# Distributed Mobile Computer Vision and Applications on the Android Platform

Sebastian Olsson
Philip Åkesson

Master's thesis
2009:E22

Lund University

Faculty of Engineering
Centre for Mathematical Sciences
Mathematics

Centrum Scientiarum Mathematicarum

**Abstract**

This thesis describes the theory and implementation of both local and distributed systems for object recognition on the mobile Android platform. It further describes the possibilities and limitations of computer vision applications on modern mobile devices. Depending on the application, some or all of the computations may be outsourced to a server to improve performance.

The object recognition methods used are based on local features. These features are extracted and matched against a known set of features in the mobile device or on the server depending on the implementation. In the thesis we describe local features using the popular SIFT and SURF algorithms. The matching is done using both simple exhaustive search and more advanced algorithms such as kd-tree best-bin-first search. To improve the quality of the matches in regards to false positives we have used different RANSAC type iterative methods.

We describe two implementations of applications for single- and multi-object recognition, and a third, heavily optimized, SURF implementation to achieve near real-time tracking on the client.

The implementations are focused on the Java language and special considerations have been taken to accommodate this. This choice of platform reflects the general direction of the mobile industry, where an increasing amount of application development is done in high-level languages such as Java. We also investigate the use of native programming languages such as C/C++ on the same platform.

Finally, we present some possible extensions of our implementations as future work. These extensions specifically take advantage of the hardware and abilities of modern mobile devices, including orientation sensors and cameras.

**Acknowledgments**

We would like to thank our advisors Fredrik Fingal at Epsilon IT and Carl Olsson at the Centre for Mathematical Sciences at Lund University.
We would further like to thank Epsilon IT for the generous accomodations during the project.

# Contents

# Chapter 1

# Introduction

## 1.1 Background

The interest in computer vision has exploded in recent years. Features such as recognition and 3d construction have become available in several fields. These computations are however advanced both mathematically and computationally, limiting the available end user applications.

Imaging of different sorts has become a basic feature in modern mobile phones along with the introduction of the camera phone, which leads to a market for computer vision related implementations on such platforms.

The progress in mobile application development moves more and more towards service development with support for multiple platforms rather than platform specific low level development. While previous mobile development was almost exclusively done in low level languages such as assembly and C, the industry moves towards the use of high level languages such as Java and C#. This switch is motivated by the huge development speed up, which leads to cost decrease and feature increase and also motivated by the huge common code base in such languages. It is made possible by the increase in computational performance much like it was when the same type of switch was introduced on desktop computers some years ago.

Power efficiency is a large issue on mobile devices, which suggests that parallel computing with multiple cores and processor types rather than an increase in single CPU clock speed is the way to go in mobile devices, again in the same manner as the idea of performance increase was changed in desktop computers some years ago. The use of dedicated processors for signaling has been utilized in mobile devices for years. Recent development has brought multiple cores and dedicated 3D graphic processors to the mobile devices.

The move to parallel computation and the natural limits of mobile devices such as network delays puts extra constraints on the high level programmer. These constraints are normally not thought of when developing a desktop application in a high level language.

## 1.2 Aim of the thesis

Our goal is to study and implement computer vision applications on handheld mobile devices, primarily based on the Android platform. We aim to do this by implementing and evaluating a number of computer vision algorithms on the Android emulator from the official SDK, and later possibly on a real device. We also want to find out what kind of calculations can be practically performed on the actual mobile device and what kind of calculations have to be performed server-side.

This master thesis will aim to use the versatility of high level development and still perform complex computations within a time span that a user can accept.

Not all users have high speed Internet access on their mobile phones, and not all users with high speed Internet access has unlimited data plans. Considerations have to be taken both regarding the time it takes to send data and the actual amount of data being transferred.

## 1.3 Related work

For a comprehensive look into the field of embedded computer vision see the book *Embedded computer vision* [16]. This book handles specific subjects such as computation with GPUs and DSPs and various real-time applications. It further discusses the challenges of mobile computer vision.

In [13], Fritz *et al* SIFT descriptors are used together with an information theoretical rejection criterion based on conditional entropy to reduce computational complexity and provide robust object detection on mobile phones. We present SIFT in Section 2.2 of this thesis.

An interactive museum guide is implemented by Bay *et al.* in [4]. This uses the SURF algorithm to recognize objects of art and is implemented on a Tablet PC. We present SURF in Section 2.3 of this thesis.

Ta *et al.* [25] implements a close to real-time version of SURF with improved memory usage on a Nokia N95.

Takacs *et al.* [26] developed a system for matching images taken with a GPS-equipped camera phone against a database of location-tagged images. They also devise a scheme for compressing SURF descriptors up to seven times based on entropy, without sacrificing matching performance.

Wagner *et al.* [30] uses modified versions of SIFT and Ferns [21] for real-time pose estimation on several different mobile devices.

## 1.4 Overview

Chapter 2 introduces the theory behind the object recognition methods used in our implementations.

Chapter 3 shows some quirks with the Dalvik VM and how to circumvent them to write as efficient code as possible. We also present some of our working implementations and the lessons learnt from implementing them.

Section 3.2.1 presents our Java implementation of SURF. In Section 3.2.2 we present an Android-optimized implementation.

In Section 3.3.1 we present a mobile application for recognition of paintings. The application uses either the Scale Invariant Feature Transform (see Section 2.2) or Speeded-Up Robust Features (see Section 2.3) running on the handheld device to detect and extract local features. The feature descriptors are then sent to a server for matching against a database of known paintings.

In Section 3.3.2 we extend the application to recognize several objects in a single image. Due to the increased complexity, the feature detection and extraction is done server side. We also extend the matching to use best-bin-first search (Section 2.4.6) for better performance with larger databases and RANSAC (Section 2.5) for increased robustness. We implemented a client-server application for recognizing liquor bottles and presenting the user with drink suggestions using the available ingredients. The server was tested in a setup on a remote server using the Amazon Elastic Compute Cloud (EC2) to simulate a real life application setup with realistic database size.

Our third application, AndTracks, is presented in Section 3.3.3. AndTracks is based on optimized SURF and implemented in native C-code.

Finally we discuss our results and possibly future work in Chapter 4.

Appendix A contains a general description of the Android platform.

Appendix B describes the concept of cloud computing with the biggest focus on Amazon Elastic Cloud Compute (EC2), which is used in the Bartendroid application.

# Chapter 2

# Theory

## 2.1 Object recognition

Object recognition in images or image sequences is a fairly large area of computer vision and can be accomplished in many different ways. The route we are taking here is based on the concept of 'local features'. Local features are distinctive points in the image, such as corners or blobs, which can be reliably found under varying viewing conditions (scale, rotation etc.).

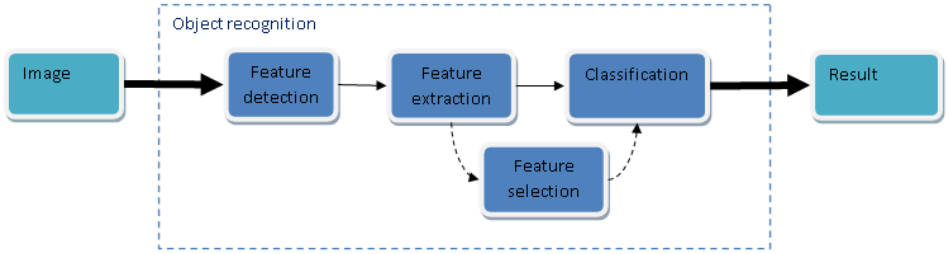



Figure 2.1: The main steps in object recognition.

Most popular algorithms consist mainly of four steps (see Figure 2.1):

- Feature detection
- Feature extraction
- Feature selection (optional)
- Classification

The feature detection step is usually performed using some edge, corner or blob detection algorithm to find features or keypoints that are in some sense 'interesting' in the image. The detection step should have a large repeatability. That is, it should have a high probability of finding the same features in multiple images of the same object.

Feature extraction is then quantifying, or describing, the detected features so that they can be compared and analyzed.

Two of the most popular methods for feature detection and extraction are SIFT (Section 2.2) and SURF (Section 2.3).

The feature selection step is optional, but might include dimensionality reduction or some information theory-based rejection criteria to make the amount of data more manageable.

Finally, the classification step consists of comparing the extracted features to a database of already known features to find the best matches. The matching can be done for example using exhaustive nearest neighbor search (Section 2.4), optimized kd-tree search (Section 2.4.5) or best-bin-first approximate nearest neighbor search (Section 2.4.6). Once the nearest neighbor search is complete, iterative methods like RANSAC (Section 2.5) can be used to discard outliers (incorrect matches) and verify the geometry to further improve the results.

## 2.2 Scale-Invariant Feature Transform

### Overview

Scale-invariant feature transform (SIFT) is an algorithm for detection and description of local image features, invariant to scale and rotation. It was introduced in 1999 by Lowe [17]. The main steps are:

1. Difference of Gaussians extrema detection
2. Keypoint localization
   (a) Determine location and scale
   (b) Keypoint selection based on stability measure
3. Orientation assignment
4. Keypoint descriptor

### Difference of Gaussians extrema detection

The keypoints are obtained by first creating a scale-space pyramid of differences between Gaussian convolutions of the image at different scales, and then finding local extreme points.

The scale-space of an image $I(x, y)$ is defined as the convolution function

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

with the variable scale Gaussian kernel

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \mathrm{e}^{-(x^2+y^2)/2\sigma^2}.$$

The result of convolving an image with the Difference of Gaussian kernel is

$$D(x,y,\sigma) = \big(G(x,y,k\sigma) - G(x,y,\sigma)\big) * I(x,y) = L(x,y,k\sigma) - L(x,y,\sigma),$$

see Figure 2.2.

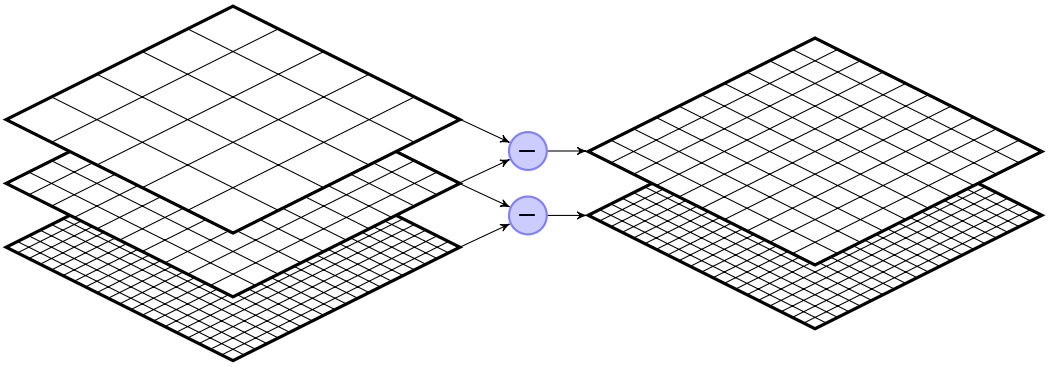


Figure 2.2: A scale-space pyramid with increasing scale (left). Neighboring scales are subtracted to produce the difference-of-Gaussians pyramid (right).

The difference of Gaussian is a close approximation [18] to the scale-normalized Laplacian of Gaussian

$$\nabla^2_{norm} L(x,y,\sigma) = \sigma^2 \nabla^2 L(x,y,\sigma) = \sigma^2 (L_{xx} + L_{yy})$$

which gives strong positive response for dark blobs and strong negative response for bright blobs. This can be used to find interest points over both space and different scales. The relation follows from the heat diffusion equation

$$\frac{\partial G}{\partial \sigma} = \sigma \nabla^2 G$$

and by finite difference approximation

$$\sigma \nabla^2 G(x,y,\sigma) = G_{xx} + G_{yy} \approx \frac{G(x,y,k\sigma) - G(x,y,\sigma)}{k\sigma - \sigma}$$

so that

$$D(x,y,\sigma) \approx \big((k-1)\sigma^2 \nabla^2 G(x,y,\sigma)\big) * I(x,y).$$

After calculating the difference of Gaussians for each scale, each pixel is then compared to its 26 surrounding neighbors in the same and neighboring scales (see Figure 2.3) to see if it's a local minimum or maximum, in which case it's considered as a possible keypoint.

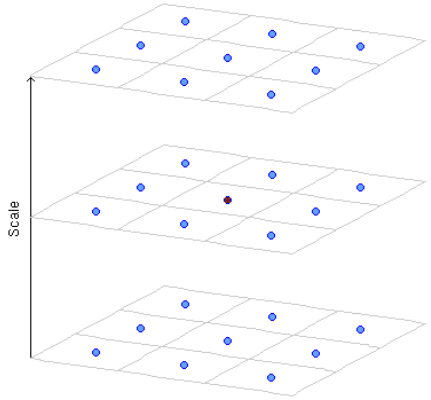



Figure 2.3: Neighboring pixels in scale-space. There are eight neighbors in the same scale and nine each in the scale above and below.

### Keypoint localization step

To accurately determine a candidate keypoint's location and scale, a 3D quadratic function is used to find the interpolated maximum. The quadratic Taylor expansion with origin in the candidate point

$$D(x, y, \sigma) = D + \frac{\partial D^{\mathrm{T}}}{\partial \mathbf{x}}\mathbf{x} + \frac{1}{2}\mathbf{x}^{\mathrm{T}}\frac{\partial^2 D}{\partial \mathbf{x}^2}\mathbf{x}.$$

The interpolated location is found by setting the derivative of $D(\mathbf{x})$ to zero, resulting in

$$\hat{\mathbf{x}} = -\frac{\partial^2 D}{\partial \mathbf{x}^2}^{-1}\frac{\partial D}{\partial \mathbf{x}}.$$

The derivatives of $D$ are approximated using differences between neighboring points. If the resulting location $\hat{\mathbf{x}}$ is closer to another point, the starting position is adjusted and the interpolation repeated. The value of $|D(\hat{\mathbf{x}})|$ at the interpolated location is used to reject unstable points with low contrast.

