HOGESCHOOL VOOR WETENSCHAP & KUNST | DE NAYER INSTITUUT
SINT-KATELIJNE-WAVER

DEPARTEMENT INDUSTRIËLE WETENSCHAPPEN

MASTER IN DE INDUSTRIËLE WETENSCHAPPEN:

ELEKTRONICA-ICT

Image recognition on an Android mobile phone

Masterproef voorgedragen tot het
behalen van de graad en het diploma
van master / industrieel ingenieur
Academiejaar 2008-2009

Door: Ive Billiauws
Kristiaan Bonjean
Promotor hogeschool: dr.ir. T. Goedemé

Prologue

Although it bears our names, this dissertation was not possible without the efforts of following people, which we would like to thank profoundly.

First and foremost, we would like to express our gratitude to our promotor, T. Goedemé for making this thesis possible. As member of the Embedded Vision research cell of the Embedded System Design group EmSD at the De Nayer Instituut, he provided us with the necessary knowledge about image recognition. Also, he was the one who introduced us to the principle of color moment invariants. Furthermore, we would like to thank K. Buys for his efforts in trying to acquire an Android mobile phone for development of our application. We are also grateful to PICS nv, for printing this document.

Abstract

The goal of this thesis was creating a virtual museum guide application for the open source mobile phone OS Android. When a user takes a picture of a painting, our application searches related information in a database using image recognition. Since a user of the application can take a picture under different circumstances, the used image recognition algorithm had to be invariant to changes in illumination and viewpoint. The execution of the algorithm had to run on the mobile phone, so there was need for a lightweight image recognition algorithm. Therefore we implemented the technique using *color moment invariants*, a global feature of an image and therefore computationally light. As different invariants are available, we tested them in Matlab before implementing them in the Android application. A couple of these invariants can be grouped as a *feature vector*, identifying an image uniquely. By computing the distances between the vectors of an unknown image and known database images, a best match can be selected. The database image to which the distance is the smallest, is the best match. Sometimes, however, these matches are incorrect. These need to be rejected using a threshold value.

Android is very flexible and provides many tools for developing applications. This allowed to develop our museum guide application in a limited amount of time. We explored, evaluated and used many of Android's possibilities. During the development, we had to optimize the calculation of the invariants, because we noticed it was a computationally heavy task. We were able to reduce the computation time from 103s to 8s in the emulator. The emulator didn't have camera support at the time of publication and we didn't have access to a real mobile phone, so we were not able to do real-life tests. We did notice, however, that the results of the calculation and the general program flow of our application were correct.

Abstract (Dutch)

Het doel van onze thesis was het maken van een virtuele museumgids voor het open source mobiele telefoon OS Android. Wanneer een gebruiker een foto neemt van een schilderij, zoekt onze applicatie, gebruikmakend van beeldherkenning, de bijhorende informatie in een databank. Aangezien de gebruiker een foto kan nemen in verschillende omstandigheden, moet het gebruikte beeldherkenningsalgoritme invariant zijn voor veranderingen in belichting en kijkhoek. De uitvoering van het algoritme moest draaien op een mobiele telefoon, dus een lichtgewicht beeldherkenningsalgoritme was noodzakelijk. Daarom hebben we een techniek geïmplementeerd die gebruik maakt van *color moment invariants*. Dit zijn globale kenmerken van een afbeelding en daarom computationeel licht. Aangezien verschillende invarianten beschikbaar waren, hebben we ze getest in Matlab voordat we ze in onze Android applicatie hebben geïmlpementeerd. Een aantal van deze invarianten kunnen gegroepeerd worden in *feature vectoren*, dewelke een afbeelding op een unieke manier identificeren. Door de afstanden tussen vectoren van een onbekende afbeelding en bekende databankafbeeldingen te berekenen, kan de beste match gekozen worden. De databankafbeelding waarnaar de afstand het kleinste is, is de beste match. Toch kan deze match in sommige gevallen niet juist zijn. Daarom verwerpen we deze resultaten door gebruik te maken van een drempelwaarde.

Android is een heel flexibel OS en biedt vele hulpmiddelen voor het ontwerpen van applicaties. Dit maakte het mogelijk om onze museumgids in een beperke tijd te ontwikkelen. Gedurende de ontwikkelperiode moesten we de berekening van de invarianten optimaliseren. We hebben de berekeningstijd in de emulator kunnen verminderen van 103s naar 8s. De emulator had echter geen cameraondersteuning op het moment van publicatie en we beschikten ook niet over een echte mobiele telefoon, dus we hebben de applicatie nooit in een echte situatie kunnen testen. We hebben echter wel gemerkt dat de berekende resultaten en de program flow van de applicatie correct waren.

Korte samenvatting (Dutch)

In 2008 bracht Google zijn eerste echte versie van de Android SDK uit. Android is een open source besturingssysteem voor mobiele telefoons. Google heeft dit ontwikkeld in het kader van de Open Handset Alliance. Dit is een groep van meer dan 40 bedrijven die zijn samengekomen om de ontwikkeling van mobiele telefoons te verbeteren en te versnellen.

Voor onze thesis hebben wij een virtuele museumgids ontwikkeld voor dit platform. Dit houdt in dat een gebruiker met onze software een foto kan trekken van een schilderij, waarna er informatie over dit schilderij op zijn telefoon tevoorschijn komt. Hiervoor hadden we een beeldherkenningsalgoritme nodig dat bovendien geen al te zware berekeningen mocht gebruiken. Een mobiele telefoon heeft gelimiteerde hardwarespecificaties en zou voor het uitvoeren van de meestal zware "image recognition"-methodes te veel tijd nodig hebben. Daarom zijn we op zoek gegaan naar een lichtgewicht beeldherkenningsalgoritme. Een tweede eis voor het algoritme is dat herkenning moet kunnen gebeuren onafhankelijk van de belichting op het schilderij en de hoek waaronder de foto getrokken wordt.

Men kan twee grote groepen beeldherkenningsalgoritmes onderscheiden: algoritmes die gebruik maken van lokale herkenningspunten in een afbeelding, en algoritmes die eerder herkenning doen op basis van het totaalbeeld. Voorbeelden van de eerste groep zijn SIFT [8] en SURF [1]. De uitvoering van dit soort herkenning houdt in dat eerst lokale herkenningspunten, zoals bv. de hoek van een tafel, gezocht worden, waarna deze dan één voor één vergeleken worden. Dit was niet interessant voor ons omdat dit hoogstwaarschijnlijk te veel berekeningstijd zou vereisen. Daarom hebben wij geopteerd voor het herkennen op basis van het totaalbeeld van een afbeelding; meer specifiek de methode gebruikt in [9].

Het algoritme maakt gebruik van *generalized color moments*. Dit zijn integralen die berekend worden over de totale afbeelding. De formule hiervoor (2.1) bevat enkele parameters. Door deze aan te passen, kunnen een aantal verschillende integralen berekend worden voor dezelfde afbeelding. Deze kunnen dan op verschillende manieren gecombineerd worden om de zogenaamde *color moment invariants* te gaan berekenen. Deze invariants zijn formules die voor afbeeldingen van een bepaald schilderij waarden zouden moeten genereren die invariant zijn voor de verandering van belichting en kijkhoek. Uit onze tests is echter gebleken dat slechts enkele van deze invarianten echt invariant waren voor onze testafbeeldingen. Maar ook deze formules kenden een zekere spreiding van waardes voor gelijkende figuren. Daarom hebben we besloten een aantal verschillende invarianten te gaan groeperen in *feature vectors*. Zo'n vector is dan als het ware een handtekening voor een afbeelding. Het vergelijken van twee vectoren kon dan eenvoudig uitgevoerd worden door de afstand tussen beide te bepalen. Als een afbeelding nu vergeleken zou worden met een hele set van databankafbeeldingen, zou de databankafbeelding waartot de afstand het kleinst is de beste match zijn. Dit houdt echter in dat ook een match gevonden zou worden als er eigenlijk geen juiste match aanwezig was. Daarom hebben we besloten een drempelwaarde te bepalen, waaraan de match moest voldoen. Een absolute threshold waarde

zou niet nuttig zijn, omdat die waarde te hard zou afhangen van de aanwezige afbeeldingen in de databank. We besloten te werken met een methode waarbij de deling van de afstand van de beste match door die van de tweede beste match kleiner moest zijn dan de threshold waarde. Nadat we onze matching methode voor verschillende invarianten uitvoerig getest hadden, hebben we besloten te werken met feature vectors bestaande uit de invariant 3.1 voor de kleurbanden R,G en B. Hierbij hebben we ook ondervonden dat een drempelwaarde van 0.65 ideaal is om voldoende foutieve matches te verwerpen.

De volgende stap was het implementeren van onze matching methode in een Android applicatie. In Android wordt gebruikt gemaakt van de Dalvik virtual machine. Applicaties voor Android zijn geschreven in Java en XML. Bij het deployen ervan worden ze eerst omgezet naar *.dex-bestanden. Deze bestanden zijn bytecodes die gelezen kunnen worden door de Dalvik virtual machine en zijn geoptimaliseerd om uitgevoerd te worden op toestellen met beperkte hardware-specificaties.

Een eerste belangrijke stap was het ontwikkelen van een databank met schilderijgegevens. Hiervoor hebben we het SQLite databanksysteem in Android gebruikt. Voor het ontwerp van onze databank hebben we besloten de verschillende gegevens zo veel mogelijk op te splitsen in aparte tabellen. Dit gaf ons de mogelijkheid om later eenvoudig aanpassingen en uitbreidingen te kunnen doen aan onze applicatie. We hebben *painting*, *vector*, *info* en *museum* tabellen aangemaakt en verbonden zoals getoond in figuur 4.2. Omdat het voor museums eenvoudig moet zijn om zulke gegevensdatabank aan te maken, besloten we de mogelijkheid te geven om ze in een XML bestand te creëren. Een XML parser in onze applicatie kan zo'n XML bestand dan omzetten in databankrijen. We zijn ook op het idee gekomen om een kleine applicatie te schrijven die toelaat zulke XML bestanden snel en eenvoudig aan te passen, maar de ontwikkeling hiervan hebben we niet kunnen voltooien.
De implementatie van het matching algoritme op Android was op zich niet erg moeilijk, ware het niet dat de berekening van de feature vector van een afbeelding enorm lang duurde. Eerst duurde dit zelfs tot meer dan 1 minuut en 30 seconden op de Android emulator. Daarom hebben we heel wat optimalisatiestappen moeten ondernemen, waarbij we onder andere gebruik hebben gemaakt van de OpenGL|ES bibliotheek die in Android aanwezig is. Dit gaf de mogelijkheid om de berekeningen te versnellen door middel van matrixbewerkingen. Ook werd de hele manier waarop de invarianten berekend werden volledig herzien. Wanneer in onze applicatie een foto getrokken wordt, berekent de software nu in ongeveer 8 seconden de feature vector. Hierna worden dan alle vectoren uit de databank opgevraagd en worden ze één voor één vergeleken met die van de getrokken foto. Als een match gevonden wordt, zal de bijhorende informatie getoond worden op het scherm. Hiervoor hebben we in XML een design gemaakt van de layout, die dan door middel van Java-code gebruikt en ingevuld kan worden.

Voor het testen van onze applicatie hebben we gebruik gemaakt van de Android emulator die geleverd werd bij de SDK. Na verscheidene tests kunnen we concluderen dat onze applicatie in principe doet wat oorspronkelijk ons doel was. Als een foto getrokken wordt, berekent ons programma de juiste waarden voor de feature vector en daarna wordt gezocht naar de beste match in de database. De informatie die bij deze match hoort wordt ook mooi weergegeven. Hierbij moeten we echter wel opmerken dat we nooit "echte" tests hebben kunnen uitvoeren. In de Android emulator was er op moment van publicatie helaas nog geen support voor het gebruik van een echte camera. Dus wanneer in de emulator een foto getrokken wordt, wordt altijd éénzelfde voorgedefinieerde afbeelding gebruikt. We hebben tijdens onze thesisperiode nooit kunnen beschikken over een mobiele telefoon waarop Android draait. Daarom hebben we het principe van de virtuele museumgids ook niet echt kunnen testen. Wel hebben we de ingebakken testafbeelding door onze Matlab-code laten analyseren en we kregen dezelfde resultaten. Onze

implementatie in Android werkt dus. Daarnaast kunnen we ook niet 100% zeker zijn dat de applicatie ook goed zal werken op zo'n telefoon. We verwachten echter wel dat de berekeningen op een telefoon niet trager zijn, omdat in de emulator elke bewerking eerst altijd naar x86 code omgezet moest worden. In de processor die in de huidige Android-telefoons gebruikt wordt zit, immers een Java-versneller, dus we vermoeden dat onze applicatie op echte hardware goed en performant zal draaien.

