, database (SQLite) and WebKit. They are all written in C or C++, but called through Java interface. We should note that the virtual machine applications are running within is not Java virtual machine, but Dalvik Virtual machine. It is a type of JVM adapted to low processing and memory power for Android.

Android runtime layer consists of this DVM and core Java libraries. The next level up is the Application Framework. This layer manages the basic functions of phone. The most important part of it is Activity Manager that manages the life cycle of an application and Content Providers managing the data sharing between applications. Other notable parts are Telephony Manager (handling voice calls), Location Manager (specifying location using GPS or cell tower) and Resource Manager. The top layer of the Android architecture is formed by applications. Some of them are preinstalled and provided by Google and parts are third party applications created by the community of developers. You can view the described structure of software layers in the diagram below.

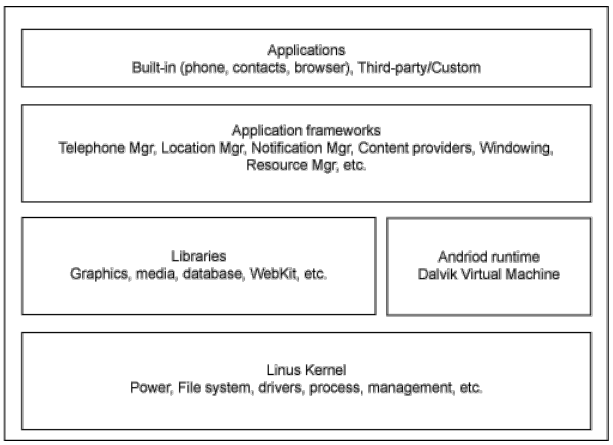


Figure 5.1: Structure of Android architecture.

Android applications are written in Java programming language and runs within an instance of Dalvik virtual machine. An application using the user interface is implemented with an activity. An activity creates a window of the application and allows the developer to view the UI. An inseparable part of the application is AndroidManifest.xml file containing necessary information how to properly install it to the device including permissions the application needs to run. An example of such a permission is acquiescence to use a camera equipment, access memory to write files or use the Internet. All these requirements need to be explicitly listed in the manifest file. We will concentrate more on activities and their development in the next sections.

There are required tools to develop Android applications. The Android software development kit (the Android SDK) is freely available on the developer website. The environment to develop an application is the Eclipse IDE. The Android SDK is provided as .zip file including Java archive file containing all necessary classes to build an application (android.jar), the SDK documentation, a directory with sample code, Android Debug Bridge and other tools needed to build an application.

In the rest of this chapter we will describe some of the implementing parts when developing an application.

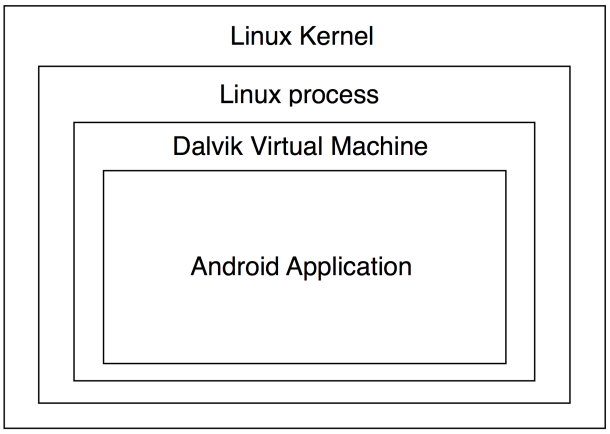


Figure 5.2: Each Android application runs within an instance of Dalvik virtual machine which is placed in Linux kernel process.

5.2 Application Components

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

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

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

- Activities
- Services
- Content providers
- Broadcast receivers

Activity is a screen providing the user interface. In the activity a developer place elements such as Views, Lists, Buttons, Labels etc. The layout of an activity and the widgets placed in the window are described in a separate XML file (we will reveal more about user interface later). Most of applications consist of more than one bounded activities. Although the activities form one application, they

are all independent from each other. If another application has a permission to do so, it can start an activity of a different app.

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

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

A *content provider* manages sharing data across the applications. When the app is storing any data - in file system, SQLite database or any online storage, through the content provider they can be accessed or even modified from other applications. An example when a content provider is used is an application working with the contacts database.

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

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

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

5.3 The Activity Lifecycle

As we already stated, an app can consist of one or more activities. Such an application switches between different states of its life cycle. We don't have such a control on the life time as we have in desktop platforms. 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:

- Starting state
- Running state
- Paused state
- Stopped state
- Destroyed state

The starting state and 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 and affects the battery and other processes. That is also the reason why we don't 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.