Keypoints on edges have large difference-of-Gaussian responses but poorly determined location, and thus tend to be unstable. The principal curvature of an edge point is large across the edge but small along the ridge. These curvatures can be calculated from the Hessian of $D$ evaluated at the current scale and location

$$\mathcal{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix},$$

The eigenvalues $\alpha$ and $\beta$ of $\mathcal{H}$ are proportional to the principal curvatures, which are the largest and smallest curvatures, and a large ratio between them indicates an edge. Since we only need the ratio it is sufficient to calculate

$$\mathrm{Tr}(\mathcal{H}) = D_{xx} + D_{yy} = \alpha + \beta,$$

$$\mathrm{Det}(\mathcal{H}) = D_{xx}D_{yy} - (D_{xy})^2 = \alpha\beta$$

and

$$\frac{\mathrm{Tr}(\mathcal{H})^2}{\mathrm{Det}(\mathcal{H})} = \frac{(\alpha+\beta)^2}{\alpha\beta} = \frac{(r\beta+\beta)^2}{r\beta^2} = \frac{(r+1)^2}{r},$$

where $r$ is the ratio between the largest and the smallest eigenvalue ($\alpha = r\beta$). Thresholding this (Lowe used $r = 10$ in [18]) gives overall better stability.

### Orientation assignment

When location and scale has been determined, gradient magnitude

$$m(x,y) = \sqrt{(L(x+1,y,\sigma) - L(x-1,y,\sigma))^2 + (L(x,y+1,\sigma) - L(x,y+1,\sigma))^2}$$

and orientation

$$\theta(x,y) = \tan^{-1}\left((L(x,y+1,\sigma) - L(x,y+1,\sigma))/(L(x+1,y,\sigma) - L(x-1,y,\sigma))\right)$$

are calculated in sample points around the keypoint. The gradients are used to build an orientation histogram with 36 bins, where each gradient is weighed by its magnitude and a Gaussian with center in the keypoint and 1.5 times the current scale. The highest peaks in the histograms are used as dominant orientations and new keypoints are created for all orientations above 80% of the maximum. The orientations are accurately determined by fitting parabolas to the histogram peaks.

### Keypoint descriptor

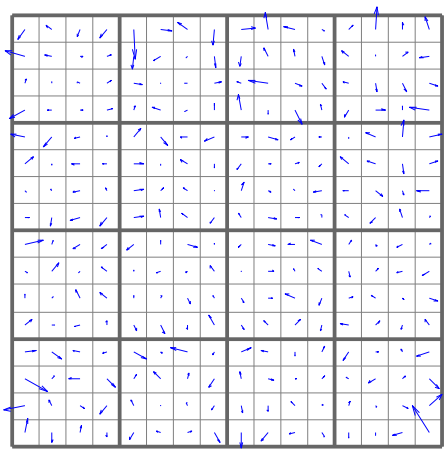
The keypoint descriptor is built by dividing the orientation-aligned region around the keypoint into $4 \times 4$ sub-regions with $16 \times 16$ sample points and calculating 8-bin histograms of gradient orientations for each sub-region (see Figure 2.4). This results in a 128-dimensional feature vector.

Normalization of the feature vector to unit length removes the effects of linear illumination changes. Some non-linear illumination invariance can also be achieved by thresholding the normalized vector to a max value of 0.2 and then renormalizing [18]. This shifts the importance from large gradient magnitudes to the general distribution of the orientations.

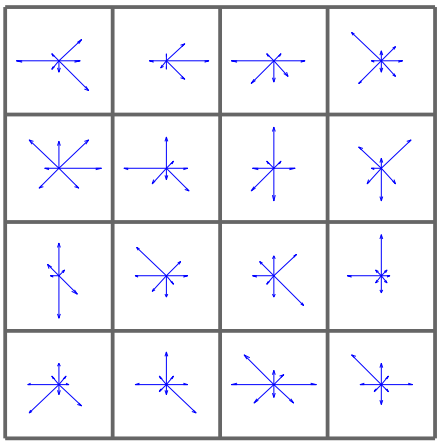
## 2.3 Speeded-Up Robust Features

### Overview

Speeded-Up Robust Features (SURF) is a scale- and in-plane rotation invariant image feature detector and descriptor introduced in 2006 by Bay et al. SURF is comparable to SIFT in performance while being faster and more efficient to compute [3]. SURF was inspired by SIFT, and many of the steps are similar. The main steps are:



(a)



(b)

Figure 2.4: The region around a keypoint is divided into $4 \times 4$ sub-regions with $16 \times 16$ sample points (a) which are then used to construct 4x4 orientation histograms with 8 bins each (b).

(a) $L(x, y, 0)$

(b) $L(x, y, \sigma)$

(c) $L(x, y, 2\sigma)$

(d) $L(x, y, \sigma) - L(x, y, 0)$

(e) $L(x, y, 2\sigma) - L(x, y, \sigma)$

Figure 2.5: Three images of Gaussians of the same source image and two images illustrating the difference between them

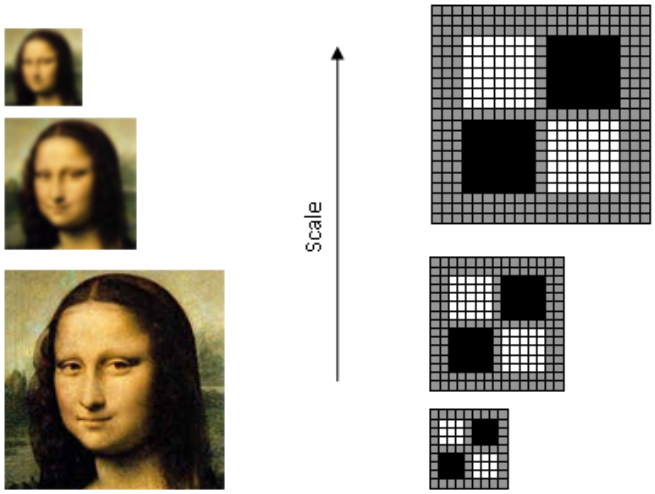



Figure 2.6: The scale invariance is achieved by applying box filters of varying sizes to the original image (right). This is in contrast to the more demanding scale-space pyramid created in SIFT where the filter remains the same and the image is resized (left).

1. Determinant of Hessian local extrema detection
2. Keypoint localization
   (a) Determine location and scale
3. Orientation assignment
4. Keypoint descriptor

For feature detection, SURF uses a determinant of Hessian blob detector. The Hessian matrix is approximated using box filters calculated with integral images. The scale invariance is achieved by applying box filters of varying sizes to the original image. This is in contrast to the more demanding scale-space pyramid created in SIFT where the filter remains the same and the image is resized.

The feature descriptors are calculated using the distribution of Haar wavelet responses.

### Integral images and box filters

The integral image, as defined in [29], $I_\Sigma$ of an image $I$ is computed as

$$I_\Sigma(x,y) = \sum_{i=1}^{x}\sum_{j=1}^{y} I(i,j).$$

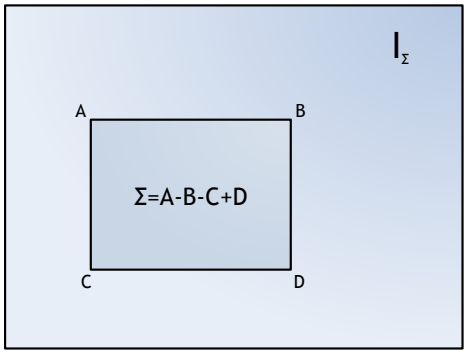



Figure 2.7: Calculation of a rectangular area sum from an integral image.

The entry in $I_\Sigma(x, y)$ then contains the sum of all pixel intensities from $(0, 0)$ to $(x, y)$, and the sum of intensities in any rectangular area in $I$ (see Figure 2.7) can be calculated with just three additions (and four memory accesses) as

$$\sum_{x=x_0}^{x_1} \sum_{y=y_0}^{y_1} I(x, y) = I_\Sigma(x_0, y_0) - I_\Sigma(x_1, y_0) - I_\Sigma(x_0, y_1) + I_\Sigma(x_1, y_1).$$

This allows for fast and easy computation of convolutions with filters composed of rectangular regions, also called box filters. See for example the Haar wavelet filters in Figure 2.10.

### Determinant of Hessian local extrema detection

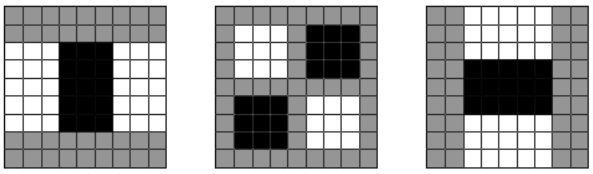
The use of the determinant of Hessian for keypoint detection can be motivated by (from Section 2.2) the relation between det $\mathcal{H}$ and the principal curvatures, which are the eigenvalues of $\mathcal{H}$. The product $\alpha\beta = \det \mathcal{H}$ is called the Gaussian curvature and can be used to find dark/bright blobs (positive Gaussian curvature) and differentiate them from saddle points (negative Gaussian curvature).

The scale-space Hessian matrix $\mathcal{H}(x, y, \sigma)$ at scale $\sigma$ in an image $I$ is defined as

$$\mathcal{H}(x, y, \sigma) = \begin{bmatrix} L_{xx}(x, y, \sigma) & L_{xy}(x, y, \sigma) \\ L_{xy}(x, y, \sigma) & L_{yy}(x, y, \sigma) \end{bmatrix},$$

where $L_{xx}$, $L_{xy}$ and $L_{yy}$ are convolutions of $I$ with the second order Gaussian derivatives $\frac{\partial^2}{\partial x^2} g(\sigma)$, $\frac{\partial^2}{\partial x \partial y} g(\sigma)$ and $\frac{\partial^2}{\partial y^2} g(\sigma)$ respectively (see Section 2.2). These second order derivatives can be approximated efficiently using integral images, with constant computation time as noted above. The filters and approximations used are illustrated in Figure 2.8. The width of the filters used at scale $\sigma$ is $\frac{9\sigma}{1.2}$.

(a) Discretized Gaussian second derivatives



(b) Box filter approximations

Figure 2.8: Discretized Gaussian second derivatives compared to their box filter approximations. The box filters are used for calculating the determinant of Hessian in the keypoint detection step of the SURF algorithm.

The approximate second order Gaussian derivatives are denoted $\hat{L}_{xx}$, $\hat{L}_{xy}$ and $\hat{L}_{yy}$. From these, the determinant of the Hessian is approximated as

$$\det(\mathcal{H}) \approx \hat{L}_{xx}\hat{L}_{yy} - \left(w\hat{L}_{xy}\right)^2,$$

where the relative weight $w = 0.9$ is needed for energy conservation in the approximation.

After thresholding the determinant values to only keep large enough responses, non-maximum suppression [20] is used to find the local minima and maxima in $3 \times 3 \times 3$ neighborhoods around each pixel in scale-space (see Figure 2.3).

### Keypoint localization

The candidate points are interpolated using the quadratic Taylor expansion

$$H(x, y, \sigma) = H(\mathbf{x}) = H + \frac{\partial H^{\mathrm{T}}}{\partial \mathbf{x}} + \frac{1}{2}\mathbf{x}^{\mathrm{T}}\frac{\partial^2 H}{\partial \mathbf{x}^2}\mathbf{x}$$

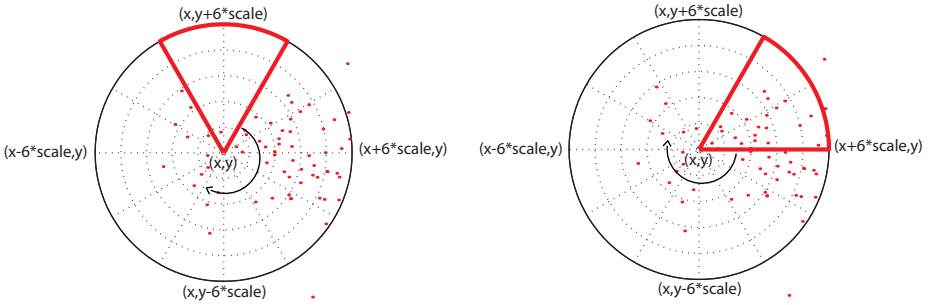



Figure 2.9: A sliding window around the keypoint is used to find the dominant orientation of Gaussian weighted Haar responses.

of the determinant of Hessian function in scale-space to determine their position with sub-pixel accuracy. The interpolation is repeated by setting the derivative

$$\hat{\mathbf{x}} = -\frac{\partial^2 H}{\partial \mathbf{x}^2}^{-1} \frac{\partial H}{\partial \mathbf{x}}$$

to zero and adjusting the position until $\mathbf{x}$ is less than 0.5 in all directions.

### Orientation assignment

To find the dominant orientation, Gaussian weighed Haar wavelet responses in the $x$ and $y$ directions are calculated in a circular region of radius $6\sigma$ around the interest point. The Gaussian used for weighting is centered in the interest point and has a standard deviation of $2.5\sigma$. The Haar wavelets responses use the integral image similarly to the previously used box filters. The responses are summed in a sliding orientation window of size $\frac{\pi}{3}$ and the longest resulting vector is the dominant orientation (see Figure 2.9).

When rotation invariance is not needed, an upright version of SURF (called U-SURF) can be computed faster since no orientation assignment is necessary. U-SURF is still robust to rotation of about $\pm 15°$ [3].

### Keypoint descriptor

The SURF descriptor is based on the distribution of first order Haar wavelet responses (Figure 2.10), computed using integral images. When the orientation has been determined, a square region of width $20\sigma$ is oriented along the dominant orientation and divided into $4 \times 4$ sub-regions. In each sub-region Haar wavelet responses are once again calculated in the $x$ and $y$ directions at $5 \times 5$ sample points with Haar wavelets of size $2\sigma$. The responses are then summed as $\sum dx$, $\sum |dx|$, $\sum dy$ and $\sum |dy|$, resulting in four descriptor values for each sub-region and a total of 64 values for each interest point.

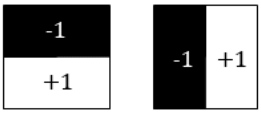



Figure 2.10: Simple 2 rectangle Haar-like features.

## 2.4 Classification

### 2.4.1 Overview

If only a single object is to be found in an image, it is usually sufficient to match the extracted features against a database of known features, using nearest neighbor search, until a certain number of matches from the same class has been found. The classification is then made by majority voting, where the image is assumed to contain the object with most matches. If the number of objects in the image is unknown however, majority voting cannot be used and outliers need to be eliminated in another way. A common method for outlier rejection is the iterative RANSAC, which is described in Section 2.5.

### 2.4.2 Nearest neighbor search

Given a feature descriptor $q \in \Re^d$ and a set of known features $P$ in $\Re^d$, the nearest neighbor of $q$ is the point $p_1 \in P$ with smallest Euclidean distance $||q - p_1||$ (see Figure 2.11). The feature $q$ is assumed to belong to the same class as $p_1$ if the ratio $\frac{p_1}{p_2}$ between the two closest neighbors is smaller than a threshold $\Theta$. In [18], $\Theta = 0.8$ was determined to be a good value for SIFT descriptors, eliminating 90% of the false matches while keeping more than 95% of the correct matches.