Contents

Acronyms

AMuGu Android Museum Guide

ASL Apache Software License

GPL Free Software Foundation's General Public License

GUI Graphical User Interface

LUT Lookup Table

OS Operating System

OCR Optical Character Recognition

SIFT Scale-Invariant Feature Transform

SURF Speeded-Up Robust Features

List of Figures

List of Tables

Chapter 1

Goal of this thesis

This thesis is a part of the research done by the research cell Embedded Vision of the Embedded System Design group EmSD at the De Nayer Instituut, led by Toon Goedemé.

In 2008 Google released their first version of the Android SDK, which is a toolkit for building application for the open source mobile phone OS Android. The goal of our thesis is using the Android platform for developing a mobile phone virtual museum guide. Basically, a user should be able to take a picture of a painting and the application should react by showing information related to this painting.

For creating this application we'll have to find or create an image recognition algorithm ideal for our application. Our application demands an algorithm which doesn't take too long to execute. A user surely doesn't want to wait five minutes in front of each painting he's interested in. For our thesis we were asked to execute the algorithm on the phone and since mobile phones have limited hardware specifications, we'll have to find a lightweight algorithm.

Another part of our thesis is exploring the Android platform. Android is an open source platform, so it has a lot of possibilities. It encourages reuse of existing components, so we'll have to evaluate all of its options to select those things we need. By combining these existing building blocks with our own code, we should be able to create a fluent and easy-to-use application.

Chapter 2, our literature study, summarizes what we've found so far in existing literature. Chapter 3 and 4 are about the development of our application. The first sheds light on the research we've done in Matlab for determining the ideal algorithm for our application. The latter details

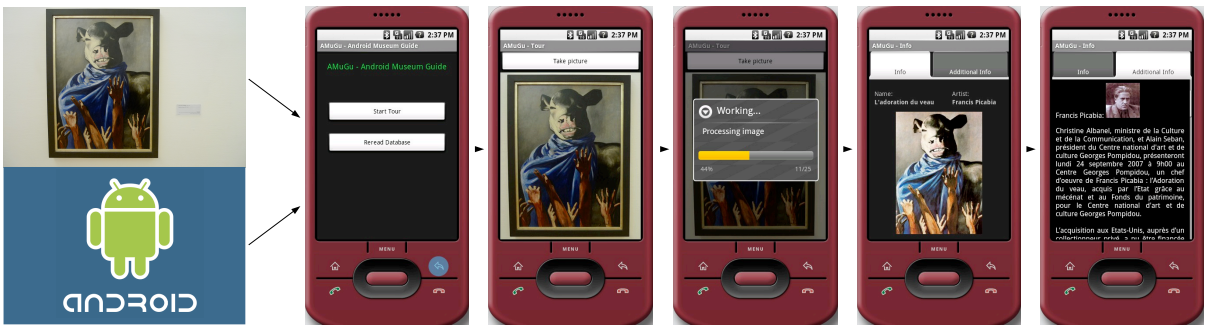


Figure 1.1: *Goal of our thesis.*

how we've implemented this in Android. We decided that museums who want to use our application should be able to easily create information about their paintings for our application. That's why we came up with the idea of a management interface written in Java. How we want to do this will be told in chapter 5. Because it originally was no part of our thesis goals, we haven't fully developed it yet. In the final chapter we'll summarize what we've realized so far and if it meets our original expectations.

Chapter 2

Literature study

Our thesis is about image recognition on an Android mobile phone. This chapter will cover two topics. First the basics of Android are explained in section 2.1: What is Android? How does it work? Next, an overview is given of the existing image recognition methods on mobile phones followed by a more detailed explanation of the method we used in 2.2. During our study, we came past a similar project called Phone Guide [3]. Section 2.3 provides a small case study about this project.

2.1 Android

Android is the first open source and free platform for mobile devices and is developed by members of the Open Handset Alliance [10]. The Open Handset Alliance is a group of over 40 companies, including Google, ASUS, Garmin, HTC,... These companies have come together to accelerate and improve the development of mobile devices. The fact that it's an open platform has many advantages:

- Consumers can buy more innovative mobile devices at lower prices.
- Mobile operators can easily customize their product lines and they will be able to offer newly developed services at a faster rate.
- Handset manufacturers will benefit from the lower software prices.
- Developers will be able to produce new software more easily, because they have full access to the system source code and extensive API documentation.
- ...

In [11] Ryan Paul gives an explanation concerning the open-source license of the Android project. As the main contributor to Android, Google decided to release this open source project under version 2 of the Apache Software License (ASL). ASL is a permissive license, which means that code released under this license can be used in closed source products, unlike copyleft licenses like GPL. This encourages more companies to invest in developing new applications and services for Android, because they can make profit from these enhancements to the platform. Note that the ASL only applies to the user-space components of the Android Platform. The underlying Linux kernel is licensed under GPLv2 and third-party applications can be released under any license.

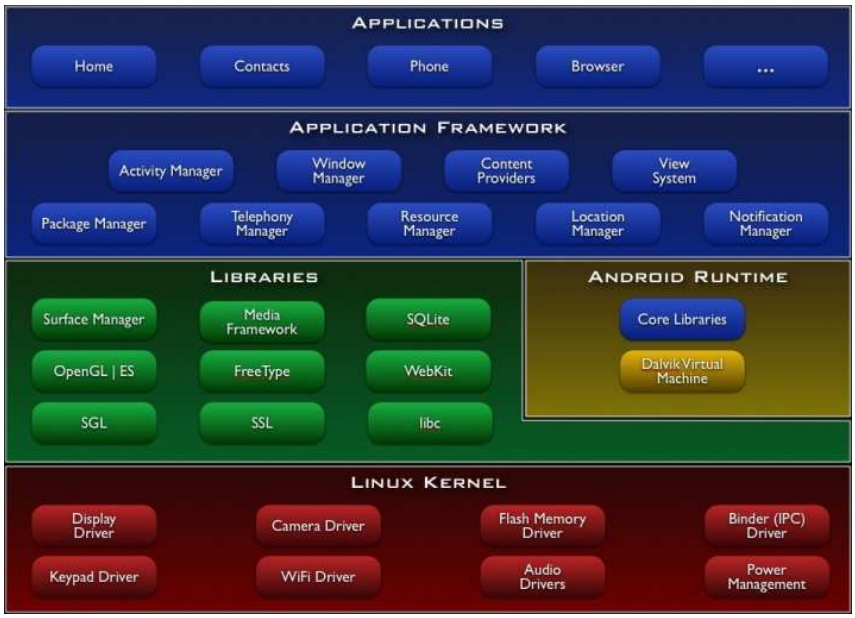


Figure 2.1: *System Architecture.*

In the following paragraphs we'll explain some details about the Android platform, based on the information found on the official Android website [6].

2.1.1 Architecture overview

Android is designed to be the complete stack of OS, Middleware and Applications. In figure 2.1 an overview of its components is shown.

Android uses the Linux 2.6 Kernel. Google decided to use Linux as Hardware Abstraction Layer because it has a working driver model and often existing drivers. It also provides memory management, process management, security, networking, ...

The next level contains the libraries. These are written in C/C++ and provide most of the core power of the Android platform. This level is composed of several different components. First there is the Surface Manager. This manager is used for displaying different windows at the right time. OpenGL|ES and SGL are the graphics libraries of the system. The first is a 3D-library, the latter a 2D-library. These can be combined to use 2D and 3D in the same application. The next component is the Media Framework. This framework is provided by a member of the Open Handset Alliance, Packet Video. It contains most of the codes needed to handle the most popular media formats. For example: MP3, MPEG4, AAC, ... FreeType renders all the bitmap and vector fonts provided in the Android platform. For datastorage, the SQLite database management system is used. Webkit is an open source browser engine and it's used as the core of the browser provided by Android. They optimized it to make it render well on small displays.

At the same level, the Android Runtime is located. Android Runtime contains one of the most important parts of Anroid: the Dalvik Virtual Machine. This Virtual Machine uses Dex-files. These files are bytecodes resulting from converting .jar-files at buildtime. Dex-files are built for running in limited environments. They are ideal for small processors, they use memory efficiently, their datastructures can be shared between different processes and they have a highly

CPU-optimized bytecode interpreter. Moreover, multiple instances of the Dalvik Virtual Machine can be run. In Android runtime we can also find the Core Libraries, containing all collection classes, utilities, I/O, ...

The Application Framework is written in Java and it's the toolkit that all applications use. These are applications that come with the phone, applications written by Google or applications written by home-users. The Activity Manager manages the life cycle of all applications (as will be explained in section 2.1.5). The Package Manager keeps track of the applications that are installed on the device. The Window Manager is a JPL abstraction on top of the lower level services provided by the Surface Manager. The Telephony Manager contains the API's for the phone applications. Content Providers form a framework that allows applications to share data with other applications. For example, the Contacts application can share all its info with other applications. The Resource Manager is used to store localised strings, bitmaps and other parts of an application that aren't code. The View System contains buttons and lists. It also handles event dispatching, layout, drawing,... The Location Manager, Notification Manager and XMPP Manager are also interesting API's which we'll describe in section 2.1.2.

The Application Layer contains all applications: Home, Contacts, Browser, user-written applications and so on. They all use the Application Framework.

2.1.2 API's

Section 2.1.1 listed some of the API's contained in the Application Framework. Now we'll go more into detail about the more important ones:

- Location Manager
- XMPP Service
- Notification Manager
- Views

The Location Manager allows you to receive geographic information. It can let you know your current location and through intents it can trigger notifications when you get close to certain places. These places could be pre-defined locations, the location of a friend and so on. The Location Manager determines your location based on the available resources. If your mobile phone has a GPS, the Location Manager can determine your location very detailed at any moment. Otherwise, your location will be determined through cell phone tower-id's or via possible geographic information contained in WiFi networks.

The XMPP Service is used to send data messages to any user running Android on his mobile phone. These data messages are Intents with name/value pairs. This data can be sent by any application, so it can contain any type of information. It can be used to develop multiplayer games, chat programs, ... But it can also be used to send geographic information to other users. For example, you can send your current location to your friends. In this way their Location Manager can notify your friends when you're in their neighborhood. The XMPP Service works with any gmail account, so applications can use it without having to build some kind of server infrastructures.

The Notification Manager allows to add notifications to the status bar. Notifications are added because of associated events (Intents) like new SMS messages, new voicemail messages and so

on. The power of the notifications on the status bar can be extended to any application. This allows developers to notify users about important events that occur in their application. Their are many types of notifications, whereas each type is represented by a certain icon on the status bar. By clicking the icon, the corresponding text can be shown. If you click it again, you will be taken to the application that created the notification.

Views are used to build applications out of standard components. Android provides list views, grid views, gallery views, buttons, checkboxes, ... The View System also supports multiple input methods, screen sizes and keyboard layouts. Further, there are two more innovative views available. The first one is the MapView, which is an implementation of the map rendering engine used in the Maps application. This view enables developers to embed maps in their applications to easily provide geographic information. The view can be controlled entirely by the application: setting locations, zooming, ... The second innovative view is the Web View. It allows to embed html content in your application.

2.1.3 Anatomy of an application

When writing an Android application, one has to keep in mind that an application can be divided into four components:

- Activity
- IntentReceiver
- Service
- ContentProvider

An Activity is a user interface. For example, the Mail application has 3 activities: the mail list, a window for reading mails and a window for composing mails. An IntentReceiver is registered code that runs when triggered by an external event. Through XML one can define which code has to be run under which conditions (network access, at a certain time, ...). A Service is a task that has no user interface. It's a long lived task that runs in the background. The ContentProvider is the part that allows to share data with other processes and applications. Applications can store data in any way they want, but to make it available to other applications they use the ContentProvider. Figure 2.2 shows this use of the ContentProvider.

An essential file in an Android application is the *AndroidManifest.xml* file. It contains information that the system must have before it can run the application's code. It describes the components mentioned above, for example it defines which activity will be started when the application is launched. It also declares different permissions, such as Internet access. The application name and icon used in the Home application are also defined in this file. When certain libraries are needed, it will be mentioned in here. These are only a few of the many things the *AndroidManifest.xml* files does.

Android is developed to encourage reusing and replacing components. Suppose we want to write an application that wants to pick a photo. Instead of writing our own code for picking that photo, we can make use of the built-in service. For that we need to make a request or an intent. The system will then define through the Package Manager which installed application would suit our needs best. It this case, that would be the built-in photogallery. If later on we decide to replace the built-in photogallery, our application will keep on working because the system will automatically pick this new application for the photo-picking.