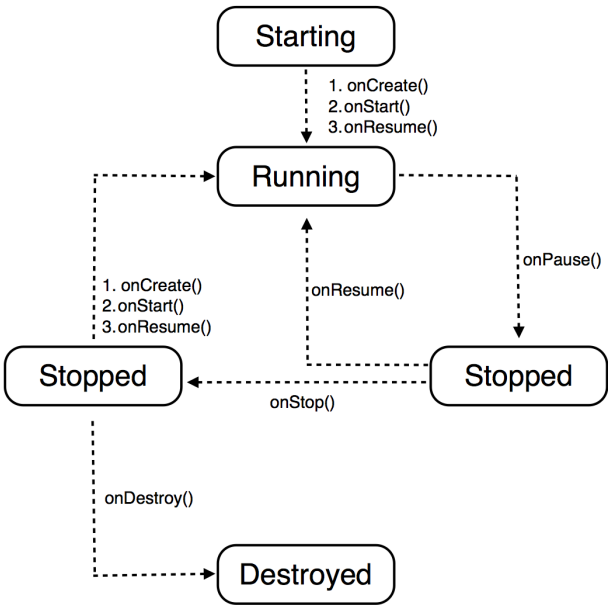


Figure 5.3: Life cycles of an application.

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

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

5.4 User Interface

5.5 Sensors

5.6 Data Storage

6. OpenCV

OpenCV is an open source library covering the area of computer vision (OpenCV is an abbreviation for Open Source Computer Vision Library). It was designed mainly for real-time applications and computational efficiency. OpenCV is written in C and C++ and can take advantage of multicore processors. Hence the OpenCV functions are efficient and The library is cross-platform and runs under Linux, Windows, Mac OS X. Even further, there is active development on interfaces for Python, Matlab, and other programming languages.

Nowadays, computer vision is widely used in many parts of computer science. We should notice that it is increasingly being used for images and video for the web applications. For example camera calibration and image stitching techniques are applied to street-map images such as in Google's Street View.

As we already mentioned, OpenCV mainly aimed at real-time applications and computational efficiency. If we are interested in further optimization we can achieve it by installing Integrated Performance Primitives (IPP), that consist of low-level optimized routines and are used automatically by OpenCV when the library is installed.

OpenCV provides an environment to create vision applications more easily and more effective and sophisticated. It contains over 500 functions spanning many areas of computer vision, for example medical imaging, camera calibration, robotics or stereo vision.

6.1 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 known for their Robot Operating System. The latest version of OpenCV is 2.4.5 released in the April of 2013.

Since the first release, OpenCV has been used in many applications. It has even been used in sound and music recognition region where the vision techniques were applied to sound spectrogram images.

6.2 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. You can view them in the figure below. First of all, it would be the CXCore component containing the basic data structures and operations on them. Then the HighGUI component ensures I/O routines and functions for loading and storing images. ML is the machine learning library; and finally Computer Vision component containing basic image processing and vision algorithm.

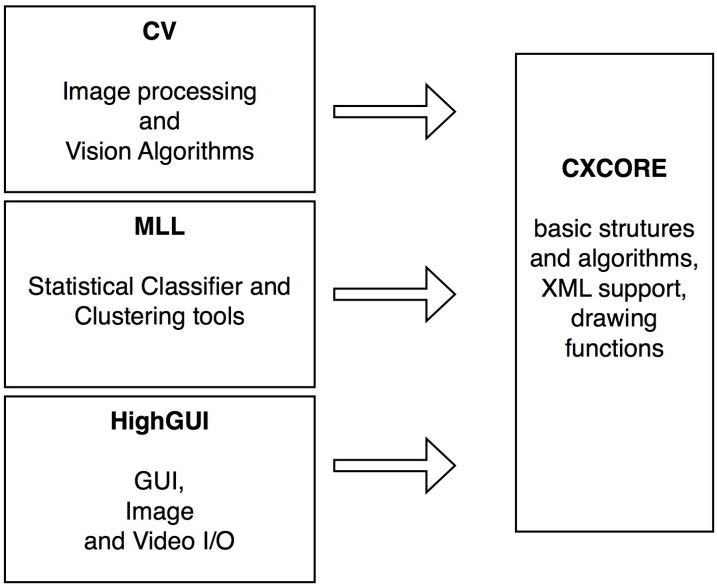


Figure 6.1: The basic structure of OpenCV library.

CXCore provides the core functionality including:

- 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

6.3 OpenCV4Android

OpenCV library was written in C and this makes it portable to almost any commercial system. After all, OpenCV was designed to be cross-platform. Since version 2.0, OpenCV includes also its the new interface written in C++. Later also wrappers for languages such as Java or Python have been developed. 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 comparison with desktop version it includes also `opencv.android` package containing all additional functions for Android platform. This package provides mainly an environment for work with camera by offering classes such as `NativeCameraView.java` or `CameraBridgeViewBase.java`.

7. Implementation

7.1

7.2

Conclusion

Bibliography

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

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

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

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

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

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

[7] S. Roy. *A Maximum-Flow Formulation of the N-camera Stereo Correspondence Problem.* Computer Vision, 1998. Sixth International Conference on.

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

List of Tables

List of Abbreviations

Attachments