The simplest way to find the nearest neighbors is exhaustive or naïve search, where the feature to be classified is directly compared to every element in the database. This search is of complexity $\mathcal{O}(nd)$, for $n$ features in the database, which quickly becomes impractical as the database grows. For large databases in high dimensions, best-bin-first search (Section 2.4.6) in optimized kd-trees (Section 2.4.6) can be used instead.

### 2.4.3 Optimized kd-trees

Kd-trees are k-dimensional binary trees [6], using hyperplanes to subdivide the feature space into convex sets. See Figure 2.12 for an example partition of a kd-tree in $\Re^2$.

In the optimized kd-trees introduced by Friedman *et al.* [12], each non-terminal tree node contains a discriminator key (splitting dimension) and a partition value. The splitting dimension is chosen as the dimension with largest variance or range, and the partition value as the median value in this dimension.

The tree is built recursively from a root node by deciding the splitting dimension $d$, finding the median value and then placing smaller elements in the left child node

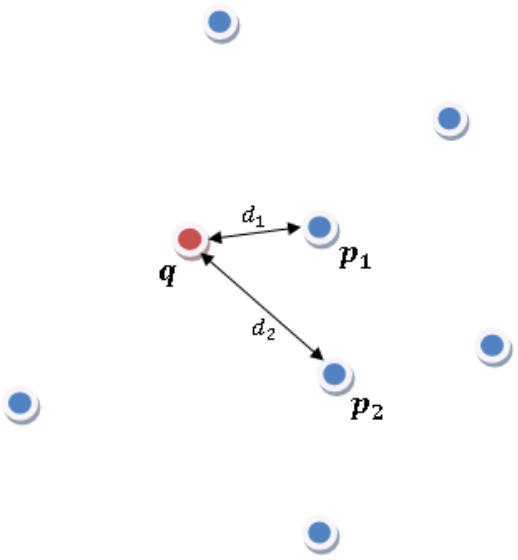



Figure 2.11: A point $q$ in $\Re^d$ and its two nearest neighbors $p_1$ and $p_2$, with distances $d_1$ and $d_2$ respectively.

and larger elements in the right child node (see Figure 2.13). When the number of elements in a sub-tree is smaller than a certain number, the child node is made terminal (called a *leaf node*) and the elements placed in a *bin* in the leaf node.

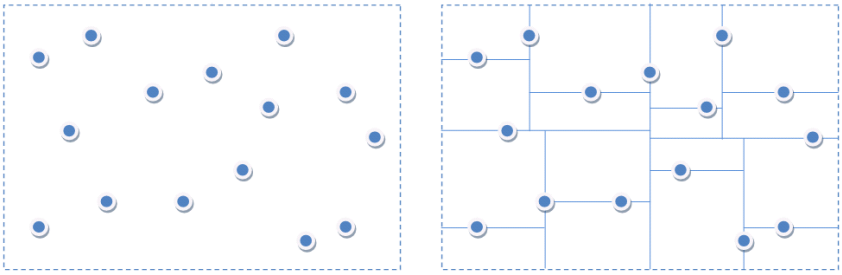
### 2.4.4 Priority queues

Basic search methods for kd-trees utilize priority queues for different measures. A priority queue is a sorted set of data where the elements are ordered by priority. The queues will most often have a fixed size where low priority elements are pushed out when higher priority elements are added to the queue.

A priority queue must per definition implement two methods:

- "**Insert**", which will add an element to the queue with a specified priority. The element with the lowest priority will be removed if the queue is full.
- "**Pop**" (**or** "**min**"), which removes and returns the element with the highest priority. This differs from normal queues or linked lists where the oldest element is returned regardless of priority.

Priority queues will most often also include a peek method, which returns the element with the highest priority without removing it from the queue. Priority queue implementations will generally have operations that run in $\mathcal{O}(\log n)$ time, where n is the size of the queue. Implementation details and more information about priority queues can be found in [27, 9].



(a) (b)

Figure 2.12: A kd-tree in $\Re^2$ before (*a*) and after (*b*) partitioning.

### 2.4.5 Kd-tree nearest neighbor search

Exact kd-tree nearest neighbor search is only faster than naïve search up to about twenty dimensions [17] with practical database sizes, and as such it becomes obvious that this does not hold for the descriptors used in SIFT or SURF. The algorithm is presented here for reference (see Algorithm 1).

A stack initially containing the root node of the kd-tree is used to keep track of which sub-trees to explore. If a child node is terminal, all elements in its bin are examined. Examined elements are placed in a priority queue, keeping the $n$ best matches based on distance from the target element. If a child node is not terminal, a bounding check is performed to see if it is intersected by a hypersphere with center in the target element and radius equal to the n:th currently best distance. If the node passes the bounding test, it is added to the stack and the search continues.

### 2.4.6 Best-bin-first search

Best-bin-first search was described by Beis and Lowe [5] as an approximate nearest neighbor search based on optimized kd-trees. It uses a priority queue instead of a stack to search branches in order of ascending distance from the target point and return the $m$ nearest neighbors with high probability, see Algorithm 2. The search is halted after a certain number $n_{\text{bins}}$ of bins have been explored, and an approximate answer is returned. According to [18], a cut-off after 200 searched bins in a database of 100, 000 keypoints provides a speedup over the exact search by 2 orders of magnitude, while maintaining 95% of the precision.

## 2.5 Random Sample Consensus (RANSAC)

RANSAC is an iterative method (see Algorithm 3) for estimating model parameters to observed data with outliers.

When using RANSAC to remove outliers from matching image points, a homog-

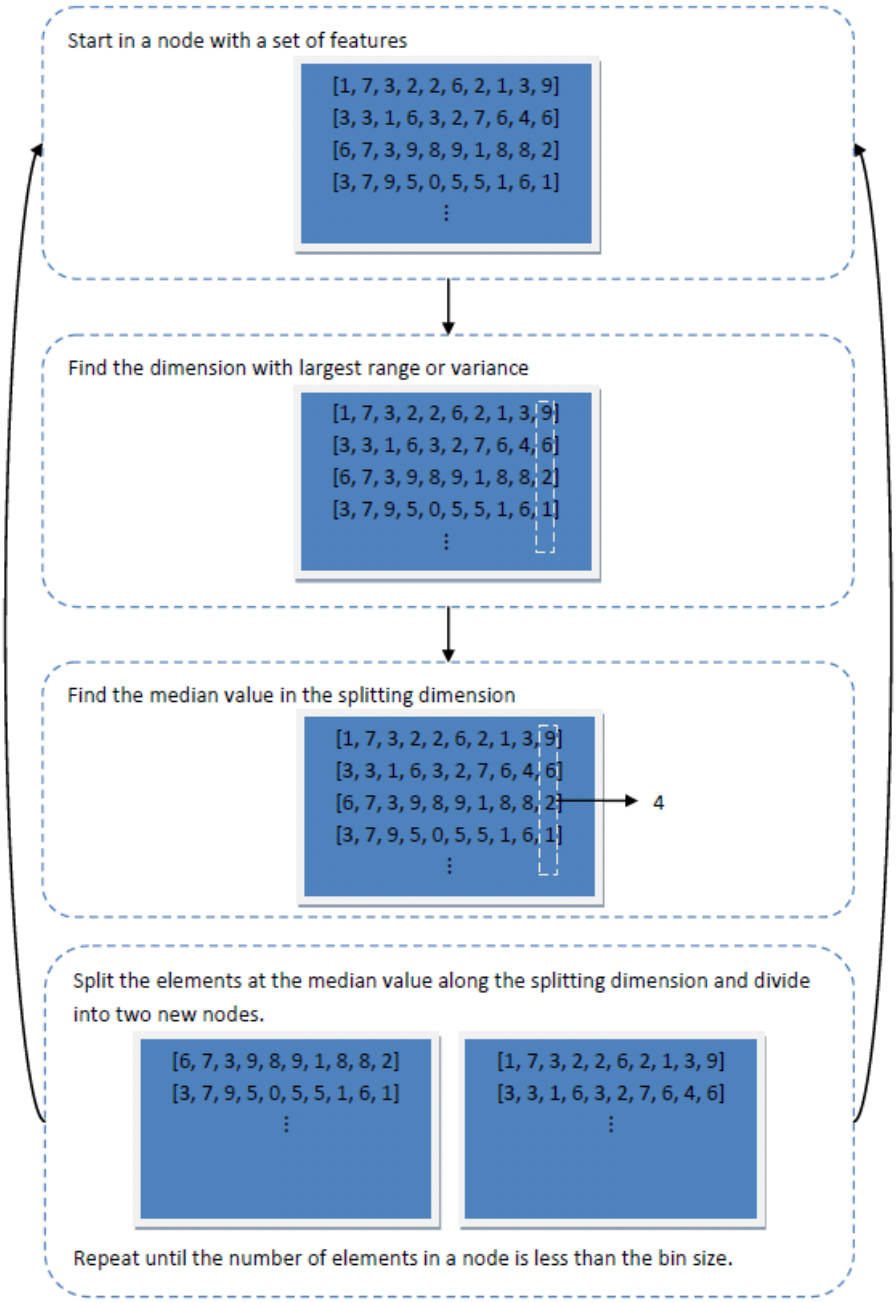



Figure 2.13: Kd-tree built in steps.

**Algorithm 1**: Kd-tree n-nearest neighbor algorithm

```
Input: Query element q
Input: Root node of an optimized kd-tree
Output: Priority queue R containing the n nearest neighbors
S = stack for nodes to explore
Put the root node on top of S
while S is not empty do
    Pop the current node from S
    foreach Child node C of the current node do
        if C is a leaf node then
            Examine all elements in the bin and insert into R
        else
            r = the n:th currently best distance in the priority queue
            if C is intersected by a hypersphere of radius r around q then
                Add C to S
            end
        end
    end
end
return R
```

raphy matrix $\mathbf{H}$ relating the matches as

$$\mathbf{x}' = \mathbf{H}\mathbf{x}$$

is used as a model. The homography can be estimated using the Direct Linear Transformation (Section 2.6). The error measurement used in the algorithm is the reprojection error $||\mathbf{x}' - \mathbf{H}\mathbf{x}||_2$.

RANSAC works by randomly selecting a subset of the matching image points and estimating a homography to it. All points fitting the model with an error less than a certain threshold $\epsilon$ are determined to be inliers and placed in a consensus set.

If the number of points in the consensus set is above the number of inliers needed for the model to be considered good, the homography is re-estimated from all inliers and the algorithm terminated. If not, a new random subset is chosen and a new homography estimated.

When the maximum number of iterations has been reached, the homography is re-estimated from the largest consensus set and returned.

## 2.6 Direct linear transformation

The Direct Linear Transformation (DLT) can be used to find a homography $H$ between 2D point correspondences $\mathbf{x}_i$ and $\mathbf{x}_i'$ in homogeneous coordinates [11].

**Algorithm 2**: Best-bin-first $n$-nearest neighbor search

```
Input: Query element q
Input: Root node of an optimized kd-tree
Output: Priority queue R containing the n nearest neighbors
S = priority queue for nodes to explore
Put the root node in S
while S is not empty do
    Pop the current node from S
    foreach Child node C of the current node do
        if C is a leaf node then
            Examine all elements in the bin and insert into R
            if n_bins bins have been explored then
                return R
            end
        else
            r = the n:th currently best distance in the priority queue
            if C is intersected by a hypersphere of radius r around q then
                Add C to S
            end
        end
    end
end
return R
```

**Algorithm 3**: RANSAC for Homography estimation

**Input**: Set of observed data points
**Input**: Maximum number of iterations $n$
**Input**: Error threshold $\epsilon$
**Input**: Number of inliers $T$ needed to consider a model good
**Output**: The estimated homography
**for** *i = 1 to n* **do**
  Choose a subset $S$ of $s$ random sample points from the observed data
  Estimate the homography $H$ from $S$
  Add all points with a reprojection error less than $\epsilon$ to the consensus set $S_i$
  **if** size($S_i$) $> T$ **then**
    $S_{\text{best}} = S_i$
    break
  **else**
    **if** size($S_i$) $> S_{\text{best}}$ **then**
      $S_{\text{best}} = S_i$
    **end**
  **end**
**end**
Estimate $H_{\text{best}}$ from $S_{\text{best}}$
**return** $H_{\text{best}}$

**Estimating projective transformations with DLT**

Two sets $\mathbf{x}_i = (x_i, y_i, w_i)$ and $\mathbf{x}'_i = (x'_i, y'_i, w'_i)$, $i = 1, \ldots, n$, of corresponding points in homogeneous 2d coordinates are related by a planar projective transformation (Figure 2.14(c)) represented by a homography

$$\mathbf{x}'_i = \mathbf{H}\mathbf{x}_i,$$

where $H$ is

$$\mathbf{H} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix}.$$

Since the non-singular matrix $\mathbf{H}$ is homogeneous, it has eight degrees of freedom and thus requires four point correspondences to be determined up to scale.

When solving for $H$, the scale problem is easily remedied by noting that $\mathbf{x}'_i \times \mathbf{H}\mathbf{x}_i = \mathbf{0}$ or

$$\begin{pmatrix} y'_i \mathbf{h}^{3\mathrm{T}} \mathbf{x}_i - \mathbf{h}^{2\mathrm{T}} \mathbf{x}_i \\ \mathbf{h}^{1\mathrm{T}} \mathbf{x}_i - x'_i \mathbf{h}^{3\mathrm{T}} \mathbf{x}_i \\ x'_i \mathbf{h}^{2\mathrm{T}} \mathbf{x}_i - y'_i \mathbf{h}^{1\mathrm{T}} \mathbf{x}_i \end{pmatrix}$$

where $\mathbf{h}^{j\mathrm{T}} = (h_{j1}, h_{j2}, h_{j3})$. This can be rewritten as

$$\begin{bmatrix} \mathbf{0}^{\mathrm{T}} & -\mathbf{x}_i^{\mathrm{T}} & y'_i\mathbf{x}_i^{\mathrm{T}} \\ \mathbf{x}_i^{\mathrm{T}} & \mathbf{0}^{\mathrm{T}} & -x'_i\mathbf{x}_i^{\mathrm{T}} \\ -y'_i\mathbf{x}_i^{\mathrm{T}} & x'_i\mathbf{x}_i^{\mathrm{T}} & \mathbf{0}^{\mathrm{T}} \end{bmatrix} \begin{pmatrix} \mathbf{h}^1 \\ \mathbf{h}^2 \\ \mathbf{h}^3 \end{pmatrix} = \mathbf{A}_i\mathbf{h} = \mathbf{0}$$

which is a set of linear equations in $\mathbf{h}$ that can be solved using standard methods. Since we generally do not have exact point correspondences, an approximate solution can be found using for example Singular Value Decomposition [15],

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^{\mathrm{T}}.$$

The solution $\mathbf{h}$ is the the singular vector corresponding to the smallest singular value. If the singular values in $S$ are sorted in descending order, this is the last column of $V$. The resulting homography $\mathbf{H}$ is then given from $\mathbf{h}$.