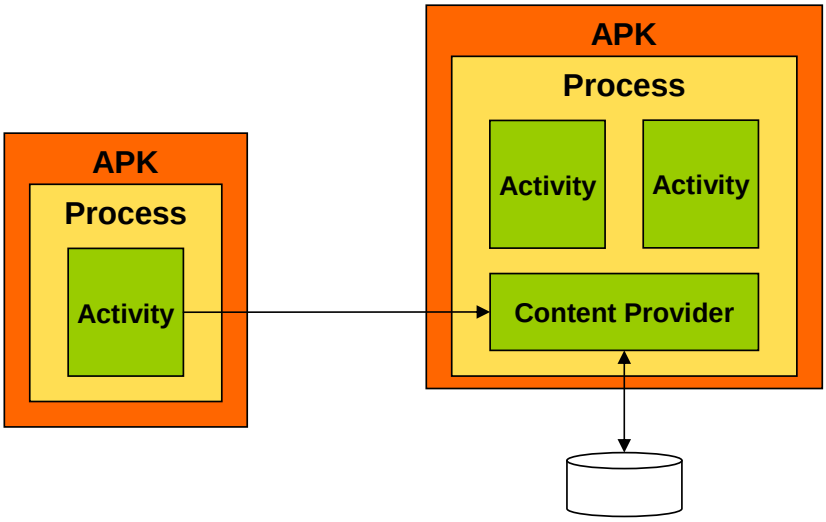


Figure 2.2: *Overview of the anatomy of a basic Android application*

2.1.4 Application model: applications, tasks, threads and processes

In an Android application, there are three important terms to consider:

- An Android package
- A task
- A process

Whereas in other operating systems there is a strong correlation between the executable image, the process, the icon and the application, in Android there is a more fluent coherence.

The Android package is a file containing the source code and resources of an application. So it's basically the form in which the applications are distributed to the users. After installing the application, a new icon will pop up on the screen to access it. Figure 2.3 shows how Android packages are built.

When clicking the icon, created when installing the package, a new window will pop up. The user will see this as an application, but actually it's a task they are dealing with. An Android package contains several activities, one of which is the top-level entry point. When clicking the icon, a task will be created for that activity. Each new activity started from there will run as a part of this task. Thus a task can be considered as an activity stack. A new task can be created with an activity Intent, which defines what task it is. This Intent can be started by using the Intent.FLAG_ACTIVITY_NEW_TASK flag. Activities started without this flag, will be added to the same task as the activity starting it. If the flag is being used, the task affinity, a unique static name for a task (default is the name of the Android package), will be used to determine if a task with the same affinity already exists. If not, the new task will be created. Otherwise the existing task will be brought forward and the activity will be added to its activity stack.

A process is something the user isn't aware of. Normally all the source code in an Android package is run in one low-level kernel process, as also is shown in figure 2.2. This behavior can be changed though, by using the process tag. The main uses of processes are:

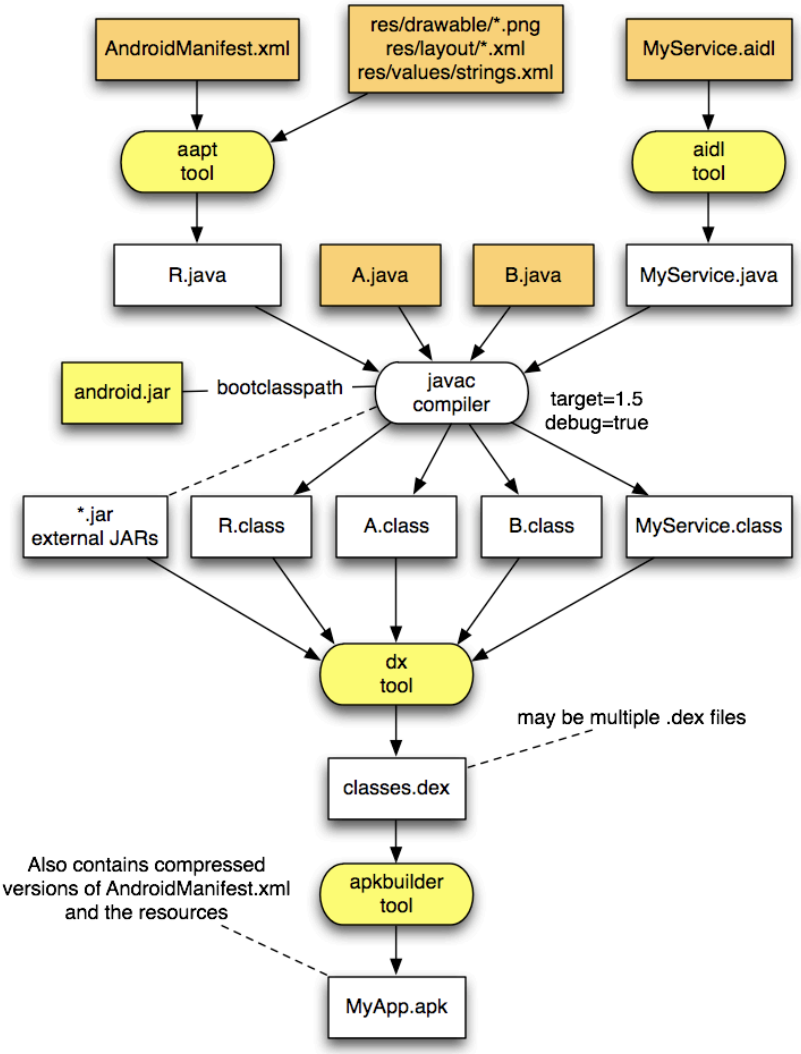


Figure 2.3: *Development flow of an Android package*

- Improving stability and security by placing untrusted or unstable code in separate processes.
- Reducing overhead by running code of multiple Android packages in the same process.
- Separating heavy-weight code in processes that can be killed at any time, without having to kill the whole application.

Note that two Android packages running with a different user-ID can not be ran in the same process to avoid violation of the system security. Each of these packages will get their own distinct process. Android tries to keep all of its running processes single-threaded unless an application creates its own additional threads. So when developing applications, it's important to remember that no new threads are created for each Activity, BroadcastReceiver, Service, or ContentProvider instance. Thus they shouldn't contain long or blocking operations, since they will block all other operations in the process.

2.1.5 The life cycle of an application

In these paragraphs we're going to explain how Android deals with different applications running at the same time. The difficulty here is that on Android, we have tightly limited resources available. Therefore, memory management is one of the most important features of the Android-system. Of course, the framework takes care of this, and the application programmer or end user doesn't even have to know about these things happening in the background. In order to provide a seamless and user-friendly interface, Android always provides the user with a "back" button, which works across applications. Whatever the user did, when he clicks the "back" button, the previous screen has to pop up.

A common example to demonstrate and explain this system is a user wanting to read his e-mail, click a link to a website, which opens in his browser. The user starts at the home screen, run by a "home" system process, where he clicks to open the mail application. The mail application is started by creating a system-process for the mail application, and by default it starts the "mail list" activity. An activity can be thought of as one window of a GUI. There, the user clicks on an e-mail. Before this new window for viewing the message is started, a "parcel" is saved in the system process for mail-list, and this parcel contains the current state of the mail-list activity. This is needed, because the system process might be erased from memory in the future (because another application needs memory), but when the user goes back to the mail-list activity, he expects to see the same window he saw before. Meaning: scrolled to the same page in the list, selected the correct item,... Everytime a new activity is started, either from the same application or another, the state-parcel is saved into the system process of the old activity. Back to the example, the message activity is opened and the user can read his e-mail message. The user clicks on the website URL inside the message, so we need to start the browser process. Also, the state of the message activity is saved in the mail system process, and the user gets to see the browser opening the URL. Now, suppose the website has a link to a map on the page. Yet again the user wants to look at this, he clicks, and a parcel with state information for the browser is saved in its system process. However now we have the home screen, the mail application and the browser running and we're out of memory. So, we have to kill an application in order to free space. The home screen can't be killed: we need it to keep the system responsive. The browser would also be a bad choice to kill, but the mail application is a couple positions back on the back-stack, and it's safe to kill. The maps application starts, creates a system process and an activity, and the user can navigate around.

Then, he wants to go back. So now the browsers activity is restored to the foreground. The state-parcel it saved, when the maps application was launched, can be discarded, as the browsers state information is still loaded. Going back on the back-stack, there is the message activity from the mail application. This process isn't running anymore, so we have to start it again. In order to do this we have to provide available memory and as the maps application isn't used anymore, it gets killed. So now mail is started again and opens a fresh message activity. This is an empty activity without any usable content to the user. However, in the system process we have saved the information parcel, and now it is used to fill in the message activity. Note that the parcel doesn't contain the content of the activity itself (the mail message) but only information needed to rebuild the original activity. In this case, this could be a description of how to get the mail message from a database. So the mail application restores the content of the message activity. The user sees the e-mail message exactly like it appeared the previous time, without knowing the activity was killed by loading the maps application!

2.2 Image recognition

Image processing can be described as every possible action performed on an image. This can be as simple as cropping an image, increasing contrast or scaling. Ever since digitalization of images came into the computer world, there was demand for image recognition. Image recognition is a classical problem in image processing. While humans can easily extract objects from an image, computers can't. For a computer an image is no more than a matrix of pixels. The artificial intelligence required for recognizing objects in this matrix, has to be created by a programmer. Therefore the most existing methods for this problem apply for specific objects or tasks only.

Examples:

- Optical character recognition or OCR transforms images into text.
- Recognizing human faces, often used in digital cameras.
- Pose estimation for estimating the orientation and position of an object to the camera.
- Content-based image retrieval for finding an image in a larger set of images.

Since we had to recognize paintings in an image, the technique we'll use is an example of the last mentioned image recognition task.

A widely used technique to recognize content in an image is shown in figure 2.4 and consists of the following two parts:

- Determining features specific for a certain image.
- Matching different images based on those features

In this section we'll take a look at a couple of frequently used techniques. Section 2.2.1 gives a summary of some techniques for determining features followed by a more detailed description of the method we'll use (i.e. color moment invariants). In 2.2.2 we'll explain some techniques that can be used to match images based on their features. We'll concentrate on matching techniques for images featured by color moment invariants.

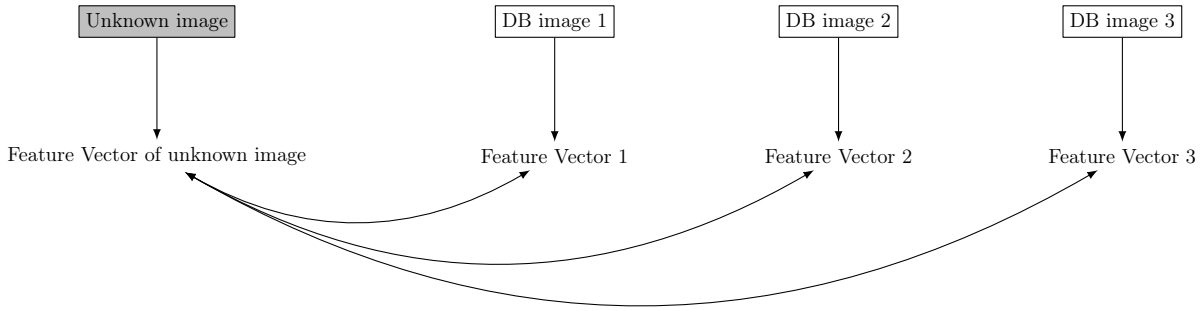


Figure 2.4: *Image recognition principle*

2.2.1 Determining features

To recognize an image we'll need to define some features that can be combined to create a unique identity for that image. Two matching images, for example two images of the same painting, should have nearly equal identities. So it's important that we'll use a robust technique to determine features that are invariant of environment conditions, like viewing-point and illumination.

2.2.1.1 Available methods

The easiest way of image recognition by far is pixel-by-pixel matching. Every pixel from the source image gets compared to the corresponding pixel in a library picture. The image has to be exactly the same in order to match. This method is not interesting at all, as it doesn't account for changes in viewing angle, lighting, or noise. Therefore, in order to compare pictures taken by an Android-cellphone with images in a database of museum-pieces, we need more advanced methods.

One method uses local features of an image; there are two commonly used algorithms:

- SIFT or Scale-invariant Feature Transform [8] is an algorithm that uses features based on the appearance of an object at certain interest points. These features are invariant of changes in noise, illumination and viewpoint. They are also relatively easy to extract, which makes them very suitable to match against large databases. These interest points or keypoints are detected by convolving an image with Gaussian filters at different scales resulting in Gaussian-blurred images. Keypoints are the maxima and minima of the difference of successive blurred images. However, this produces too many keypoints, so some will be rejected in later stages of the algorithm because they have low contrast or are localized along an edge.
- SURF or Speeded-Up Robust Features [1] is based on SIFT, but the algorithm is faster than SIFT and it provides better invariance of image transformations. It detects keypoints in images by using a Hessian-matrix. This technique for example is used in "Interactive Museum Guide: Fast and Robust Recognition of Museum Objects" [2].

The method that we'll use, generates invariant features based on color moments. This is a method using global features, and not based on local features as SIFT and SURF use. When using local features, every feature has to be compared, and processing time is significantly higher

than when using global features. Local features could be used though, and they will perform better on 3D objects. Our main goal however is 2D recognition, and we think color moments can handle this with a good balance of recognition rate and speed. Color moments combine the power of pixel coordinates and pixel intensities. These color moments can be combined in many different ways in order to create color moment invariants. In section 2.2.1.2 we'll give a more in-depth explanation of this technique.

2.2.1.2 Our method: Color moment invariants

The method we implemented, is based on the research done by Mindru et al. in [9]. In this section we explain our interpretation of that research.

For our thesis we want to be able to recognize paintings on photographs taken in different situations. This means that the paintings will undergo geometric and photometric transformations. Our way to deal with these transformations is deriving a set of invariant features. These features do not change when the photograph is taken under different angles and different lighting conditions. The invariants used in our thesis are based on generalized color moments. While color histograms characterize a picture based on its color distribution, generalized color moments combine the pixel coordinates and the color intensities. In this way they give information about the spatial layout of the colors.

Color moment invariants are features of a picture, based on generalized color moments, which are designed to be invariant of the viewpoint and illumination. The invariants used in [9] are grouped by the specific type of geometric and photometric transformation. We'll cover all groups (GPD, GPSO, PSO, PSO* and PAFF) and decide which group fits best for our application.

As the invariants are based on generalized color moments, we'll start by explaining what generalized color moments are. Consider an image with area Ω. By combining the coordinates x and y of each pixel with their respective color-intensities I_R, I_G and I_B we can calculate the generalized color moment:

$$M_{pq}^{abc} = \iint_{\Omega} x^p y^q I_R^a I_G^b I_B^c dxdy \tag{2.1}$$

In formula (2.1) the sum p+q determines the order and a+b+c determines the degree of the color moment. If we change these parameters, we can produce many different moments with their own meaning. Further we'll only use moments of maximum order 1 and maximum degree 2.

We'll give a little example to explain how these generalized color moments can be used for producing invariants. As mentioned above, different color moments can be created by changing the parameters a, b, c, p and q:

$$M_{10}^{100} = \iint_{\Omega} x I_R dxdy \tag{2.2}$$

$$M_{00}^{100} = \iint_{\Omega} I_R dxdy \tag{2.3}$$

Equation (2.2) results in the sum of the intensities of red of every pixel evaluated to its x coordinate. Equation (2.3) gives the total sum of the intensities of red of every pixel. Consider a painting illuminated with a red lamp, so every pixel's red intensity is scaled by a factor R. The

result of the equations (2.2) and (2.3) can't be invariant to this effect, because they both are scaled by R. To eliminate this effect, we can use (2.2) and (2.3) as follows:

$$\frac{M_{10}^{100}}{M_{00}^{100}} \tag{2.4}$$

Dividing (2.2) by (2.3) results in the average x coordinate for the color red. This number, however, is invariant of the scaling effect, because the scaling factor R is present in both the nominator and the denominator. This is only a very simple example, so we can't use it to compare two images in real situations. The invariants that we'll use for our image recognition algorithm are of a higher degree of difficulty.

(a) Original (b) Euclidean (c) Affine (d) Projective

Figure 2.5: *Geometric transformations*

As we told before, the different invariants, listed in [9], are grouped by their type of geometric and photometric transformations. Figure 2.5 shows some examples of geometric transformation. However, most of the geometric transformations can be simplified to affine transformations:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \tag{2.5}$$

If in any case a projective transformation can't be simplified, we'll have to normalize it before we can calculate invariants, because projective invariant moment invariants don't exist.

If we consider the photometric transformations, there are three types:

- Type D: diagonal

$$\begin{pmatrix} R' \\ G' \\ B' \end{pmatrix} = \begin{pmatrix} s_R & 0 & 0 \\ 0 & s_G & 0 \\ 0 & 0 & s_B \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{2.6}$$

- Type SO: scaling and offset

$$\begin{pmatrix} R' \\ G' \\ B' \end{pmatrix} = \begin{pmatrix} s_R & 0 & 0 \\ 0 & s_G & 0 \\ 0 & 0 & s_B \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} o_R \\ o_G \\ o_B \end{pmatrix} \tag{2.7}$$

- Type AFF: affine

$$\begin{pmatrix} R' \\ G' \\ B' \end{pmatrix} = \begin{pmatrix} a_{RR} & a_{RG} & a_{RB} \\ a_{GR} & a_{GG} & a_{GB} \\ a_{BR} & a_{BG} & a_{BB} \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} o_R \\ o_G \\ o_B \end{pmatrix} \tag{2.8}$$

The first two groups of invariants are those used in case of an affine geometric transformation. They are named GPD and GPSO. The G stands for their invariance for affine geometric transformations. The first is invariant for photometric transformations of the type diagonal (eq. (2.6)), the latter for photometric transformations of the type scaling-offset (eq. (2.7)). Diagonal means only scaling in each color band.

The second two groups can be used in case of both affine and projective geometric transformations. Note that normalization of the geometric transformation is needed, because invariant features for projective transformations can't be built. These groups are named PSO and PAFF. The first is used for scaling-offset photometric transformations (eq. (2.7)) , the latter for affine transformations (eq. (2.8)). In [9] they noticed that, in case of PSO-invariants, numerical instability can occur. That's why they redesigned these invariants and called these new invariants PSO*.

A couple of the invariants presented in [9] are listed in section 3.1.

2.2.2 Image matching

Section 2.2.1.2 clarifies how to derive features, invariant of viewpoint and illumination changes. Many different invariants are possible by combining generalized color moments in different ways. Consider we've found a type of invariant (i.e. GPD, GPSO, PSO* or PAFF) that suffices our needs. We can group some of the invariants of that type in a so-called feature vector. This feature vector can be considered as a unique identification of an image. When two images present the same content, their feature vector will be nearly identical. Mathematically this means that the distance between these vectors is very small. Based on these distances we can try to find an image-match in a large image database.

Consider two vectors in a n-dimensional space: $\begin{bmatrix} x_1 & x_2 & ... & x_n \end{bmatrix}$ and $\begin{bmatrix} y_1 & y_2 & ... & y_n \end{bmatrix}$. The distance between these vectors exists in different forms. A first distance is the Euclidean distance (eq. 2.9). This distance is the straight distance between two vectors, as shown in figure 2.6, and is based on the Pythagorean theorem.

$$d_E(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{2.9}$$

A second distance is the Manhattan distance (eq. 2.10). This distance is the sum of the distances in each dimension, as shown in figure 2.7.

$$d_M(x,y) = \sum_{i=1}^{n}|x_i - y_i| \tag{2.10}$$

Both of these distances can be used for image matching, but a problem can occur. Suppose we use two invariants to build our feature vector. The first invariant x_1 results in values in a range $[x_{1,min}, x_{1,max}]$. The second invariant x_2 results in a much wider range of values $[x_{2,min}, x_{2,max}]$. If we calculate the Euclidean distance or the Manhattan distance with one of the formulas given above, the second invariant will have much more influence on the result. Thus the image matching process is based on the second invariant only. This problem can be solved by normalizing the values of each dimension.

Figure 2.8 shows a couple of feature vectors composed of x_1 and x_2. The values of x_1 and x_1 have a standard deviation σ_1 respectively σ_2. Each vector can be normalized by dividing the value of

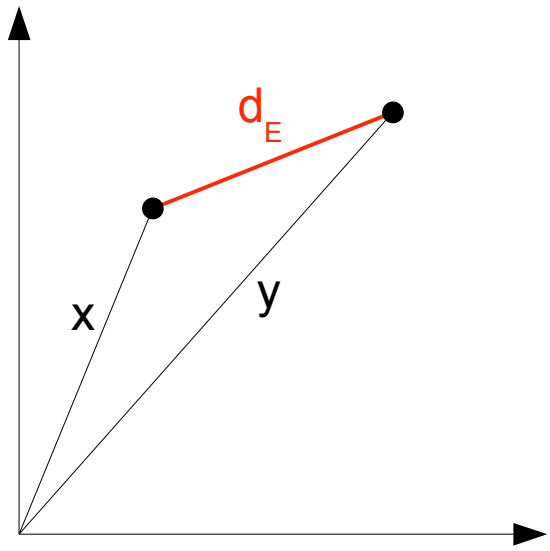


Figure 2.6: *An example of an Euclidean distance.*

each dimension x_i by their standard deviation σ_i. We can adapt the Euclidean distance and the Manhattan distance in the same way, resulting in the normalized Euclidean distance (eq. 2.11) and the normalized Manhattan distance (eq. 2.12).

$$d_{E,n}(x,y) = \sqrt{\sum_{i=1}^{n}(\frac{x_i - y_i}{\sigma_i})^2} \tag{2.11}$$

$$d_{M,n}(x,y) = \sum_{i=1}^{n}|\frac{x_i - y_i}{\sigma_i}| \tag{2.12}$$

These distances are also known as the Mahalanobis distances [13].

If we want to match an image with a large database, we'll need to calculate the distance to every image in this database. This results in a list of distances of which the smallest distance represents the best matching image. This however doesn't guarantee that this image has the same content as the original image. It is possible that the database doesn't contain such an image, so the image recognition algorithm would select the most resembling, but faulty image. We'll need to reject that result. This can be done by giving certain restrictions to the best match found in the database. There are two possibilities:

- $d_{min} < d_{th}$: A certain threshold distance d_{th} is defined and the smallest distance d_{min} has to be smaller than this threshold distance. Determining this threshold distance is not easy and therefore it's not a good solution.
- $\frac{d_{min}}{d_{min+1}} < r_{th}$: The distance of the second best match has to be r_{th} times larger than the one of the best match. This is a better solution as it gives more flexibility.

For matching images we'll have to decide on which distance we'll use and decide if we need normalization. Based on the calculated distances we can choose the best match from a database,

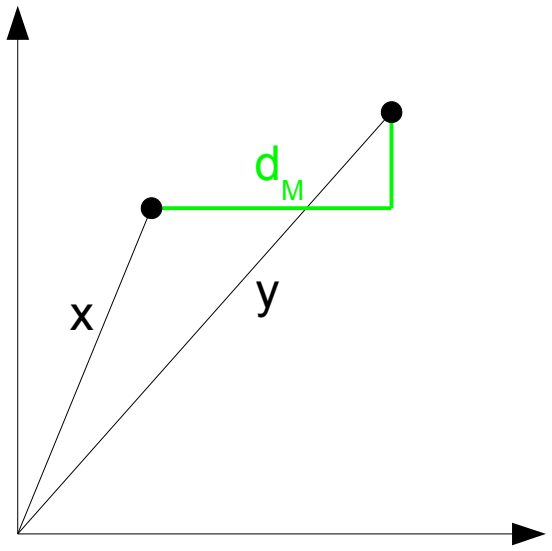


Figure 2.7: *An example of a Manhattan distance.*

but in some cases we'll need to reject that result. In chapter 3 our research and tests on this technique are explained. However, others have already done research on the same topic (image recognition on a mobile phone) using other techniques. The next section sheds some light on that work.

2.3 Case Study: Phone Guide

During our study of the topic, we came past another project with the same goal as we have. In Phone Guide [3] Föckler et al. also tried to make an interactive mobile phone based museum guide. There are two main differences between their and our project worth mentioning:

1. They used one specific telephone, where our project is focussed on an entire platform. This means our version has to be robust enough to run on multiple devices. More specifically, we have to make sure devices with different cameras, different hardware still have to be able to recognize an object.
2. Image comparison is realized with single-layer perceptron neural networks.

A short summary of their work is as following. First they extract a *feature vector* from each source image of the object. Comparing the feature vectors of a sample image with one from a library could be done using several techniques. First there is one-to-one matching, but when a library grows big this becomes exponentially slower. Also nearest-neighbor matching could be a possibility, but they chose to follow a linear separation strategy implemented with a single-layer artificial neural network. Using this approach, the feature vectors from an object get compressed into a weight vector. This weight vector is unique to the object, which facilitates matching.

In order to create the feature vectors, they calculate the color ratio in the image as well as gradients to detect edges and reveal the structure of the image. With our solution of moment invariants, we don't have to deal with edge detection. However, the use of neural networks (when properly trained) allows very high recognition rates and fast matching.

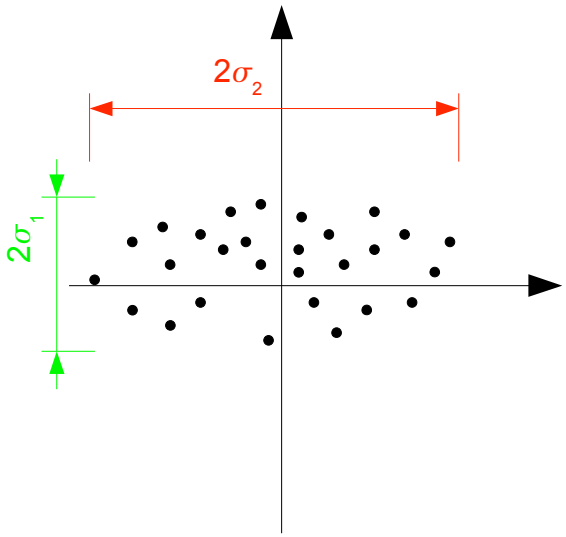


Figure 2.8: *Standard deviation of dimension values.*

2.4 Conclusion

For image recognition, many possible types, tasks and algorithms are available. One of these tasks is content-based retrieval of an image out of an image database. In our application we'll need to do this. Two large groups of algorithms are available for this: image recognition using local features and image recognition using global features. The first group isn't interesting for our thesis as it requires more computation time. Therefore we've used image recognition based on global features, more specifically *color moment invariants* proposed by Mindru et al. [9], in our thesis. Using these invariants, *feature vectors* can be created. These will be used to match an unknown image to a large database of known images. However, more research was needed to determine which invariant to use in our eventual Android application. Chapter 3 details our experiences.