To get robust estimation, all points should be normalized before estimating $\mathbf{H}$ [15]. The normalization is performed separately for $\mathbf{x}_i$ and $\mathbf{x}'_i$ by centering the points around the origin and scaling their mean distance from origin to $\sqrt{2}$.

If $\hat{\mathbf{x}}_i = \mathbf{T}x$ and $\hat{\mathbf{x}}'_i = \mathbf{T}'x'$ are the normalized points and $\hat{\mathbf{H}}$ is the homography from $\hat{\mathbf{x}}$ to $\hat{\mathbf{x}}'$, then $\mathbf{H} = \mathbf{T}'^{-1}\hat{\mathbf{H}}\mathbf{T}$ is the homography from $\mathbf{x}$ to $\mathbf{x}'$.

**Affine and similarity transformations**

An affine transformation maps parallel lines to parallel lines, but orthogonality is not preserved (as illustrated in Figure 2.14(b)). A two-dimensional affine transformation can be expressed as

$$\mathbf{H} = \begin{pmatrix} h_{11} & h_{12} & t_x \\ h_{21} & h_{22} & t_y \\ 0 & 0 & 1 \end{pmatrix},$$

where $t_x$ and $t_y$ is the translation. Estimation of the affine transformation is done in the same way as above, requiring three point correspondences since it has six degrees of freedom.

When only two matching points are available, it is still possible to estimate the translation $t_x$ and $t_y$, rotation $\theta$ and scale $s$ with a similarity transformation (see Figure 2.14(a))

$$\begin{pmatrix} s\cos\theta & -s\sin\theta & t_x \\ s\sin\theta & s\cos\theta & t_y \\ 0 & 0 & 1 \end{pmatrix}.$$

The similarity transformation has four degrees of freedom and can thus be estimated from only two point correspondences.

(a) (b) (c)

Figure 2.14: Mona Lisa under a similarity transformation (a), affine transformation (b) and projective transformation (c).

# Chapter 3

# Applications, implementations and case studies

## 3.1 Writing good mobile Android code

### 3.1.1 The target hardware platform

The implementations of this thesis have been targeted at the Android Dev Phone 1. This is essentially an HTC Dream (G1) phone without operator locks. The relevant hardware specifications are:

- Qualcomm MSM7201A ARM11 processor at 528MHz
- 3.2 inch capacitive touch screen
- 192 MB DDR SDRAM and 256 MB Flash memory
- 3.2 megapixel camera with auto focus
- Quad band mobile network access and 802.11 b/g wireless LAN connectivity

### 3.1.2 General pointers for efficient computer vision code

#### Introduction

Handling images involves large matrices and complex algorithms.

One must understand the nature of the target platform in order to understand how to write efficient code. Incorrect handling of memory and floating point values can make a computer vision implementation run many times slower than it could do with rather simple code changes.

However, it should be noted that micro optimizations should not be used instead of algorithmic optimizations, but rather as a complement.

A general overview of the Android platform can be found in Appendix A.

### Handling matrices

Image handling is essentially handling large matrices and the implementations in this thesis are no exceptions. The Java collections package is very useful, but carries too much overhead in comparison to speed when it comes to computer vision. This is especially evident in mobile implementations. Using basic arrays is almost always preferred for the heavy algorithms. There are pitfalls even with arrays as data storage. Java can handle multi dimensional matrices with the syntax `array[m pos][n pos]`, but accessing these matrices is much slower than accessing a one dimensional array such as `array[pos]`. It is for this reason better to address two dimensional matrices through one single dimension of size $width * height$ by addressing as `array[`$row * width + col$`]`. This abovementioned layout of the array can and should of course be changed to fit the order in which memory is accessed. It is much faster to access elements in order `array[pos], array[pos+1], array[pos+2]` than for example `array[0], array[1000], array[2000]`. This might not seem completely obvious due to the $\mathcal{O}(1)$ nature of array access, but the access will be optimized by the caching functions of the processor and virtual machine and the next element will therefore often be available to the application even before a request has been sent to the memory pipeline. Accessing the array out of order will render constant cache misses which will lead to delays due to the slow nature of RAM in comparison to cache.

### Floating point operations

Computer vision implementations are most often heavily based on floating point values since most operations cannot be realized by integer numbers. Handling floating point values on a desktop computer is often as fast as integer numbers due to hardware implementations for fast floating point operators. Embedded devices such as mobile phones will however most often not include floating point compatible processors. Floating point operations will be implemented in software in such applications, rendering the operations many times slower than said operations using integers.

The magnitude of this obstacle can however be lowered by good programming practices. As with all mobile development one has to take limitations in to considerations continuously, and one way is to always use integers when possible. Programs can often contain thousands of unnecessary floating point operations on integer numbers or even countable numbers in decimal form. An obvious remedy is to realize countables such as $[0.1, 0.2, 0.3]$ as $[1, 2, 3]$.

If remedies cannot be found as easily as mentioned above one should consider using fixed point arithmethic. Specifying the position of the decimal point can often be done for a specific computer vision application, but limits of the input to said function must be implemented due to the risk of unwanted behavior in any realm where the fixed point operations give too inaccurate answers. Fixed point numbers can in many situations give more precise answers than floating point operations due to the inexact nature of floating point numbers, but if the fixed decimal point position is not carefully chosen to match the number of significant figures the results can lead to overflows and other unwanted behavior.

This leads to the conclusion that one should be aware and replace unnecessary floating point values and implement fixed point operations where speed is an issue.

### 3.1.3 Java and Android specific optimizations

#### Introduction

Since Android provides the possibility to implement Java code with standard industry methodologies, it is of course possible and sometimes preferable to use the same structure of implementation as one would on a normal desktop application. The target systems will however have the same hardware limitations as any other mobile platform does. This means that some measures have to be taken in order to produce code that can run smoothly. This problem is very evident when it comes to processes with high computational complexity such as object recognition.

The Android platform is built with a high priority on the keeping of a small memory footprint rather than keeping the processing speed up. This means that some common usages such as method nestling on the stack and context switching between different threads are implemented in a way that preserves memory rather than processes quickly [2].

#### Using static references when possible

Since the creation of objects and the keeping of said objects in memory are costly on the Android platform, the use of static methods where possible is recommended. This is of course in contrary to common Java methodology which means the developer has to compare the possible scenarios and use the most fitting method.

One has to remember that Java does not have explicit freeing of the memory. This is handled by a garbage collector which looks for unused objects and frees the memory they used when it is sure that this object cannot be used again. Static references can sometimes be useful on the Android platform in order to make sure that unused data is not kept in memory even though other parts of the same object is to be used later. The garbage collector will not be able to run smoothly in parallel if a lot of unused data is kept for a long time and later freed all together at the same time.

#### Referencing objects wisely

This section is connected to the previous section since it also concerns the garbage collector. The garbage collector will not be able to free large objects that are still used in parts. Keeping unnecessarily large object is a common error in Android graphics development. One will often find oneself in situations where the ”context” of the current activity has to be sent to a method that displays graphics. It will often be possible to send objects that are in a high level of the application and get the expected behavior, but this also means that said level cannot be freed by the garbage collector until the method is done. If one sends the lowest possible object level to the same graphics method there will be a greater possibility that most of the memory can be freed. An example can be seen in Listing 3.1. In this listing one can see that if the

high level object is passed on to gain access to the view, one will have to keep the useless objects until the high level object is released by the method it is passed on to. Referencing the low level object instead will make it possible to free the high level object. This problem can often be passed on in further calls, making the high level object live long past its recommended expiration time.

Listing 3.1: An example of object referencing

```
<High level object that is referenced instead of low level object>
        <Useless objects ought to be collected by the garbage collector>
        <Low level object which ought to be referenced instead of the high level object>
                <Useful view that the receiving method wants to gain access to>
```

Listing 3.2: Some loop examples

```
/* Example 1 shows a bad looping practice since the length of "vec"
will be referenced run and the addition of 2 will also be done each time. */
for (int i = 0; i < vec.length+2; i++) {
        //Do something
}

/* Example 2 shows how example 1 can be improved to run faster due to less
calls to the field length and less additions. */
int maxI = vec.length+2;
for (int i = 0; i < maxI; i++) {
        //Do something
}

/* Example 3 shows a bad looping practice for iteration over a Java
collections object. Calling hasNext() and next() on the iterator each time
will be very costly. */
Iterator<String> myIterator = myCollection.iterator();
while (myIterator.hasNext()){
        String myString = myIterator.next();
        //Do something with myString
}

/* Usage of the "for each" loop will most often make a local reference of the objects
in array form and thus keep the number of method calls at a minimum. */
for (String myString: myCollection) {
        //Do something with myString
}
```

### Looping wisely

The Dalvik compiler and verifier contains very few optimizations that most normal JREs do, which makes wise looping a high priority. This has lead to behavior such as the fact that the iterator concept is very slow on Android and should never be used. Manual caching of values such as loop stop values is recommended as well as the use

of the Java 1.5 ”for each” loop in the cases where the underlying implementation is free of time consuming procedures such as iterators. Listing 3.2 shows some examples of how large structures should and should not be iterated over.

If one needs to loop the same amount of times over multiple collections it is recommended to first retrieve the collection data as a local array with toArray() and retrieve data from these arrays instead. This will add the overhead of the new array, but since it will only contain references to objects it will take very little memory and the advantage of the speed up will be worth it.

#### Using native C/C++ through JNI for high performance methods

Although Android programs are written in Java it is possible to cross compile hardware specific code and utilize in Java using the Java Native Interface (JNI). In some implementations this will give a high performance boost. This could be especially interesting in computer vision applications.

There are some important points that one should know prior to JNI implementation:

- One must recognize the fact that the use of JNI somewhat defeats the purpose of Android's Java concept since the implementations will be at least processor architecture specific, and at most processor specific. This will however not be a big problem in many cases since the JNI code will most often be smaller code snippets that perform large loops. The code will most often be compilable on most platforms and such advanced solutions will probably often be bundled with hardware or at least specific for said hardware.

- Due to the fact that it is preferable not to use Java collections and similar in Android it is often simple to convert existing Java code to C code without much hassle. This code will of course not be optimized for C, but on the other hand it will still often be much faster than the equivalent Java implementation.

- A very important thing to remember is that if one is to optimize large loops with JNI one should include the loop in JNI and not just the calculations within. The large cost of the repeated context switches in Android will otherwise in many cases be higher than just running everything in Java.

- One must make sure that the native code is extremely stable. It must be tested for all possible input types. If the native code crashes it will destabilize the whole VM (Dalvik) and will most likely cause your application to crash without the possibility to display a message. This means that if the native code leads to a segmentation fault due to a bad memory pointer it will terminate the whole application. One must thus remember to check the input data in Java before the JNI call, because there is no exception handling to rely on.

## 3.2 SURF implementations

### 3.2.1 JSurf

JSurf is our Java implementation of SURF. JSurf was originally a port of OpenSURF 1.2.1. We have since rewritten the whole library to implement our own optimizations specific for Java, instead of following the development of OpenSURF. Our aim is to release it as open source.

### 3.2.2 AndSurf

AndSurf is an Android optimized version of JSurf. AndSurf is fitted to overcome the algorithmic flaws and choices of the Android platform. This includes inlining of methods to reduce context switches and removal of all Java collections and other imports in favor to primitive types such as integer arrays and float arrays.

### 3.2.3 Native AndSurf (JNI, C)

AndSurf was very slightly modified to become C code instead of Java and included through the JNI interface. This proved slightly faster than the Java implementation on a desktop computer and many times faster than the Java implementation on the mobile phone. This proves that native code still is important for slightly heavy computations on mobile devices. One should however still do most of the application in Java and use the native code to overcome certain obstacles when needed.

### 3.2.4 A comparison of implementations

To benchmark our SURF implementations we chose to compare our implementations to each other and to certain known implementations. All benchmarks are done with SURF upright descriptors.

In the desktop case we chose to compare AndSurf, JSurf, AndSurf in native code, OpenSURF, OpenCV's implementation of SURF and the original SURF implementation (available in binary format). This can be seen in Figure 3.1. All native implementations used comparable compiler optimizations. It is interesting to notice that C implementation outranks all the other implementations and that our Java implementations are comparable for small images and as fast or faster than known C implementations on larger images.

In the mobile case we chose to compare AndSurf, JSurf and our AndSurf JNI implementations. This can be seen in Figure 3.2. The tests were done on the Android Developer Phone 1 (ADP1), which is essentially a HTC G1 ("Dream"). It is notable that the native implementation is approximately ten times faster than the Java implementations on the mobile phone, while on the desktop it is only 2-3 times faster.

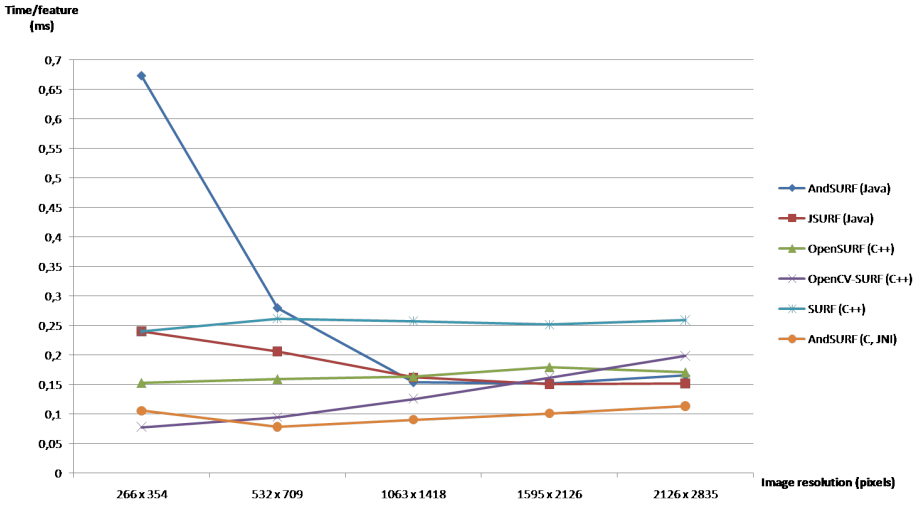



Figure 3.1: The average count of millisecond per discovered feature with comparable settings for different SURF implementations on a desktop computer.