As an open source OS, Android is very flexible and provides many tools for easily creating applications. Each of its possibilities have been examined and evaluated during the further development of our application. The SQLite database, for example, proved to be very useful as we needed it for storing images.

Chapter 3

Image recognition in Matlab

Chapter 1 told what our application should do. One of these tasks is calculating a vector of invariants and determining the best matching painting in the database. This involves mostly mathematical actions, which should be tested first. That's why we decided to use Matlab as a simulation environment before actually implementing the algorithms in our Android application.

In this chapter we'll explain the different steps we've executed in Matlab: first in 3.1 how we've implemented the different color moment invariant calculations, then in 3.2 how we've tested the actual invariance of each of these invariants. Section 3.3 shows how we attempted to test a first version of the matching algorithm. An important part of this algorithm is the used threshold. We also did some tests to determine its value, which is described in section 3.4. Section 3.5 summarizes our eventual results and how we interpreted them. Section 3.6 ends with a conclusion.

3.1 Implementation of invariants

Based on the paper of Mindru et al. [9] we mentioned in our literature study in section 2.2.1.2, we decided to implement some of the proposed invariants in Matlab.

We selected a couple of invariants with varying computational complexity and varying goals: invariant for geometric transformations or not, etc. For ease of use, we gave these invariants "friendly names", being GPDS1, GPDD1, GPSOS12, GPSOD1, GPSOD2, PSO1, PSO2 and PSO3. All of these invariants are specified by using "generalized color moments" as defined by formula (2.1) on page 12.

Here is a list of the invariants we tested:

Type 1 (GPD)

GPD: invariant to affine Geometric and Photometric transformations of type Diagonal

GPDS1:

$$S_{02} = \frac{M_{00}^2 M_{00}^0}{\left(M_{00}^1\right)^2} \tag{3.1}$$

GPDD1:

$$D_{02} = \frac{M_{00}^{11} M_{00}^{00}}{M_{00}^{10} M_{00}^{01}} \tag{3.2}$$

Type 2 (GPSO)

GPSO: invariant to affine Geometric and Photometric transformations of type Scaling and Offset

GPSOS12:

$$S_{12} = \frac{\left\{ \begin{array}{c} M_{10}^2 M_{01}^1 M_{00}^0 - M_{10}^2 M_{00}^1 M_{01}^0 - M_{01}^2 M_{10}^1 M_{00}^0 \\ +M_{01}^2 M_{00}^1 M_{10}^0 - M_{00}^2 M_{10}^1 M_{01}^0 - M_{00}^2 M_{01}^1 M_{10}^0 \end{array} \right\}^2}{\left(M_{00}^0\right)^2 \left[M_{00}^2 M_{00}^0 - \left(M_{00}^1\right)^2\right]^3} \tag{3.3}$$

GPSOD1:

$$D_{02} = \frac{\left[M_{00}^{11} M_{00}^{00} - M_{00}^{10} M_{00}^{01}\right]^2}{\left[M_{00}^{20} M_{00}^{00} - \left(M_{00}^{01}\right)^2\right]\left[M_{00}^{02} M_{00}^{00} - \left(M_{00}^{01}\right)^2\right]} \tag{3.4}$$

GPSOD2:

$$D_{12}^1 = \frac{\left\{ \begin{array}{c} M_{10}^{10} M_{01}^{01} M_{00}^{00} - M_{10}^{10} M_{00}^{01} M_{01}^{00} - M_{01}^{10} M_{10}^{01} M_{00}^{00} \\ +M_{01}^{10} M_{00}^{01} M_{10}^{00} + M_{00}^{10} M_{10}^{01} M_{01}^{00} - M_{00}^{10} M_{01}^{01} M_{10}^{00} \end{array} \right\}^2}{\left(M_{00}^{00}\right)^4 \left[M_{00}^{20} M_{00}^{00} - \left(M_{00}^{10}\right)^2\right]\left[M_{00}^{02} M_{00}^{00} - \left(M_{00}^{01}\right)^2\right]} \tag{3.5}$$

Type 3 (PSO)

PSO: invariant to Photometric transformations of type Scaling and Offset

PSO1:

$$S_{11} = \frac{M_{00}^0 M_{10}^1 - M_{10}^0 M_{00}^1}{M_{00}^0 M_{01}^1 - M_{01}^0 M_{00}^1} \tag{3.6}$$

PSO2:

$$S_{12}^{1} = \frac{M_{pq}^{0} M_{pq}^{2} - \left(M_{pq}^{1}\right)^{2}}{\left(M_{00}^{0} M_{pq}^{1} - M_{pq}^{0} M_{00}^{1}\right)^{2}}, pq \in \{01, 10\} \tag{3.7}$$

PSO3:

$$S_{12}^{2} = \frac{M_{00}^{0} M_{00}^{2} - M_{00}^{1} M_{00}^{1}}{(M_{00}^{0} M_{10}^{1} - M_{10}^{0} M_{00}^{1})(M_{00}^{0} M_{01}^{1} - M_{01}^{0} M_{00}^{1})} \tag{3.8}$$

The implementation of all algorithms in Matlab was facilitated by using a self-written GCM-function ("generalized color moments") which looked like:

```
function [m]=gcm(I,p,q,a,b,c)
  m=0;
  [height,width,k]=size(I);

  for x=1:width
    for y=1:height
      m=m+(x^p*y^q*I(y,x,1)^a*I(y,x,2)^b*I(y,x,3)^c);
    end
  end
```

Listing 3.1: *Original implementation of the Generalized Color Moments-function*

This approach worked, however performance wasn't great. We also noticed execution only happened on one of the cores of our computers, while our Matlab version should support multiple cores. Especially because we wanted to test the performance of algorithms on a bunch of pictures, improving performance became very important in order to reduce computation time so we decided to try and optimize the code. The cause of our low performance lay in the fact Matlab isn't really optimized for loops. We used two nested For-loops, and eliminating them would prove beneficial. Therefore we adapted our code to use matrix operations, which Matlab can execute more efficiently. After optimizing, our code was finished:

```
function [m]=gcm(I,p,q,a,b,c)
  [height,width,k]=size(I);

  Iscaled = I(:,:,1).^a .* I(:,:,2).^b .* I(:,:,3).^c;
  LinX = 1:width;
  LinY = 1:height;
  LinX = LinX .^p;
  LinY = LinY .^q;
  LinY = transpose(LinY);
  Mat = LinY * LinX;
  Ipos = Iscaled .* double(Mat);

  m = sum(sum(Ipos));
```

Listing 3.2: *Optimized implementation of the Generalized Color Moments-function*

3.2 Suitability of invariants

The previous section (3.1) gave some possible algorithms to be used for our application. Yet, we first had to find out which was the most suitable. In order to test the algorithms, three image sets specifically crafted for testing image recognition algorithms were used. The three sets were each based on one object, being either a book, another book or a bottle. The goal was only to find an algorithm which performs good at 2D-objects, so the bottle was the least relevant. However, because the bottle was always viewed from the same side, we chose to include it in our tests. The image sets we used for this can be found in the appendix.

Next, the vector with color moment invariants for each of the images were calculated, 11 images per object so 33 vectors in total. Of course we wrote a small Matlab-script to automate this and output the resulting vectors in a CSV-file.

This CSV-file was the input for another Matlab-script, which calculates the Euclidean distance (see formula 2.9 on page 14) of all possible combinations. Starting from N=33 images, we got a resulting matrix of N*N size which was then scaled. Each cell of the matrix contained a number between 0 and 1, with 0 meaning "no difference between the images" and 1 "maximal difference between images over the entire set". Soon we realized it was very hard to judge an algorithm purely by looking at the numbers. That's why we transformed this matrix into a grayscale image of 33*33 pixels. When two vectors are exactly the same, their distance is 0 or, when viewed in the image, a black pixel. A matching value of 1 resulted in a grayscale value of 255, white. This image will be called a "map" in the rest of our thesis.

After implementation of the invariants listed in section 3.1 on page 20, maps were generated for each invariant. The results can be found in section 3.5 on page 26. In an ideal situation, each of the 3 objects matches itself, and none of the other objects. A map for a perfect invariant is figure 3.1. Therefore, the suitability of an invariant can be judged by comparing its resulting map to the ideal map.

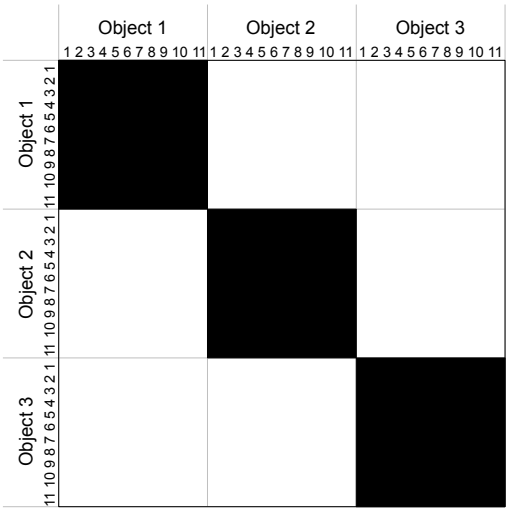


Figure 3.1: *Ideal result: a map for a perfectly working invariant formula*

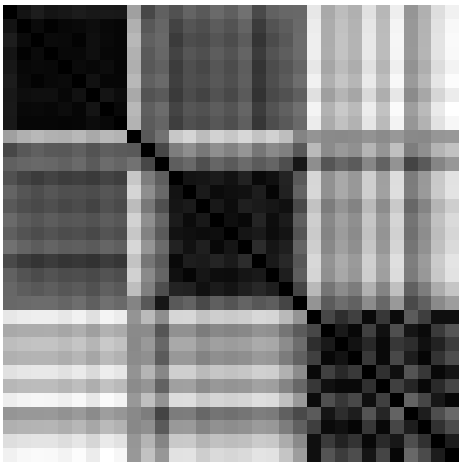


Figure 3.2: *Generated result for the GPDS1-invariant*

3.3 Matching feature vectors

The bitmaps described in the previous section, give a good general overview of the capabilities of each invariant. The next step in our Matlab research was testing if we could actually find a match for an image in a database. Such a database is filled with calculations of feature vectors of a set of test images.

As explained in section 2.2.2, the matching algorithm tries to find the best match in a database based on the distance between the feature vectors. The database object of which the feature vector has the smallest distance to the vector of the "unknown" object is the best match. However, as we've also explained, even when no real match is present in the database, we'll always find a best match. So we use a threshold value to reject these false matches. The number resulting from dividing the smallest distance by the second smallest should be smaller than the threshold.

In the Matlab file *findmatch.m*, we implemented what's explained above. The function *findmatch* expects a name of a test set, an invariant type (see section 3.1) and a threshold value. It then loads the CSV file of the databank and the CSV containing the vectors of the entire set. Then it starts searching for a best match for each vector in the test set. Each match is stored in the matrix CSV, which eventually will be the result of the *findmatch* function.

```
1  function [CSV]=findmatch(setname,invariant,thr)
2
3  DB=csvread(['./testresultaten/databank/db_',upper(invariant),'.csv']);
4  objectnr=0;
5  testfile=['./testresultaten/',setname,'/testres_',upper(invariant),'.csv'];
6  testset=csvread(testfile);
7  CSV=zeros(size(testset,1),5);
8
9  for l = 1:size(testset,1);
10    imgnr=testset(l,1);
11    if imgnr==1
12       objectnr=objectnr+1;
13    end
14
15    res=testset(l,2:4);
16    % initialize distances at 2000. This garantuees that the first computed distance
          will be smaller.
17    smallest=[2000 0];
18    sec_smallest=[2000 0];
19
20    for nr=1:size(DB,1)
21      dist=sqrt((res(1)-DB(nr,2))^2+ ...
22          (res(2)-DB(nr,3))^2+ ...  %computing euclidean distance
23          (res(3)-DB(nr,4))^2);
24
25      if dist<smallest(1)
26         sec_smallest=smallest;
27         smallest=[dist nr];
28      end
29    end
30
31    if smallest(1)/sec_smallest(1)<thr  % check if best match meets requirement
32       res=smallest(2);
33    else
34       res=0;
```

```
35      end
36
37      CSV(l,:)=[objectnr,imgnr,smallest(2),sec_smallest(2),res];
38  end
```

Listing 3.3: *findmatch: Test implementation of matching algorithm*

Notice we decided to test the Euclidean distance and not the Manhattan distance. We did generate bitmaps for both distances, but Euclidean resulted in a clearer difference between positive and negative matches. We also decided not to combine different invariant types, because we noticed that the calculation of one invariant type only already took some time in Matlab. Calculating two of these invariants would take too long to be useful in our eventual Android application. That's why we also didn't use the Mahalanobis distances, explained in section 2.2.2.

The *findmatch* function generates a long list of numbers as output. The first and the last number of each row of the list are the most interesting part for us. If these numbers are equal, we've found a positive match. If these numbers differ, we've found a false positive match. This means our algorithm decided the images match, but in reality they are a different object. The last number can also be 0, wich means that we've found a match and then rejected it because it didn't qualify with the threshold specification.

3.4 Finding the ideal threshold value

The results of the *findmatch* function, described in section 3.3, are useful for determining if we can find enough matches and also for finding the ideal threshold value for our algorithm. However, examining the output lists of *findmatch* manually would be too much work and we wouldn't get a good overview of the results. Therefore we created another function called *findmatchbatch*, which repeatedly executes the *findmatch* function while incrementing the threshold value with step size 0.01. Each result of *findmatch* is examined and *findmatchbatch* keeps track of the amount of matches, false positives and rejected matches for each threshold value. When all threshold values have been examined, it generates a graph containing the percentages of each of the aforementioned numbers.

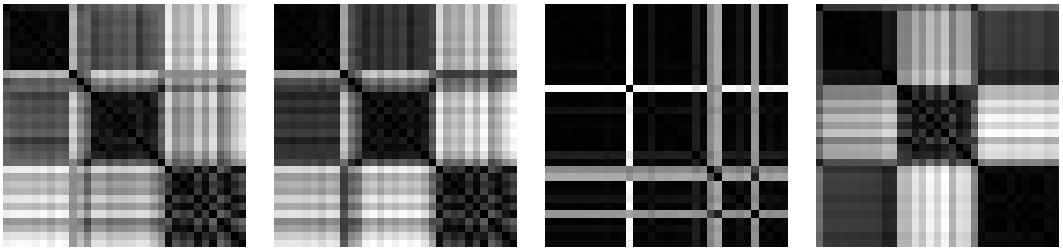
```
1  function findmatchbatch(setname,invariant)
2
3  for i=1:100;
4     findmatchres = findmatch(setname,invariant,i/100);
5     res_orig_obj = findmatchres(:,1);
6     res_found_match = findmatchres(:,5);
7     total = size(findmatchres(:,5));
8     matches(i) = sum((res_found_match-res_orig_obj)==0)/total(1);
9     iszero(i) = sum(res_found_match==0)/total(1);
10 end
11
12 misses = 1 - matches - iszero;
13 YMatrix1=[matches' misses' iszero'].*100; % matrix used for constructing the graph
14
15 % Creation of graph
16 ...
```

Listing 3.4: *findmatchbatch: Creation of graphs showing influence of threshold value on recognition rate*

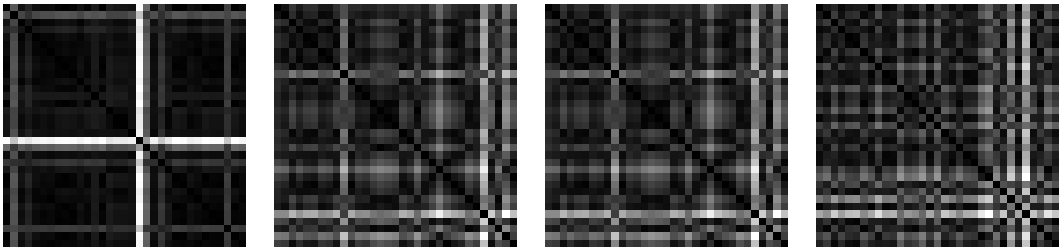
Based on the generated graph, we were able to determine the threshold value we would need in our Android application. The threshold value should deliver a high percentage of positive matches and if possible most of the false positives should be rejected. The results of this function and our interpretation of the graphs are listed in the next section.

3.5 Results

The previous two sections described methods to determine suitability of different invariants, and how to get an ideal threshold value for matching. The testing of algorithms as explained in section 3.2 generated the maps to measure the suitability of algorithms, and the method described in section 3.3 calculated the ideal threshold value.



(a) GPDS1 (b) GPDD1 (c) GPSOS12 (d) GPSOD1



(e) GPSOD2 (f) PSO1 (g) PSO2 (h) PSO3

Figure 3.3: *Maps for the algorithms listed on page 20 generated using the method described in section 3.2*

The maps in figure 3.3 were calculated (method described in section 3.2) for all of the invariants listed on page 20 for an identical set of 33 images with 11 images of each of the 3 objects. From these maps, we picked the 3 most useful ones to continue testing: GPDS1, GPDD1 and GPSOD1, because they resemble the ideal map (figure 3.1) best.

The next step in our research was generating the graphs with the *findmatchbatch* function described in section 3.4. We've tested GPDS1, GPDD1 and GPSOD1 on a set of 475 photos of real paintings, exhibited in the Pompidou museum in Paris, France. These photos can be found in the appendix. The results are shown in figures 3.4, 3.5 and 3.6. Each figure contains a green, a blue and a red graph. The green graphs represent the percentage of correct matches for each threshold value. As told in section 3.3, the correct matches are the rows of the *findmatch* output where the first and last number are equal. The red graphs is the number of rows where the first and last number differ. These are the incorrect matches which weren't detected as incorrect. This number has to be as low as possible. The sum of the remaining rows is represented by the blue graph. These results are all the correct and incorrect matches which were rejected because of the threshold requirements.

Figure 3.4 shows the graph of the matching results of the GPDS1 invariant. A first thing to notice is that a threshold of 1 or 100% allows 90% positive matches (green graph). The remaining 10% are all false positives (red graph), because a threshold of 1 can never reject a result (blue graph). When lowering the threshold value, at first the amount of faulty matches starts decreasing while the amount of positive matches stays nearly equal. This means that the lowered threshold value only effects the false positives. However when passing the limit of 65%, a significant decrease of positive matches occurs. The reason for this effect can be found in the fact that only 5% of

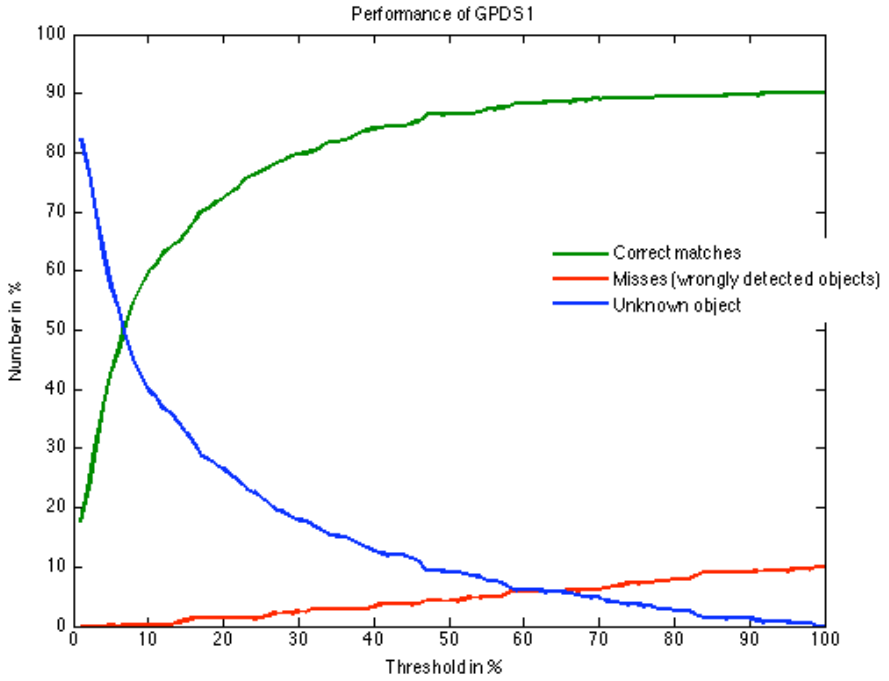


Figure 3.4: *Matching results of GPDS1 in function of the threshold*

false positives are detected. Statistically it's now more likely that a positive result is rejected. Therefore we decided that 65% would be a good threshold value for this invariant. It allows a recognition rate of about 88% and only 5% of undetected bad matches.

The next graph we generated is shown in figure 3.5. It visualizes the matching results of the GPDD1 invariant. At the threshold of 100%, the recognition rate is at about 88%, which is a little lower than in the previous graph. When lowering the threshold value, the amount of positive matches stays stable until threshold 75% is reached. When lowering this value, the same effect as in figure 3.4 can be detected. So in this case a threshold value of 75% assures the best results. With that value a recognition rate of nearly 87% and about 7% of undetected bad matches can be obtained.

The third invariant we had chosen, was GPSOD1. The results of *findmatchbatch* with this invariant are shown in figure 3.6. This graph clearly shows that, although this invariants allows a higher recognition rate, it's also more influenced by the threshold value. At threshold 100% it has about 92% positive matches and only 8% of false matches. When lowering the threshold, mostly the positive matches are rejected. We could conclude that this invariant is less stable. However, at threshold 75%, only 4% of false positives aren't detected, while recognizing 90% of the objects.

Table 3.1 summarizes the results we've collected. Based on these results, we had to choose one invariant for our Android implementation. Mathematically, the GPSOD1 invariant delivers the best results. But, keeping in mind that we'll have to implement it on a mobile phone, computationally it's two or three times heavier than the other two invariants. The small advantage of the little better recognition rate doesn't compensate its computation time. Therefore we decided not to use the GPSOD1 invariant. Of the remaining two invariants, the GPDS1 invariants has a slightly better recognition rate. It has no disadvantages against the GPDD1 invariant, so we

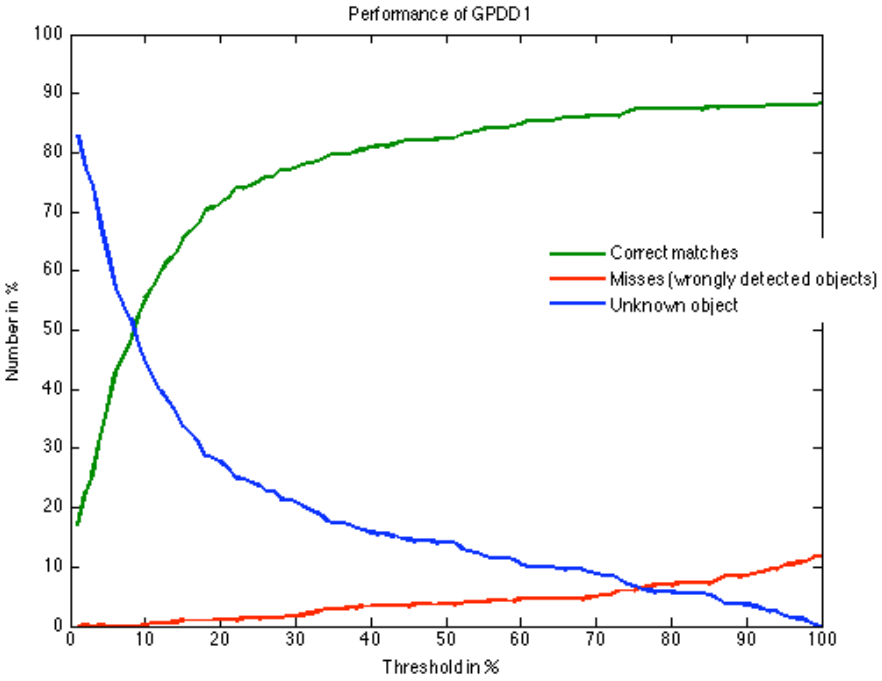


Figure 3.5: *Matching results of GPDD1 in function of the threshold*

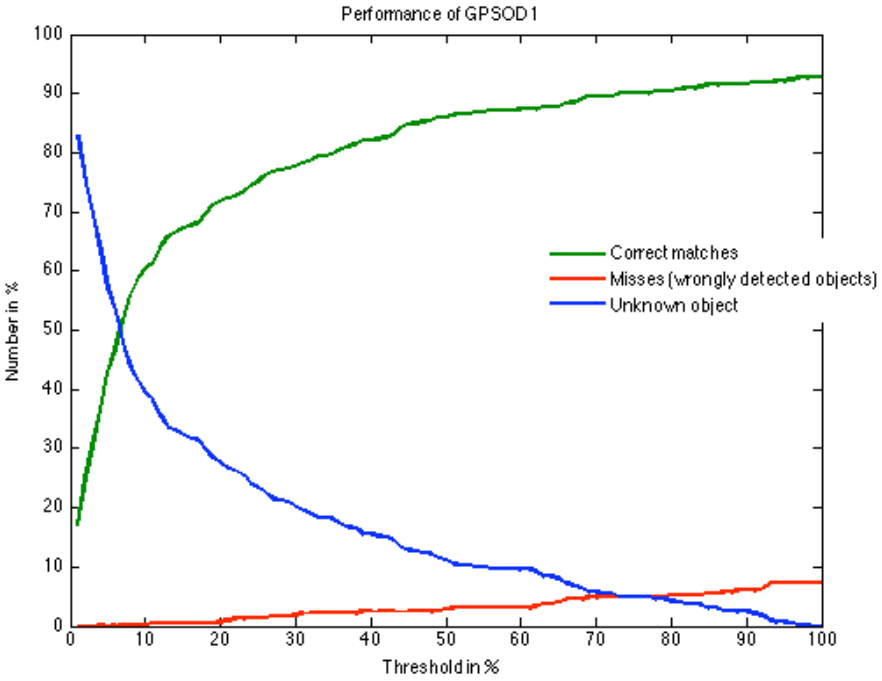


Figure 3.6: *Matching results of GPSOD1 in function of the threshold*

decided it would be the best invariant for our application with a threshold value of 65%.