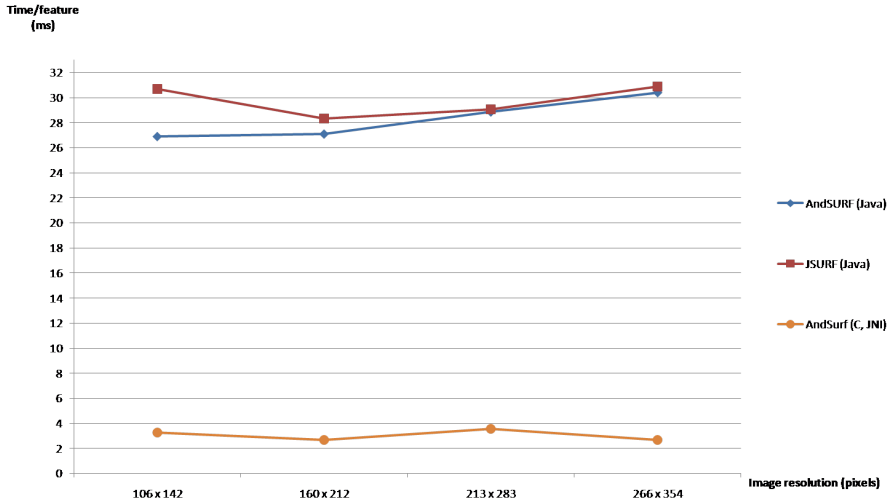



Figure 3.2: The average count of milliseconds per discovered feature with comparable settings for different SURF implementations on the Android Developer Phone 1, or HTC G1 (”Dream”).


## 3.3 Applications

### 3.3.1 Art Recognition

#### Purpose

The purpose of this application is to be able to identify paintings via the camera of the mobile phone. After identification the user should be presented with additional information regarding the painting and the painter. One should be able to read about other paintings by said painter and an additional function could be to receive suggestions on similar art.

#### Basic structure

The application is implemented as a client-server set-up. The client takes a picture of a painting and extracts local features, which are sent to the server. The server will compare these features against a database of known paintings and their local features. The server will then send formatted results with the relevant information regarding this painting.

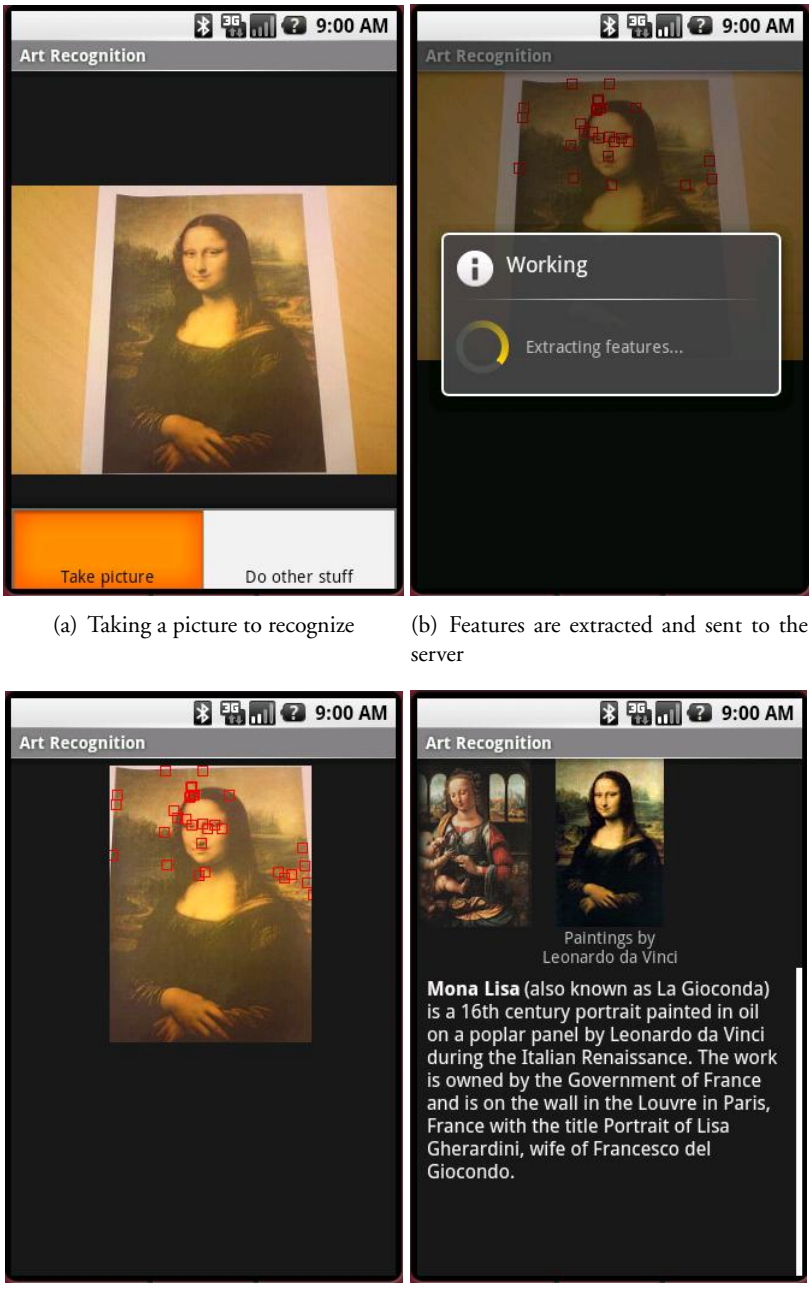
#### Taking a picture

As can be seen in Figure 3.3(a) a picture is taken using the camera of the mobile phone. This picture is then resized to a smaller size if it is unnecessarily large.

#### Extracting features and recognizing art

This is the heart of the application and can be seen in Figure 3.3(b). Instead of sending the picture to the server for feature extraction, the process is done on the phone. The phone will extract one feature at a time and immediately send it to the server, which performs recognition as soon as a feature is available.

Even though extraction is slower on the client than it would be on a server, it will allow for plenty more clients per server. As an added bonus the local features will be marked in the picture in a cognitively rewarding fashion as opposed to just waiting for the delays of the network and the server's feature extraction to be done.




(a) Taking a picture to recognize

(b) Features are extracted and sent to the server

(c) The calculated boundaries of the object are cut

(d) The results and suggested information is displayed

Figure 3.3: A picture of a painting is recognized with the phone.

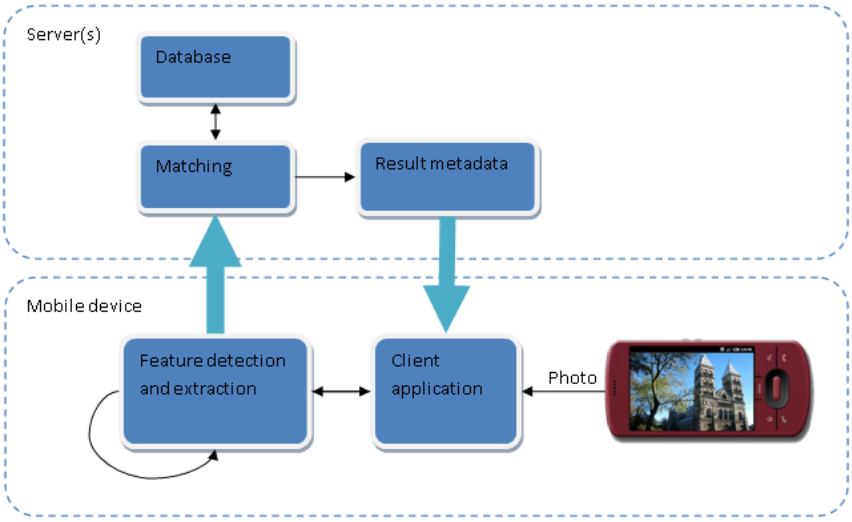



Figure 3.4: Basic structure of the ArtRec client-server set-up.

The feature extraction and comparison implementation is abstracted to support any kind of feature extraction. The current implementation supports both SIFT and SURF, but can easily be implemented for other methods.

A well modeled idea of this implementation is to make sure to account for every real time situation and never let the client idle when it could work on something that is needed later and never let it work when it is not needed. This can be noticed in the fact that the server will send a stop signal to the client immediately when it has found a match to make sure that the client does not spend unnecessary time on the feature extraction. This means that feature extraction can be done with high precision and still at a high speed. Another thought of feature is that the server could abort the recognition if it deems it to unlikely that it will find a match further on.

### Cutting the frame

When the server has found a match it will attempt to help the client with the cutting of the paintings frame/border. The server will find the affine transformation matrix from the known painting's geometry to the taken picture's geometry. When the server has this affine transformation matrix it will find where the corner points $(0, 0)$ and $(< paintingwidth >, < paintingheight >)$ from the trained painting ought to be in the taken picture. Theses corner points are used in the client as cutting points to remove useless pixels and make a smooth transition to the gallery possible. Figure 3.3(c) shows and example of where an image has been cut according to the transformation matrix that was found on the server.

### Displaying the gallery

If a result is found the application will translate and resize the cut picture with an animation that fades into the correct gallery position. The gallery can be seen in Figure 3.3(d) and it displays information about the recognized painting and also about other paintings by this painter.

### An example session

The user will select which method it prefers and send the command using a protocol as below:

```
beginRecognition SURF clientside

<local feature>
<local feature>
...
<local feature>
endRecognition
```

The above example assumes that all local features have to be sent. In a normal case where a match has been found prematurely it will break when possible (prior to `endRecognition`).

In the case where the server has found a match it will send an answer as below:

```
beginRecResponse
2,10,100,12
leonardo_da_vinci.mona_lisa.jpg
endRecResponse
```

Where `2,10,100,12` are the cutting points ($x_0$,$y_0$,$x_1$,$y_1$) and `leonardo_da_vinci.mona_lisa.jpg` tells the client where to look for painter information and which painting of the painter one has photographed. The painting information is gathered from a web server with known directory syntax for fast fetching and formatting using XML and HTML.

There is support for both setting which local feature extraction method to use and also sending settings to said method as can be seen in the above examples where SURF is the method and no settings are sent (blank line).

### Evaluation

We began by modifying an available SIFT implementation to fit our needs. This included some optimizations and behavior modifications to be able to interact with our thought out program structure. Extracting feature from an image on the phone using SIFT proved very time consuming even with our optimizations.

Our next step was to look for an available Java implementation of the SURF algorithm, which is faster than SIFT in most cases. Since no satisfying implementation

was available, we implemented SURF in Java as mentioned in Section 3.2.1. We noticed that the preparation of the determinant of Hessian matrix took a large portion of the application time even though only parts of the matrix were needed if sufficient information was found in a painting. Since experienced speed was a large issue, we opted to fill portions of the determinant matrix when needed in the extraction process. The extraction process continuously sends features to the server, which performs comparisons to known features immediately. When sufficient information is available to the server, it will tell the client to stop extraction which leads to very few unnecessary computations, including determinant of Hessian computations.

It was noticed that the amount of false positives quickly grew as the database grew due to the simple hard limit for the number of feature matches. We realized that some kind of geometric comparison method such as RANSAC was needed to discard outliers and provide better answers in a database with more than a few pictures.

Yet another crucial problem was quickly noticed. It quickly became obvious that the search time on the server grew rapidly as the database increased. This showed that a simple exhaustive search using an Euclidean distance metric is too slow for a useful application with hundreds of pictures in the database. The search time for such a database can be seen in Figure 3.6(a).

### 3.3.2 Bartendroid

#### Purpose

The purpose of this application is to be able to identify multiple bottles in a picture and use that information in order to inform the user of possible drink combinations which can be mixed from said ingredient. After the recognition and suggestion finding is done the user is presented with a list of possible drinks and the ability to read the recipes and learn more about the beverages.


#### Limitations

This application requires much more feature extraction and also requires the possibility to filter the features more efficiently than the art recognition did. This is due to the fact that the input image will be able to have multiple object matching to different known object and that these individual object can be situated anywhere in the image. Art recognition will most often handle an input image that mainly consists of the painting and has very little noise. Since the art recognition

was dependent on the ability to stop finding features when the recognition was good enough and this implementation will have to continue searching for features until the extraction process is done it will not be plausible to do the feature extraction on the client side.

### Basic structure

The basic structure of the Bartendroid client-server setup can be seen in Figure 3.5.

The client snaps a picture of a collection of beverages, which is sent to the server for recognition. This means that the option to extract features locally is not available for reasons mentioned previously. The server will reply with identification names for the beverages and the client can ask the server for recipes using said beverages if it is interested.

### Extracting features and recognizing beverages

The server will receive an image in binary form and perform some kind of feature extraction on the image. These extracted features will be matched against a known set of features to find fitting match pairs. Since the amount of features required to be able to recognize many different sorts of beverages from somewhat different angles is very large, a naïve approach is not feasible. The implementation instead utilizes a kd-tree of known features and does a tree search for approximate matches with an acceptable error probability.

The search algorithm works by first finding all matching features in the feature database. These matches are sorted by which destination image in the database they match so that the located bottles can be separated from each other. If the amount of matches to a database image is high enough, which is approximately 10 good matches in our implementation, the matches are passed on to the next step.

Since the database must be able to contain hundreds of bottles there will often be incorrect matches that survive the initial thresholding. This means that a simple threshold for match count is not a good enough parameter for identification and further steps need to be taken to remove the outliers. RANSAC is used for each matching feature set to filter out the geometrically impossible matches such as mirrored point matches. Once again this will be approximate and not always give an answer, but with the appropriate amount of tries it will give an acceptable error rate. The matches that survived RANSAC is considered correct and the identification strings of the matching objects are sent to the client.