Invariant	Threshold	Pos. matches	Bad matches
GPDS1	65%	88%	5%
GPDD1	75%	87%	7%
GPSOD1	75%	90%	4%

Table 3.1: *Matching results overview*

The graphs gave a good overview of the recognition rate of each invariant. Based on them we've chosen the GPDS1 invariant for our application. The graphs, however, didn't contain any information about why matches failed, which could be interesting for taking into account when developing the Android application. The failed matches consist of the incorrect matches and the rejected match results. By examining one of the CSV output files of the *findmatch* function, we noticed that mostly the pictures taken from extreme viewing angles or with a different camera orientation tended to match badly. We used this knowledge to add a solution to this problem in the eventual application. As explained in section 4.2, we allowed one painting to have several feature vectors, each for a different viewing angle or orientation. For example: one feature vector for portrait orientation and one for landscape orientation.

3.6 Conclusion

The research we've done in Matlab has proven to be very useful for the further development of our application. Although Mindru et al. [9] claimed that their formulas should be invariant to photometric and geometric changes, we've found that only the simplest invariants worked in our tests. Based on the bitmaps we've generated of our test set of optimized images, we decided that only the GPDS1, GPDD1 and GPSOD1 invariants gave proper results.

From the results of the graphs of these 3 invariants we concluded that, although they all gave similar recognition rates, the GPDS1 invariant would suit best for our application. The GPSOD1 invariant would have actually been better, but it requires too much computation time to be usable on a mobile phone. We'll use the GPDS1 invariant with threshold 0.65 to reject as many bad matches as possible. Also, pictures taken from extreme viewpoint angles or different camera orientation aren't always recognized, so we'll compensate this in our application. The next chapter details the implementation in our Android application.

Chapter 4

Implementation in Android

After finding a suitable algorithm, we started to implement it optimally in Android using Eclipse and the Android SDK (including an Android Emulator). We decided to call our application "Android Museum Guide" or AMuGu. AMuGu should be able to take pictures, calculate their feature vectors, find a match in a database and show the related information. Each of these tasks are detailed in this chapter. Section 4.1 first explains our general idea of the application. In order to find matches, we needed to build a database first. We've decided on using XML files to build our database. This to improve modifiability and user-friendlyness. However, we don't work with the XML files themselves in our application, we only import them into Android's built-in SQLite database system. Section 4.2 details the structure of our database, how we built our database from an XML file and how we store and retrieve information. How we've accomplished to execute the necessary image processing and even some optimizations, is clarified in 4.3 and 4.4. In 4.5 the matching process is explained, using the image processing and the database. Eventually, section 4.6 describes how graphical user interfaces (GUI) can be built in Android and how we've created our GUI. In section 4.7 the results we have achieved with our application are listed. Section 4.8 concludes this chapter. All source code of our application, which can be found in the appendix, is created using the official Android Dev Guide [5].

4.1 Implementation idea

A general flowchart of our Android Museum Guide (AMuGu) application is given in 4.1:

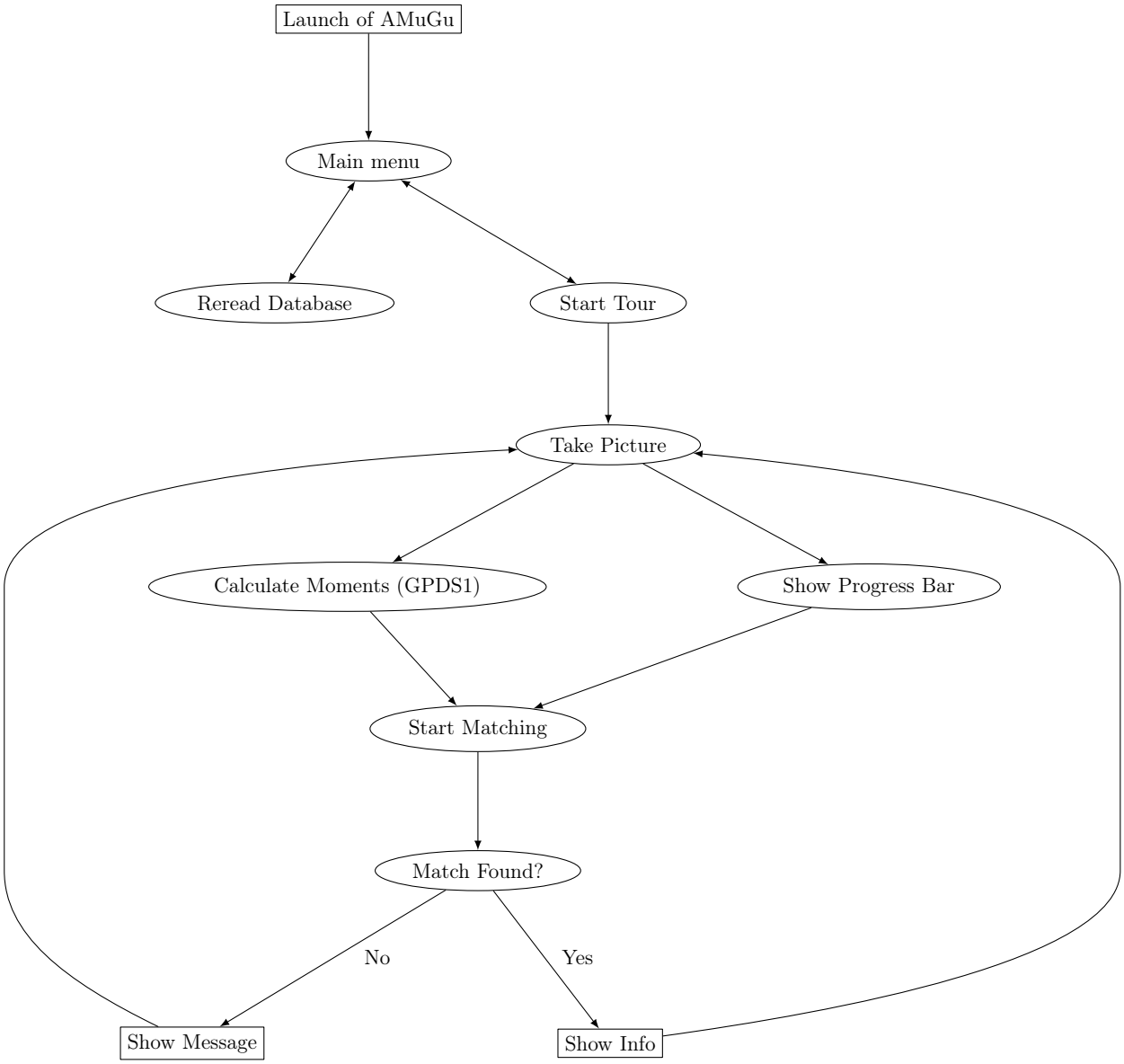


Figure 4.1: *Program flow for the Android application*

In short, when users start the application they are presented a main menu, which has 2 options: either read an XML-file into the SQLite database as explained in section 4.2 or start the museum "tour" by showing a camera preview image. When the user starts the tour, he can point the mobile phone to a painting and take a picture by using the touchscreen or a physical button on the device. Then the calculation is started and in the meanwhile the user sees a progress bar. When the calculation is complete, the database is searched for a matching known painting. If no match could be found or if it doesn't match accurately enough, the user gets a message on his screen. Otherwise, the info window is displayed with a "nice" photo of the painting. This

summary of the painting also contains the title, artist, etc. In case the museum added a link to the Info-window, an additional tab is shown for "Additional info". This is basically a web browser, meant to display an additional info page about the painting. The advantage of this approach is the flexibility, because a museum can change layout and style as they wish. They can also choose whether they want to store the web page on an internal or external webserver (intranet or internet) or if they want to store the page on the Android telephone. When the user is done reading the info or he closes the message in case no match was found, the user returns to the camera preview activity.

4.2 Database

In order to find a match for a certain picture of a painting and provide information about that painting, we needed a database. Several things were to be considered when working with a database. The first and most important step was developing a good structure for the database. After constructing the database we had to find an intuitive interface providing an easy way to load data in the database and modify it. When the data was created, we only had to make our application ask for it when needed. All of these steps will be explained in this section.

4.2.1 Database design

Our database must be able to provide all the data needed to make our application work. Therefore we need to create sufficient tables with their data fields and the tables should be linked in such a way that it can easily provide related data. The database has to hold a library of paintings. Basically this information consists of:

- the name
- the artist
- an image of the painting
- other optional info
- the vector of invariants of a photograph taken of the painting

We could have just thrown this all together in one painting table, but we decided not to. Instead we created separate *vector*, *painting* and *info* tables. We created the separate *vector* table to be able to provide more than one describing vector for a certain painting. This way we can add vectors calculated from photographs taken from more extreme angles, which, as we explained in 3.5, tend to not always be recognized. Splitting the *painting* and *info* tables wasn't really necessary, but we decided to do so in order to provide more flexibility towards possible changes in the future. Additionally, we created a *museum* table, again to provide more flexibility. For example, it is needed if we ever want to expand our application so it can contain libraries of more than one museum. Each table now had to be filled with its related data fields and an additional ID field, the so-called primary key, which uniquely describes each table entry. This resulted in the following table structures:

- painting table : pid
- vector table: inv1, inv2, inv3, vid
- info table: name, artist, img, optional, iid
- museum table: mname, mid

Note that for the *optional* and *img* fields we decided them to be pathnames of respectively an html and an image file. Putting optional info in html files allows museums to give this info a personal touch and layout. It also allows to change the info (and also the image) without having to change the database entries. In our application we'll use a WebView (see section 2.1.2) to visualize this information.