When testing Bartendroid we chose to place it on a Amazon Elastic Compute Cloud (EC2) server to imitate a real world application setup of varying database sizes. For more information on EC2 see Appendix B.1.1. This server was run on the EC2 ”High-CPU Medium Instance” with Ubuntu Linux. The server trained on different image set sizes and a kd-tree was built offline for fast searching when a request would come in. We began by using a simple exhaustive feature to feature Euclidean distance-match method like the server side of Art Recognition had used. This proved very slow as the database size increased and without RANSAC the results become less and less

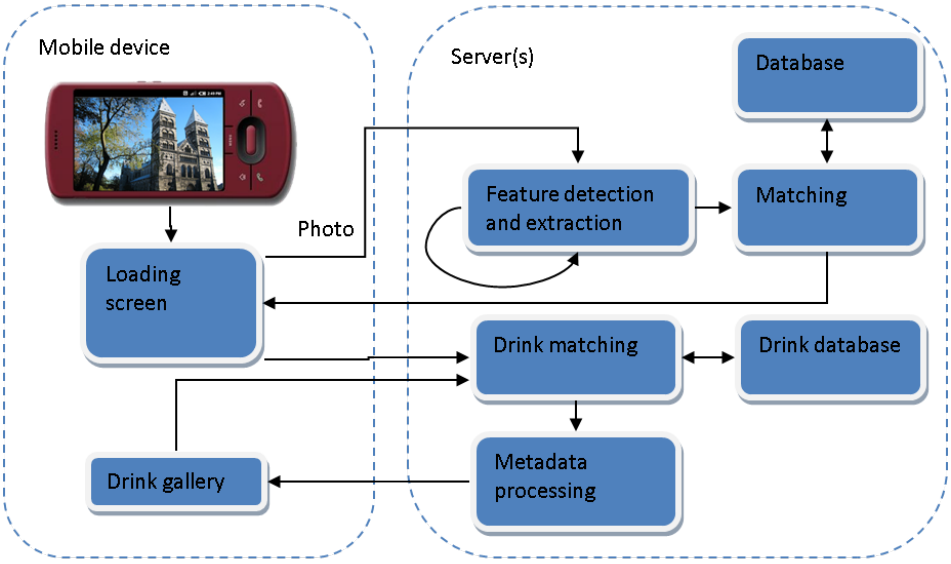


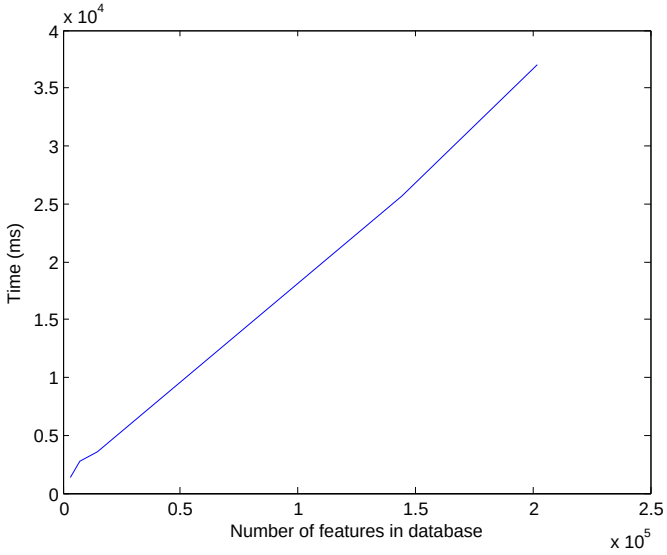

Figure 3.5: Basic structure of the Bartendroid client-server set-up.

reliable. In Figure 3.6(a) one can see the time needed for searching different database sizes on the EC2. As can be seen it scales very poorly since the search time is increased linearly as the amount of features is increased.

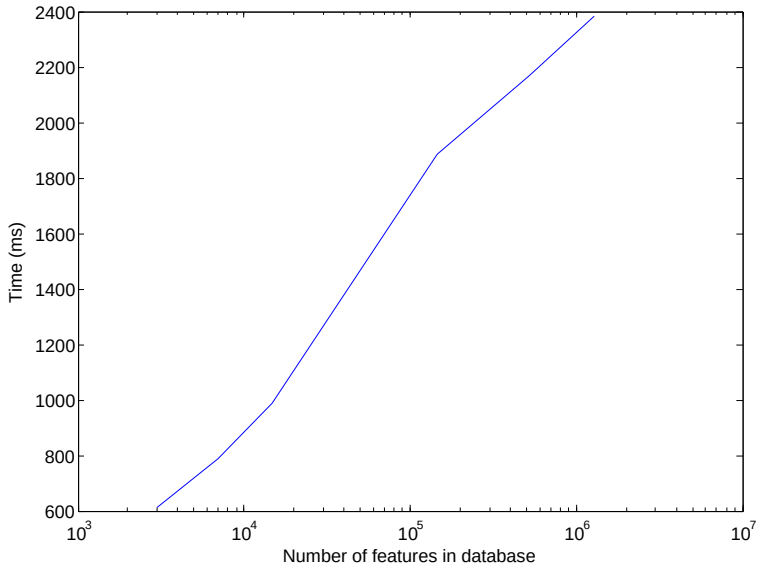
With these results in mind we turned to a best-bin-first kd-tree implementation (See Section 2.4.6) together with RANSAC for more useful and faster results. This would prove to be much faster and robust to different database sizes and image types. The end to end time from the photo being taken to the results being presented was approximately 10 seconds with most database cases on a wireless LAN network connection and it took approximately twice that time when sending pictures on a UMTS mobile connection. The interesting part is the combined time for search and RANSAC as the database is increased with large amounts of features from other objects than the sought after ones. In Figure 3.6(b) it can be seen that the search time seems to be increased logarithmically as the number of features is increased, which is seen as a line on the logarithmic scale. It should further be noted that the computation time seems to be approximately 2 seconds with a database of one million features. One million features would approximately be the amount needed to describe one thousand bottles. This implementation mostly uses one processor core of the two available, which means that two concurrent Bartendroid sessions could be running on the EC2 with the same separate speed as Figure 3.6(b) shows.

### Requesting drink suggestions

The client will receive the list of matching beverages. The client can choose to request drink matches for these beverages, a subset of the beverages or even other beverages.




(a) The exhaustive search time as the feature database is increased




(b) The best-bin-first kd-tree search and RANSAC time as the feature database is increased. Notice the logarithmic scale on the size axis.

Figure 3.6: The search time for the available methods of the ArtRecognition and Bartendroid server. An image will at an average consist of approximately 500 features. A bottle will be described by between one and four images in Bartendroid.


Figure 3.7: Screenshot from Bartendroid presenting drink suggestions.

In our implementation we simply ask the server to produce drink suggestions for the same beverage list as was received. Upon receiving a drink suggestion query the server will match the input beverages to a matrix of known drinks and their ingredients and return a list of drink identification strings ordered by the number of used ingredients. The suggestions need not contain all the ingredients.

### Displaying drink suggestions

The client will receive the list of drink suggestions in the form of identification strings. These identification strings are used to fetch the drink names in clear text and also names and pictures of bottles and drinks, full recipes and additional useful information. All this meta data is fetched from a web server using the identification strings in order to not strain the recognition server and to simplify information displaying using HTML. See Figure 3.7 for an example.

### An example session

The user will select which method it prefers and send the command using a protocol as below:

```
beginRecognition SURF clientside

<binary jpeg image>
endRecognition
```

When the server has read the image it will process it as mentioned in the previous section and in the case where the server has found matches it will send an answer as

below:

```
beginRecResponse

absolut_vodka
bombay_sapphire
stolichnaya_vodka
endRecResponse
```

The client will most likely ask for drinks that contain the found ingredients by sending the following to the server

```
beginDrinkFinding

absolut_vodka
bombay_sapphire
stolichnaya_vodka
endDrinkFinding
```

To which the server might reply

```
beginDrinkResponse

gin_and_tonic
backseat_boogie
vodka_cola
gin_gin
endDrinkResponse
```

The client will then handle the rows as mentioned in the previous section and display the information in a graphic fashion using meta data from a web server. At this point the client knows both the possible drinks and the names of all beverages involved. This could be used to save the recognized information in the user's virtual liquor cabinet for future mixing without the need to photograph every container every time.

**Evaluation**

Doing the extraction on the server is the right choice both from a speed point of view and a bandwidth point of view, since the transfer size of the local features will most often be as large or larger than the actual image. The use of tree search and RANSAC for filtering became crucial for this application since the uncertainty and slow speed of the simple original ArtRec method proved very slow and arbitrary on a large data set and with multiple objects per image.

### 3.3.3 AndTracks

#### Purpose

The purpose of this application is to find and track a single object in near real-time, using SURF computed entirely on the mobile device. This entails feature localization and matching of both the original object to recognize and the images in the video stream. These features also need to be matched on the mobile phone to produce tracking results.

#### Limitations

This kind of implementation needs to be done fully on the mobile device. This means that the performance of said implementation has to be excellent. The looping nature of such implementations puts constraints on performance in Java and Android. These obstacles must be overcome in order to make a tracking program useful, since a small delay is crucial. A short extraction time can further lead to an even higher frame rate than expected due to the fact that the tracker can be fairly certain of where the object might be and thus limit the search.

The user will select an object using the camera of the phone. This object will be recognized in the following camera preview frames if possible and marked for the user to see. The implementation can be seen in Figure 3.9.

#### Selecting an object to track

The user is presented with a screen where one can select what to recognize. A touch on the screen on an appropriate place will present a square which can be moved and resized using the fingers until it contains the appropriate object. The user then clicks the camera button, which snaps a photo. The selection can be seen in Figure 3.8.

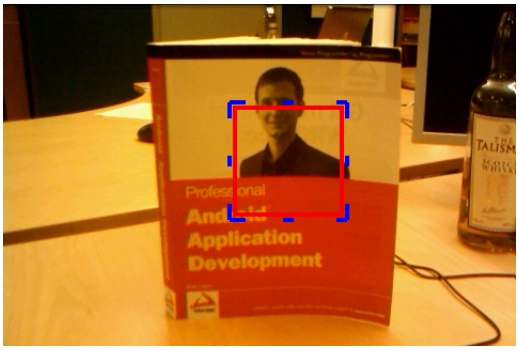
#### Extracting features from the known object

The camera focus and full resolution photography is used to get a clear picture of the object to recognize. This costs more time than using the cameras preview frames, but will only be done for the object image and provides better features due to the higher quality of the image.

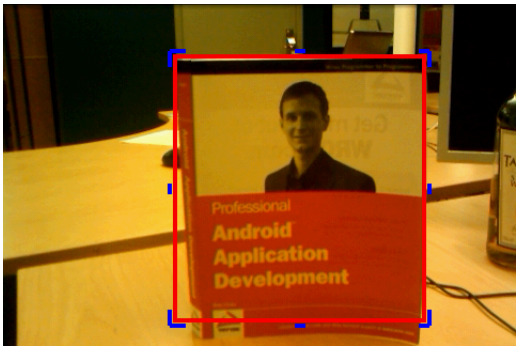
The feature extraction can be done using AndSurf in Java or native code or JSurf. In our initial attempt we used JSurf and quickly started implementing an Android evolution of this implementation using only primitive data types and most Android optimizations available. Later we implemented the same method using a version of the optimized code in native mode for the ARM 11 processor.

#### Extracting features from the video feed

The camera preview feed is used to track object since the low shutter speed and high resolution makes photography useless for tracking purposes other than to learn about the original object.



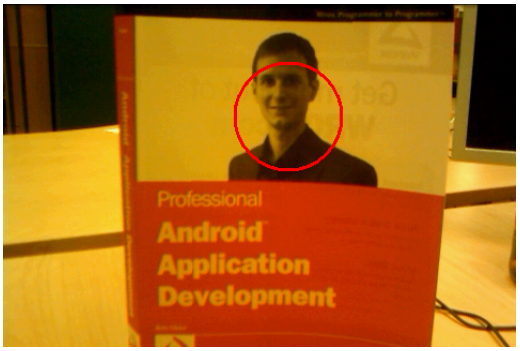

(a) The user clicks the screen to mark the object to be recognized




(b) The touch screen is used to resize and move the box to exactly fit the sought after object where after a picture is taken of said object.


(c) The selected object is found in the video stream


(d) The object is still found when the camera is moved




(e) Position updates are presented as soon as they are available from the recognition process

Figure 3.8: An object is selected which makes it possible for the mobile phone to recognize it in the following input stream from the camera.

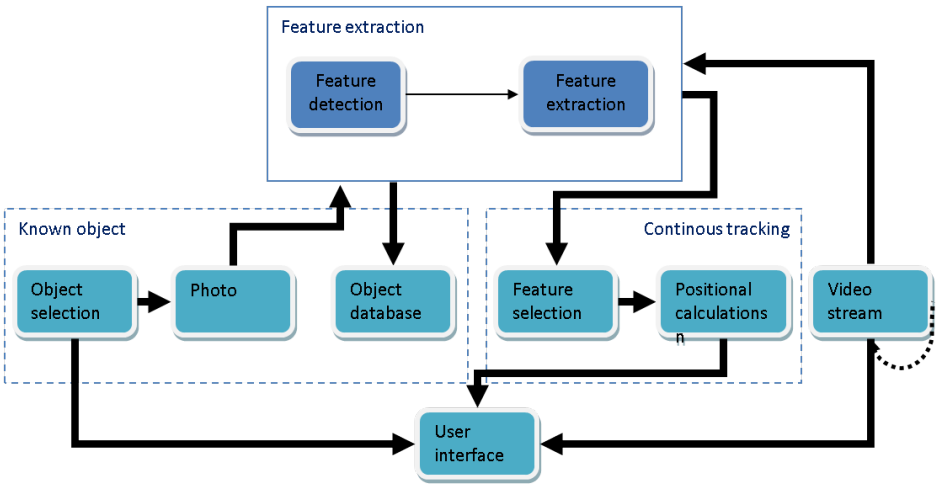



Figure 3.9: The structure of the tracking application

When a preview image is available it has to have features extracted for matching to be possible. Our initial implementation used the Android optimized Java implementation AndSurf, which proved much faster on the phone than our earlier implementations. This implementation still proved to be too slow for even near real time implementations. At this point we chose to implement a scaled down SURF method in native C code and compile optimized for the ARM 11 processor. This led to a huge speed-up, as can be seen in Figure 3.2.

### Finding the object

The object has to be localized in the video feed using the extracted features from the previous two steps. A simple brute force feature-to-feature Euclidean distance comparison was implemented since there will only be a few hundred features per image. This implementation was ported to native C code to speed up the matching process.

The matched features alone cannot be used to make a useful judgment of the object's location. The features have to be filtered geometrically using for example RANSAC in order to remove the false positives with a high probability. The original implementation of RANSAC that was used in Bartendroid proved too slow. A new and optimized implementation was therefore created to increase the performance.

One can assume that the object will be located near the position in the previous frame. The implementation looks for the object in the neighborhood of the previous match and expands the search area continuously until a match is found or the whole input is used. This leads to a large average speed up and a lower sensitivity to small movements of the object or the camera, but will also lead to the loss of frames when the search area is adapted to fast movements.

When benchmarking object tracking using AndTracks on one object laying on a monotone background we received the times in Figure 3.3.3 for different amounts of camera movement. The dotted lines leading up to crosses indicate where a match could not be found and the whole lines and circles indicate where a match was found, and thus the object was followed in the video stream.

It can be noted that the search algorithm which limits the cut window size seems to be slower than simply processing the full image for large movements. A more stable window size choice could be obtained by a more advanced control system. It should however also be remembered that the calculation of a full window takes so much time that a fast moving object will most likely have moved far away from said position even if there is a match in the image that was obtain prior to calculation.

#### Displaying the match

A red circle is displayed where the object is found. The location of the circle center is the average (X,Y)-position of the inlier features. This can be seen in Figure 3.8(c), 3.8(d) and 3.8(e).

#### Evaluation

The implementation shows that extraction and matching can be done on the Android platform. The implementation does not achieve real time functionality, but shows promise as a proof of concept.

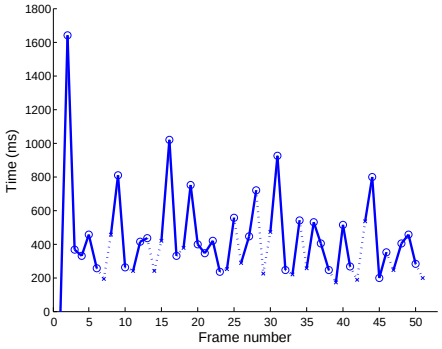
It became obvious that the implementation had to use native code to be useful, since Java implementations proved to slow. This will of course put a platform dependency on the implementation, but such will exist even in the Java case. Due to the varying nature of mobile phone cameras the implementation will have to be somewhat tweaked to be useful. Recompiling the standard C99 C code will be much quicker than for example optimizing the SURF parameters for the specific platform. A full platform fitting could however most likely be done in a few hours due to the general implementation of the program as a whole.

Object tracking using a native implementation on the Android Developer Phone 1 gave a typical processing time ranging between 200 milliseconds and two seconds per frame.

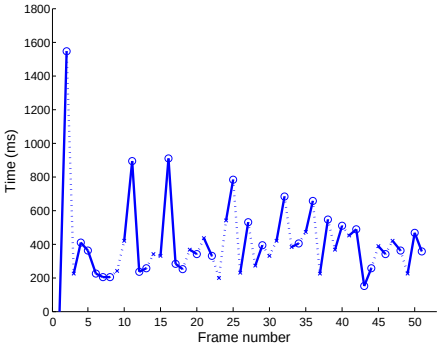
#### Possible uses of AndTracks

The use cases of tracking on mobile phones can be divided into two major sections:

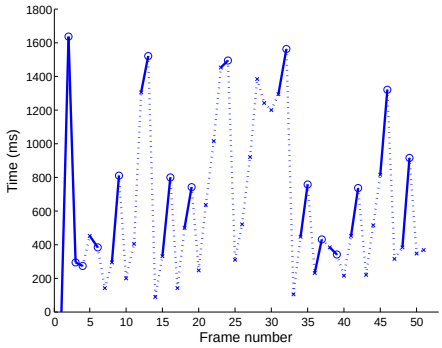
- **Augmented reality** is the case where the reality of the camera's video flow is altered in real time. An example of this would be presenting meta data such as names of sights in real time as the camera is moved around in a city or perhaps a museum. Our implementation is mainly suited for such specific object recognizing.




(a) An example with no camera movement




(b) An example with some camera movement. The oscillating camera movement had an approximate frequency of $\frac{1}{5}Hz$




(c) An example with much camera movement. The oscillating camera movement had an approximate frequency of $\frac{1}{2}Hz$

Figure 3.10: The time per frame for three AndTracks sessions. The circles and whole lines describe frames where the object was found and the dotted lines and crosses describe frames where no match was found. The camera was moved smoothly oscillating from left to right so that the object moved in the image plane from the left edge to the right edge.

- **Object tracking from a mounted video camera**. This can be very useful to model flows of objects or to notice when an object type is visible in the video flow. Our methods of object recognition are not optimized for this use case since this use case is object type specific, but not object specific.

**Performance increasing extensions**

The implementation has promising performance increasing possibilities. Some of them are:

- **Position estimation and filtering**. Since the camera knows where the object has been previously it is possible to estimate a likely position of the object at the current time. This can lead to the possibility to clip the input image for faster localization, extraction and matching. A simple position estimation like optical flow can be used to accommodate this, or even a more advanced filtering method such as Kalman filters [31].

- **Use of sensors to estimate position**. If the device has position sensors one can use these to increase the application performance. It can be used both to estimate the position and to cut an input image instead of or together with the previous example or just to improve the cognitive experience of the image overlay on the preview area. If a rectangle is chosen to illustrate the location, one can compensate for camera movement by moving the rectangle appropriately until a new image is available from the feature matcher. This will of course not compensate if the object moves instead of the camera, but it could greatly improve the delay experience for small camera movements.

- **Using pre defined objects to recognize**. If one wants to recognize certain object in the video stream that are perhaps not currently viewable one could teach the tracker objects that can be recognized in a manner like ArtRec and Bartendroid. The offline nature of such techniques makes it possible to provide proven robust features for better and quicker matching. Many different pictures and angles can be used to form 3D models or just to be faster. The features can be filtered to remove unused features when compared to known video streams. This will speed up the recognition process as well as increase the accuracy.

- **Live filtering of features**. The knowledge of the object is trained using a single image from a single angle. Just like when using pre defined objects one could make the features more robust even when the user selects an object if one uses live filtering. Since matches are found continuously in the video stream, it will become obvious which features are left unmatched. As the camera or the object moves the originally matching features will be less useful do to change of angles. This can be compensated by extending the known feature set with features from different angles that ought to belong to the object or by removal of features that are no longer in the cameras point of view. This will mean that matches and speed will increase gradually until an optimized feature set is achieved. Grading

features and removing useless features in each step could increase performance greatly.

- **Use of smaller feature descriptors**. If the descriptors are of a simpler nature, one can extract and compare features much faster. This will of course lead to more matching features, but if the inlier filtering with RANSAC is done properly, one can probably increase the overall speed while the accuracy remains at an equal level. It is also possible to utilize the repeating nature of a live video stream since the same descriptors will be calculated often. Caching of descriptors for the same feature can be done using the previously mentioned extensions.

# Chapter 4

# Conclusions

## 4.1 Evaluation of Android as a platform for Computer Vision applications

This thesis has proven that implementation of computer vision applications in a high level language such as Java on a modern mobile platform such as Android is possible if the correct considerations are taken. Even though the Java implementations are usable for certain applications it became obvious during the process that one has to implement some parts of the programs in native code if the applications are to work offline with reasonable speed.

One is often constrained by the computing power of the device, but the fast network speed of modern mobile networks and the seamless integration of applications into the Android operating system leads to the possibility to make such constraints less visible to the end user. Handheld devices do not only put constraints compared to a desktop computer, but they also provide added functionality such as high resolution cameras and in some cases sensors to assist calculations.

The user interface possibilities of the Android platform make it possible to keep the user busy while calculations are processed. This lowers the perceived runtime of the calculations and gives an overall positive cognitive experience.

## 4.2 Future work

There are plenty of promising future possibilities for mobile computer vision with platforms such as Android.

### 4.2.1 Increased computational power

The big issue is still the low computational power of mobile devices. As processors become more and more power efficient, they also become useful in mobile devices. We will probably see multi core processors with equal individual core clock frequency to that we have in our single core today. One can often trace the changes in the

desktop market a few years back to the awaiting changes in the mobile market. Many of the computer vision algorithms can be parallelized due to the fact that the cores can operate on different parts of the processed image or matrices obtained from the images. This opens for almost linear increase in computational power per added core.

### 4.2.2 Dedicated graphical processors

Certain modern mobile phones have dedicated graphical processing for 2D and 3D animation. This can be used to speed up certain matrix calculations, as mentioned earlier in the thesis. At the time of writing there are however only mobile phones with support for OpenGL ES 1.0. OpenGL ES 1.0 brings very little help that is useful in computer vision to the table. But promise lies ahead as Android will have support for OpenGL ES 2.0 [23] in the next version, named ”Donut” release of Android.

The OpenGL ES shader language (GLSL ES or ESSL) included in OpenGL ES 2.0 has many built-in functions for vector and matrix operations. In many cases these functions are SIMD (Single Instruction, Multiple Data) which can be performed element wise operations on vectors with no extra overhead compared to a single scalar operation.

### 4.2.3 Floating point processors

Since mobile devices normally lack processors with floating point operations, such operations have to be realized in software. This becomes a large constraint in computer vision applications since many of the applications need to use millions of calculations on numbers in decimal representation. Using a VFP co-processor will make floating point operations run even faster than fixed point converted operations.

### 4.2.4 Faster mobile access with 3G LTE

With 3G Long Term Evolution (LTE) mobile access, often dubbed ”4G”, one can send data at a much higher rate and often lower latencies. This opens up for faster seamless transfer to computational servers for some heavy computer vision needs. This will probably mostly be interesting when many combined operations such as recognition of multiple bottles in a large image are needed, rather than in real time video applications.

# References


[1] Amazon Elastic Cloud Compute. Amazon elastic cloud compute. `http://aws.amazon.com/documentation/`, June 2009.

[2] Android open source project. Designing for performance. `http://developer.android.com/guide/practices/design/performance.html`, April 2009.

[3] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. Speeded-up robust features (surf). *Computer Vision and Image Understanding*, 110(3):346–359, June 2008.

[4] H. Bay, B. Fasel, and L. Van Gool. Interactive museum guide: Fast and robust recognition of museum objects. In *Proceedings of the first International Workshop on Mobile Vision*, 2006.

[5] J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, page 1000. IEEE Computer Society, 1997.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[7] D. Bornstein. Dalvik VM internals. In *Google I/O Developer Conference*, May 2008.

[8] P. Brady. Anatomy & physiology of an Android. In *Google I/O Developer Conference*, May 2008.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[10] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the cloud: the montage example. In *Proceedings of the ACM/IEEE conference on Supercomputing*. IEEE Press, 2008.

[11] D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, August 2002.

[12] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.

[13] G. Fritz, C. Seifert, M. Kumar, and L. Paletta. Building detection from mobile imagery using informative SIFT descriptors. In *Proceedings of the Scandinavian Conference on Image Analysis*, pages 629–638, 2005.

[14] Google App Engine project. Google app engine documentation. `http://code.google.com/intl/sv-SE/appengine/docs/`, June 2009.

[15] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2000.

[16] B. Kisacanin, S. S. Bhattacharyya, and S. Chai, editors. *Embedded Computer Vision*. Springer Publishing Company, Inc., 2008.

[17] D. G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of International Conference on Computer Vision*, pages 1150—1157, 1999.

[18] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[19] R. Meier. *Professional Android application development*. Wiley Publishing, inc, Indianapolis, IN, USA, 2009.

[20] A. Neubeck and L. Van Gool. Efficient non-maximum suppression. In *Proceedings of the 18th International Conference on Pattern Recognition (ICPR)*, pages 850–855. IEEE Computer Society, 2006.

[21] M. Ozuysal, P. Fua, and V. Lepetit. Fast keypoint recognition in ten lines of code. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2007.

[22] T. A. O. S. Project. Dalvik bytecode verifier notes. `http://android.git.kernel.org/?p=platform/dalvik.git;a=blob_plain;f=docs/verifier.html;hb=HEAD`, february 2009.

[23] D. Sparks. Inclusion of OpenGL ES 2.0 hardware bindings in Android. `http://groups.google.com/group/android-developers/msg/32de221c98196581`, February 2009.

[24] Sun Grid Engine. Sun grid engine documentation. `http://www.sun.com/software/sge/techspec.xml`, June 2009.

[25] D.-N. Ta, W.-C. Chen, N. Gelfand, and K. Pulli. SURFTrac: Efficient tracking and continuous object recognition using local feature descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[26] G. Takacs, V. Chandrasekhar, N. Gelfand, Y. Xiong, W.-C. Chen, T. Bismpigiannis, R. Grzeszczuk, K. Pulli, and B. Girod. Outdoors augmented reality on mobile phone using loxel-based visual feature organization. In *Proceedings of the 1st ACM international conference on Multimedia Information Retrieval (MIR)*, pages 427–434. ACM, 2008.

[27] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (SFCS)*, pages 75–84. IEEE Computer Society, 1975.

[28] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

[29] P. Viola and M. Jones. Robust real-time object detection. In *International Journal of Computer Vision*, 2001.

[30] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, and D. Schmalstieg. Pose tracking from natural features on mobile phones. In *Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 125–134, 2008.

[31] A. Yilmaz, O. Javed, and M. Shah. Object tracking: A survey. *ACM Comput. Surv.*, 38(4):13, 2006.

# Appendix A

# Android

## A.1 Introduction

Android is a complete software platform for different sorts of devices. This means that it is an operating system and software stack to run Android specific software. It was introduced by Google in 2007 and is aimed at changing the way mobile applications are developed. The system is based on Linux and runs Java programmed applications on top. The big difference from normal mobile Java development is that Android supports a large part of the Java SE[1] through Apache Harmony instead of the crippled Java ME package. Android also does not use Sun's Java Virtual Machine or even Java byte code, but instead uses its own engine named 'Dalvik' together with its own set of byte code. This means that Android does not run Java applications, but rather Android applications programmed in the Java language with Android extensions [19, pages 1-3].

## A.2 The Linux kernel

Android is based on the Linux 2.6 kernel. Just as in any other Linux system the android kernel provides core system functionalities mostly linked to the hardware abstraction [19, page 13].

Even though Android is based on a Linux kernel, it is not a GNU/Linux, which is what is normally thought of in relation to the word 'Linux'. Android does not include all standard Linux utilities, it has no native window manager and it lacks compatibility with the standard C/C++ libraries of the GNU C library (glibc).

Android includes its own C library (libc) called Bionic. Bionic is optimized for embedded applications, with small and fast code paths and a custom implementation of POSIX threads (pthreads). Furthermore, it has built-in support for some Android specific features like system properties and logging. The reason that Bionic is not fully compatible with glibc that it lacks some POSIX features which are not needed in Android. These missing features are mostly concerning C++ libraries [8].

[1] Java SE is the Standard Edition that is normally used on desktop platforms.

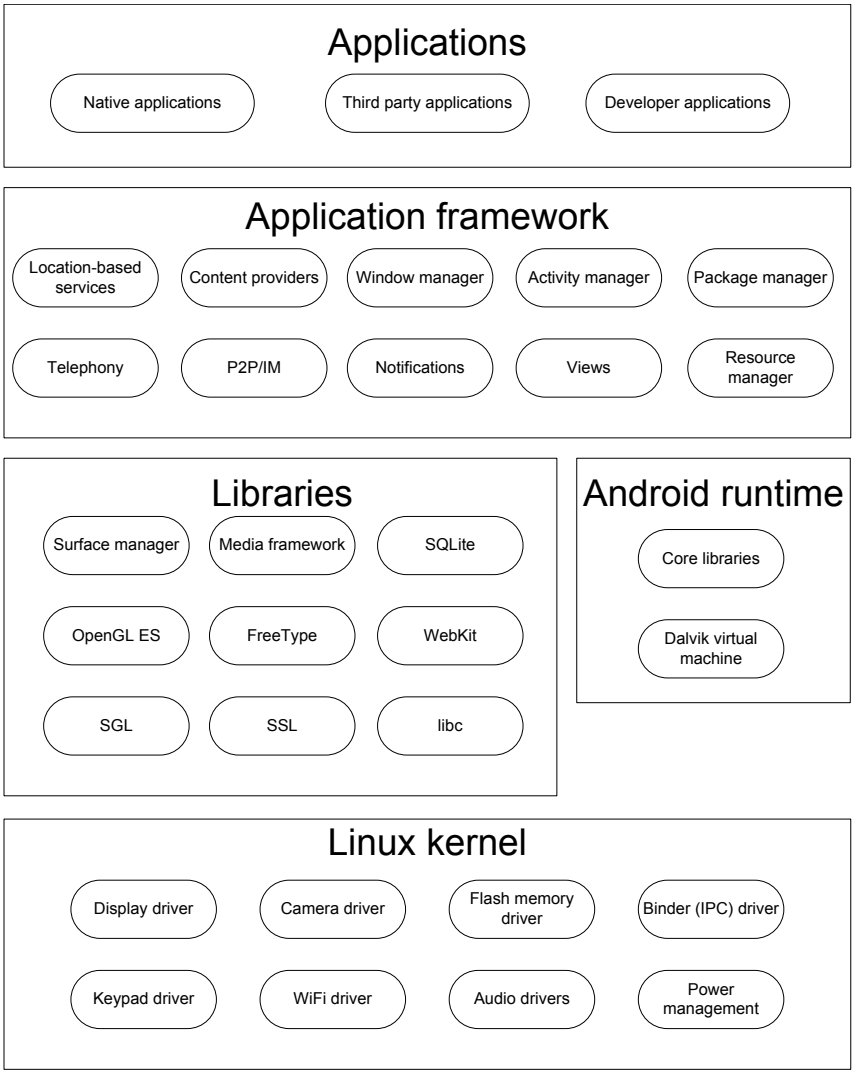



Figure A.1: The Android stack

## A.3 The system libraries

System libraries (written in C/C++) are exposed through the application framework and provide graphics capabilities, hardware acceleration etc. and also includes a fully functional SQL server (SQLite), SSL libraries for security, an OpenCORE based media framework and WebKit for web browsing [19, page 13].

## A.4 Android runtime

### A.4.1 Core libraries

Android implements a core set of standard Java libraries through Apache Harmony. This means that the Android Java library set differs from Java ME in that it is not sandboxed and stripped, but is rather 'desktop Java on a small screen'. It is almost fully compatible with Java SE 1.6 except for graphical parts, but in return adds its own graphical environment and a large collection of fitting help classes and implementations to speed up Android development.[19, page 13]

### A.4.2 Dalvik virtual machine

Dalvik is the Virtual Machine (VM) of the Android system. It is actually technically not a Java Virtual Machine (JVM) since it comes with a whole different instruction set than a JVM would interpret. Android applications are written in Java code, but compiled to Dalvik byte code.

Dalvik is a register based machine, as opposed to most JVMs which have a stack based structure. This structure choice will normally require larger instructions and thus often larger files, but that is not true in the case of Dalvik due to its trimmed down VM. Dalvik will also reduce the number of CPU instructions needed per VM instruction due to the fact that a register machine does not read from a stack and push the answer back but rather keeps addresses in instructions. This means that the Dalvik engine will generally be faster and load smaller files than a JVM would do. It should be noted that the Dalvik engine will not do some optimizations such as loop unrolling in the manner that a normal JDK/JVM would. This means that the programmer will have to do some extra optimizations and smart design choices [7].

### A.4.3 The code verifier

The local system (device) VM that loads the compiled Dalvik code (.dex) contains a system specific verifier. This verifier will check that the loaded code is type safe, is compatible with the device and also perform local and system specific optimizations of the byte code where possible [22].

## A.5 Application framework

The application framework contains the code base for hardware interaction, the specialized design patterns and other parts of the Android development methodology. The main parts are

- **Activity manager** handles the activities of your application.
- **Views** are the building blocks of UI creation and manipulation.
- **Notification manager** provides different types of signaling alternatives to make the user and device aware of changes.
- **Content providers** provide a simple way of sharing data between applications. A normal Android application has its own file system which cannot be manipulated from other applications. Data channels are instead opened through content providers.
- **Resource manager** handles the resources of the application. Application resources are entities such as UI elements, strings and other parts that can be externalized from the code into XML files and fetched when needed [19, pages 15,52-53].

## A.6 The structure of an application

"All applications are created equal."

There is no difference between built-in applications and third-party applications. It's possible to replace core apps such as the contact list, dialer or home screen.

An application can be built by one or more parts. These parts are defined in XML and described in this section.

Each application defines its permissions in the application manifest. Typical permissions are Internet access and camera usage. The user decides at install time if these needed permissions are acceptable. This means that the user will only agree to a permission once, but it will be granted for as long as the application is installed.

### A.6.1 Activity

An activity is a part of an application which displays a graphical interface of some sort and can retrieve input from the user. Most applications will be made of a collection of activities. These activities contain views to display information. A very simple example of an activity can be seen in Listing A.1.

### A.6.2 Service

Activities can contain services. Services are background processes which can run during a long period of time. These will most often sleep when there is nothing to be

done and process information when wanted. This is somewhat like the concept of ”server daemons” in Unix. An example of a service use case would be donwloading weather information for display in a widget.

Listing A.1: Basic activity stub

```
public class MyActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

### A.6.3 Broadcast receiver

A broadcast receiver receives and may answer specific events. Broadcast receivers must process very quickly and are used as small background services which can run when called. An example would be a broadcast receiver which receives an event which says that a photo has been taken and logs this to a specific file.

### A.6.4 Content providers

Android applications do not share a file system beyond data saving on an SD card. Content providers are meant to serve applications with useful data of specific sorts. This data can for example be contacts in a phone book, bookmarks from a web browser etc.

## A.7 The life cycle of an application

### A.7.1 Intents

Applications are started from a system called ”Intents”. Intents define that an application is capable of handling a job. That job can be for example ”viewing a web page”, ”calling a phone number” or ”sending a text message”. When one application wants to perform a task that another application should implement it sends out an intent. If there is an application that can handle the intent it will be started. If there are more than one application that can handle an intent, one will get a menu to choose from or the preferred one will be selected automatically depending on the situation. Explicit staring of an application from for example an application menu is done by sending a specific intent to start said application.

### A.7.2 Knowing the state of an application

Android applications need to keep track of their state at all times. Instead of explicitly starting and stopping applications, the system is designed to be able to pause appli-

cations when they are not displayed and later possibly bring them back. This means that an application has to save its state when it gets a pause command or data might be lost if Android decides to shut down the application in order to save memory or battery power.

If an application does not correctly pause its execution when told to it will continue to run in background until it might be killed at any time without any further warning. This behavior makes it very simple to resume a session from the previously run since paused applications will keep their memory space until some other application claims it. It can also have the effect that applications will become useless when resumed unless first explicitly killed and later started again since clicking a ”back” button or similar and starting the application again will not have the expected effect of an application restart unless the applications is designed for this behavior.

# Appendix B

# Distributed computing in The Cloud

”The Cloud” is a metaphor for the Internet. The metaphor has become popular as of late to describe the fact that the user does not know where in the world his data is sent or received, nor should he or she need to know that [28]. Cloud computing has emerged as an alternative to the traditional setup of servers at local companies, schools etc. Instead of having the need to have a powerful server which is unused during long times one can rent computing power and bandwidth from a cloud computing service. This can be compared to super computers, but with the added possibility of fast internet access to users from around the world and the possibility to seamlessly scale the application as the user base or application evolves.

This type of setup leads to more efficient power usage, less server technicians and a system which can evolve and expand organically at a predictable cost.

Many different companies offer cloud computing services with some difference in target groups.

## B.1 Cloud computing services

### B.1.1 Amazon Elastic Compute Cloud (EC2)

The Amazon Elastic Computer cloud is a system where the user is granted one or more virtual servers to configure and run as one pleases [1]. To start a server one can choose among a list of common systems such as different Linux distributions and Microsoft Windows setups. A suitable hardware setup is chosen to accompany this software setup. The server can thereafter be started and configured as suitable. The system can be started in a data center in the USA or Europe depending on the administrator's choice.

A large difference from a normal server setup is that the virtual server has its data saved only when it is running. As the server is terminated it will no longer exist, and also not cost anything. This adds the possibility to start up a large number of new servers for whatever need the administrator might have. One must however remember

| Type | Memory | Storage | Platform | Cores | ECU/Core |
|---|---|---|---|---|---|
| Small | 1.7 GB | 160 GB | 32-bit | 1 | 1 |
| Large | 7.5 GB | 850 GB | 64-bit | 2 | 2 |
| Extra Large | 15 GB | 1690 GB | 64-bit | 4 | 2 |
| High-CPU Medium | 1.7 GB | 350 GB | 32-bit | 2 | 2.5 |
| High-CPU Extra Large | 7 GB | 1690 GB | 64-bit | 8 | 2.5 |

Table B.1: The available instance types at Amazon EC2.

to save data elsewhere since it is destroyed when the instance is terminated. If one has created an instance with some practical settings and for example one's favorite computer vision application it is possible to save the set up in order to be able to quickly start up a new identical server at any time. User data can be saved in the Amazon Simple Storage Service (S3) and accessed by all instances and even other services, making it a useful system for data storage and retrieval. S3 storage costs ranges from $0.18 to $0.12 per GB and has added transfer costs. Direct traffic between instances in the same EC2 network is free of cost.

See [10] for a more detailed analysis of EC2 for data intensive and computationally heavy applications.

#### Instance types and pricing

One EC2 Compute Unit (ECU) is approximately equal to the CPU capacity of a 2007 Xeon or Opteron processor. This means that the High-CPU set ups will be able to perform advanced computer vision calculations at speeds exceeding a top of the line desktop computer for each instances.

The cost for an EC2 server is charged per unit used with no minimal cost. The default Linux set up costs $0.10 per hour with added costs such as $0.10 per GB of incoming data. For a computer vision application one might consider the High-CPU Medium set up, which costs $0.20 per hour for a Linux set up, but is up to five times faster than the basic setup.

Multiple instances can be combined to a fast super computer at a low cost. One can have a basic set up and increase the computational power organically if the traffic increases, with a higher hourly cost as a result.

#### Publishing an application

The EC2 behaves as a normal server, which means that applications are started in the same way as on any other server system and accessed through the provided IP address. This makes it straight forward to publish the application for anyone with previous knowledge of the chosen operating system.

### B.1.2 Sun Grid Engine

Sun Grid Engine is a cloud computing system operated by Sun, which mainly focuses on batch jobs and other computationally intensive applications [24]. Grid is useful for scientific calculations and for example video rendering, but not for end user distribution. Large computations that will take hours or days to run on a desktop computer can be sent to the Grid for faster calculation as long as the application follows certain rules, such as the ability to run on the Solaris 10 operating system and an application size below 10 GB.

#### Pricing

Sun Grid computing is charged per CPU hour. Each hour costs $1. The calculations are not limited to a single CPU, but can be run as fast as needed with the cost based on the number of CPUs used. This means that if the application uses 60 CPUs for one minute it is charged as one hour.

#### Publishing an application

Sun Grid calculations are published to the Grid as batch jobs. A shell script that controls the data flows is specified together with the needed arguments for said script. The applications themselves need sometimes not implement much computation, but rather reference one of the many available libraries for computation and rendering, such as Calculix, Fractal and Blender.

### B.1.3 Google App Engine

Google App Engine is a cloud computing system operated by Google, which focuses on web applications that run in a limited time span [14]. Web applications on App Engine can be programmed in the Java or Python languages with simple support for modern Web 2.0 interfaces through Google Web Toolkit (GWT). This makes it suitable for scalable web applications, but less attractive for computer vision applications. Applications on the App Engine need to be accessed through the World Wide Web and are strictly limited in the choices of programming languages and other libraries. There are hard limits on request times and transfer sizes, making App Engine unsuitable for intense calculations.

#### Pricing

App Engine differs from Amazon EC2 and Sun Grid in that the basic system is free of charge, with the added possibility of higher limits at a price. The free setup will stop working when certain limits are met. The price of added computational power is $.10-$.12 per CPU hour and $0.10 per incoming GB of data. Some limits are not possible to overcome even with a non-free plan.

#### Publishing an application

A web application for App engine is pushed to the service using Google's provided software. No explicit server configuration is needed, or possible. The user is given a web address where the application can be accessed as web page.

### B.1.4 Discussion

This section has discussed various systems for cloud computing. The three provided examples will often differ greatly in application development and execution, which means that one ought to take note of where the application is to be run prior to development start.

In this thesis we have used the Amazon Elastic Compute Cloud (EC2) for the Bartendroid server in Section 3.3.2. This was a choice that grew out of the interest to investigate an industry size system and the possibilities of distribution. Using EC2 is excellent for a service, since one can configure the host system as needed and thus run most applications that are wanted. This means that the EC2 can have support for legacy code if it is configured to have that.

Our position is that the EC2 is the best solution for services that rely on heavy computing, while Sun Grid is the best solution for very heavy batch jobs such as rendering and Google App Engine is most suitable for web based applications that will finish within a limited time. Due to the advanced possibilities and APIs of App Engine, one can for example build a web application that calls EC2 for some heavy calculations and otherwise makes due with the available power of the App Engine. Our systems have not used web based access for the computer vision part, but it is probable that many applications will do that in order to get easy access to the application regardless of calling device type and operating system.

Master's Theses in Mathematical Sciences 2009:E22
ISSN 1404-6342

LUTFMA-3182-2009

Mathematics
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/