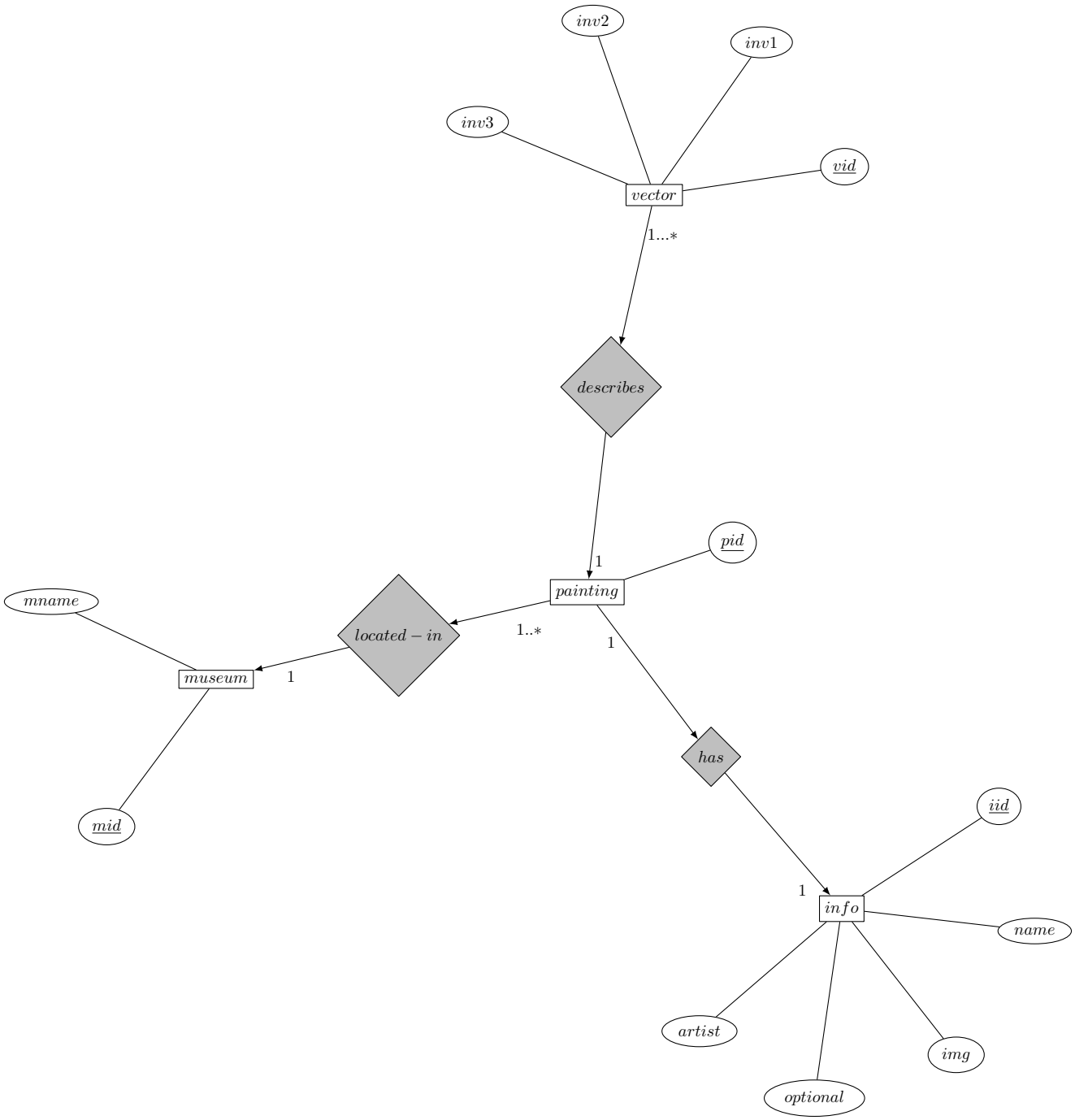


Figure 4.2: *Entity-Relation data model of the AMuGu database*

Figure 4.2 shows the Entity-Relation data model of our database. It illustrates which tables are present and which fields each table contains. But it also describes how the tables are linked: one or more vector tables describe one painting entry, a painting entry has one corresponding info entry and each painting is located in one museum. Programatically these links are achieved by adding one extra linking field to the linked tables (this is not shown in the figure). This field corresponds with the primary key of the table it is linked to. Thus, the *vector* table has an extra *pid* field and the *painting* table has extra *mid* and *iid* fields.

4.2.2 Database creation and usage in Android

In the previous section we designed a database model for storing data about paintings. Before we could use it in our application, we first had to create it. For this we used the SQLite database management system and XML files containing structured painting data. In this section we'll explain why we did this.

As we told in section 2.1.1, Android uses the SQLite database management system which can be accessed through a small C programming library. Where other database management systems are mostly standalone processes, SQLite runs as a part of the program it is called from. The program can use SQLite by making simple function calls, which reduces latency because no inter-proces communication is needed. Therefore SQLite is ideal for applications on mobile phones. In our application we didn't need to use these C function calls, because Google provided database interface classes called SQLiteDatabase and SQLiteOpenHelper. In our application we created a general database interface class called DbAdapter. This class contains an object of the SQLiteDatabase class and an extension of the SQLiteOpenHelper class, called *DatabaseHelper*. Further it has some methods for opening and closing the communication with the database, creating entries, deleting entries, retrieve data from the database and executing raw queries.

In order to use the database, we need to open it first to allow communication. Therefore a new instance of the DbAdapter class needs to be created and the method *open()* has to be executed.

```
1 public DbAdapter open() throws SQLException {
2   Log.v("DbAdapter", "Start␣Opening...");
3   mDbHelper = new DatabaseHelper(mCtx);
4   mDb = mDbHelper.getWritableDatabase();
5   Log.v("DbAdapter","Opened␣succesful!␣Version:␣" + mDb.getVersion());
6   return this;
7 }
```

Listing 4.1: *open(): Initialization of database communication*

Basically this method creates a new instance *mDbHelper* of the DatabaseHelper class and requests a SQLiteDatabase object *mDb*. The first will be used for creating, opening, updating and closing the database. The latter will be used for executing queries. The DatabaseHelper class extends the SQLiteOpenHelper class, so it also contains its methods. One of these methods is *onCreate()*, shown in listing 4.2. This method is automatically called when creating an object of this class, so we decided to override this method in order to automatically create the database if needed.

```
1 @Override
2 public void onCreate(SQLiteDatabase db) {
3   db.execSQL(DATABASE_CREATE_MUSEUM);
4   Log.v("DbAdapter","Museum␣table␣created.");
5   db.execSQL(DATABASE_CREATE_VECTOR);
6   Log.v("DbAdapter","Vector␣table␣created.");
7   db.execSQL(DATABASE_CREATE_PAINTING);
8   Log.v("DbAdapter","Painting␣table␣created.");
9   db.execSQL(DATABASE_CREATE_INFO);
10  Log.v("DbAdapter","Info␣table␣created.");
11 }
```

Listing 4.2: *Creating new database tables*

As you can see, we create each table of the database by calling *execSQL*. The argument of these methods are strings, which we also created in the DbAdapter class. For example:

```
private static final String DATABASE_CREATE_VECTOR =
  "create␣table␣" + TABLE_VECTOR
    + "␣("
    + KEY_P_VID + "␣integer␣primary␣key␣autoincrement,␣"
    + KEY_F_PID + "␣integer,␣"
    + KEY_INV1 + "␣double,␣"
    + KEY_INV2 + "␣double,␣"
    + KEY_INV3 + "␣double␣"
    + ");";
```

Listing 4.3: *Creation of vector table*

So to summarize: when the *open()* method is called on a DbAdapter object, a new instance of the DatabaseHelper class is created, resulting in calling the *onCreate()* method which creates the database if it didn't exist yet. Then by calling the *getWritableDatabase()* method on this instance, a SQLiteDatabase object will be returned, making the database accessible for executing queries.

While developing our application we decided that it should be easy for us and possible users of our application to create a lot of new entries without having to execute actual SQL queries. Another great way of bringing together structures consisting of primitive data is using XML files. That's why we decided to create our database by parsing an XML file. XML files have a pretty straightforward structure (shown in listing 4.4), making it easy for museums to create new databases or edit them. In a later phase of our thesis we even started the development of a little Java application, which acts as a GUI for creating and editing the XML file. We'll further explain this application in 5.

```
<museum name="Pompidou">
  <painting>
    <vector>
      <inv1>1.2157</inv1>
      <inv2>1.2329</inv2>
      <inv3>1.2708</inv3>
    </vector>
    <info>
      <name>L'adoration du veau</name>
      <artist>Francis Picabia</artist>
      <img>veau.jpeg</img>
      <opt>veau.html</opt>
    </info>
  </painting>
  <painting>
    <vector>
      <inv1>1.3891</inv1>
      <inv2>1.7651</inv2>
      <inv3>1.0372</inv3>
    </vector>
    <info>
      <name>La vache spectrale</name>
      <artist>Salvador Dali</artist>
      <img>vachespec.jpeg</img>
      <opt>vachespec.html</opt>
    </info>
  </painting>
</museum>
```

Listing 4.4: *An example of an XML database file*

Figure 4.3 shows the program flow for creating a database from an XML file. We noticed that reading a large database was time consuming, so we decided to run it in a background service. We start this service in the *onCreate()* method of the main activity of our application. We also provided the possibility to reread the database in case something went wrong when launching the application.

The XML parser class we used in in our application has three methods just for parsing the XML file. The first method *startElement()* is called when an opening XML tag is detected. As can be seen in listing 4.5, this method sets some booleans to determine if it passed a certain opening tag and it extracts attributes from these tags if present. The details of this function

```
1  /**
2  * Is called when an opening XML tag is found.
3  * Can also handle attributes.
4  */
5  @Override
6  public void startElement(String namespaceURI, String localName,
7    String qName, Attributes atts) throws SAXException {
8    if (localName.equals("museum")) {
9      long lm = new File("/sdcard/files/AMuGu/db.xml").lastModified();
10     museum.setMDate(lm);
11     // Extract an Attribute
12     String attrValue = atts.getValue("name");
13     museum.setMName(attrValue);
14     AMuGu.dbHelper.createEntry(museum);
15     this.in_museum = true;
16   } else if (localName.equals("painting")) {
17     this.in_painting = true;
18     Log.v("XMLHANDLER", "Adding new Painting...");
19     previous_p=p;
20     p++;
21   } else if (localName.equals("vector")) {
22     this.in_vector = true;
23     previous_p=p;
24   } else if (localName.equals("inv1")) {
25     this.in_inv1 = true;
26   } else if (localName.equals("inv2")) {
27     this.in_inv2 = true;
28   } else if (localName.equals("inv3")) {
29     this.in_inv3 = true;
30   } else if (localName.equals("info")) {
31     this.in_info = true;
32   } else if (localName.equals("name")) {
33     this.in_name = true;
34   } else if (localName.equals("artist")) {
35     this.in_artist = true;
36   } else if (localName.equals("img")) {
37     this.in_img = true;
38   } else if (localName.equals("opt")) {
39     this.in_opt = true;
40   }
41 }
```

Listing 4.5: *startElement(): determine if opening tag is passed*

The second method (listing 4.6) is called automatically when the content between two tags is

read. Based on the aforementioned booleans, we determine where this content should be stored. We decided to temporary store this content in Java objects representing table entries. This would make it easier to store it in the database afterwards.

```
/**
* Is called on <tag>content</tag> structure.
* @param ch - array of characters containing content between XML tags
* @param start - starting offset in ch
* @param length - numbers of characters to use
*/
@Override
public void characters(char ch[], int start, int length) {
  data = String.copyValueOf(ch, start, length);

  /**
  * Setting up an InvVector object.
  */
  if (this.in_vector) {
    vectors.add(new InvVector());
    this.in_vector=false;
  }
  if (this.in_inv1) {
    vector=(InvVector) vectors.get(vectors.size()-1);
    vector.setInv1(Float.parseFloat(data));
  }
  if (this.in_inv2) {
    vector=(InvVector) vectors.get(vectors.size()-1);
    vector.setInv2(Float.parseFloat(data));
  }
  if (this.in_inv3) {
    vector=(InvVector) vectors.get(vectors.size()-1);
    vector.setInv3(Float.parseFloat(data));
  }

  /**
  * Setting up an Info object
  */
  if (this.in_artist) {
    info.setArtist(data);
  }
  if (this.in_name) {
    info.setName(data);
  }
  if (this.in_img) {
    info.setImg("/sdcard/files/AMuGu/info/" + data );
  }
  if (this.in_opt) {
    info.setOptInfo("/sdcard/files/AMuGu/info/" + data);
  }
}
```

Listing 4.6: *Extracting data from between XML tags*

When passing a closing tag we should set the booleans on false again and, as show in figure 4.3, when passing the closing painting tag we should store the painting in the database. This last part is done by calling the method *commitPainting()*.

```
1 /**
2 * Is called to create the database entry with the data extracted from the XML file.
3 */
4 private void commitPainting() {
5   AMuGu.dbHelper.createEntry(info);
6   Cursor c = AMuGu.dbHelper.fetchInfo();
7   c.moveToLast();
8   int iid = c.getInt(c.getColumnIndex(DbAdapter.KEY_P_IID));
9   painting.setIid(iid);
10  c.close();
11  c = AMuGu.dbHelper.fetchMuseum();
12  c.moveToLast();
13  int mid = c.getInt(c.getColumnIndex(DbAdapter.KEY_P_MID));
14  painting.setMid(mid);
15  c.close();
16  AMuGu.dbHelper.createEntry(painting);
17
18  c = AMuGu.dbHelper.fetchPaintings();
19  c.moveToLast();
20  int pid = c.getInt(c.getColumnIndex(DbAdapter.KEY_P_PID));
21  Log.v("XMLHANDLER",info.toString());
22  Log.v("XMLHANDLER",painting.toString());
23  Iterator<InvVector> i = vectors.listIterator();
24  while(i.hasNext()) {
25    InvVector v = i.next();
26    v.setPid(pid);
27    AMuGu.dbHelper.createEntry(v);
28    Log.v("XMLHANDLER",v.toString());
29  }
30  c.close();
31  vectors.clear();
32  Log.v("XMLHANDLER", "Painting␣added!");
33 }
```

Listing 4.7: *commitPainting(): Store painting extracted from the XML file in database*

Each step of the *commitPainting()* method is listed in listing 4.7. The method first creates an entry in the *info* table in the database. This entry will automatically get a new *iid* value from the SQLite database management system. By calling the *fetchInfo()* method, we get a cursor over all *info* table entries. The last created entry can be reached by calling the *moveToLast()* method on the cursor object. We then can get the *iid* field from this entry to use in the *painting* entry that will be created next. The same thing is done for retrieving the *mid* of the last *museum* entry. When the new *painting* entry is created we execute the same steps for creating the *vector* entries.

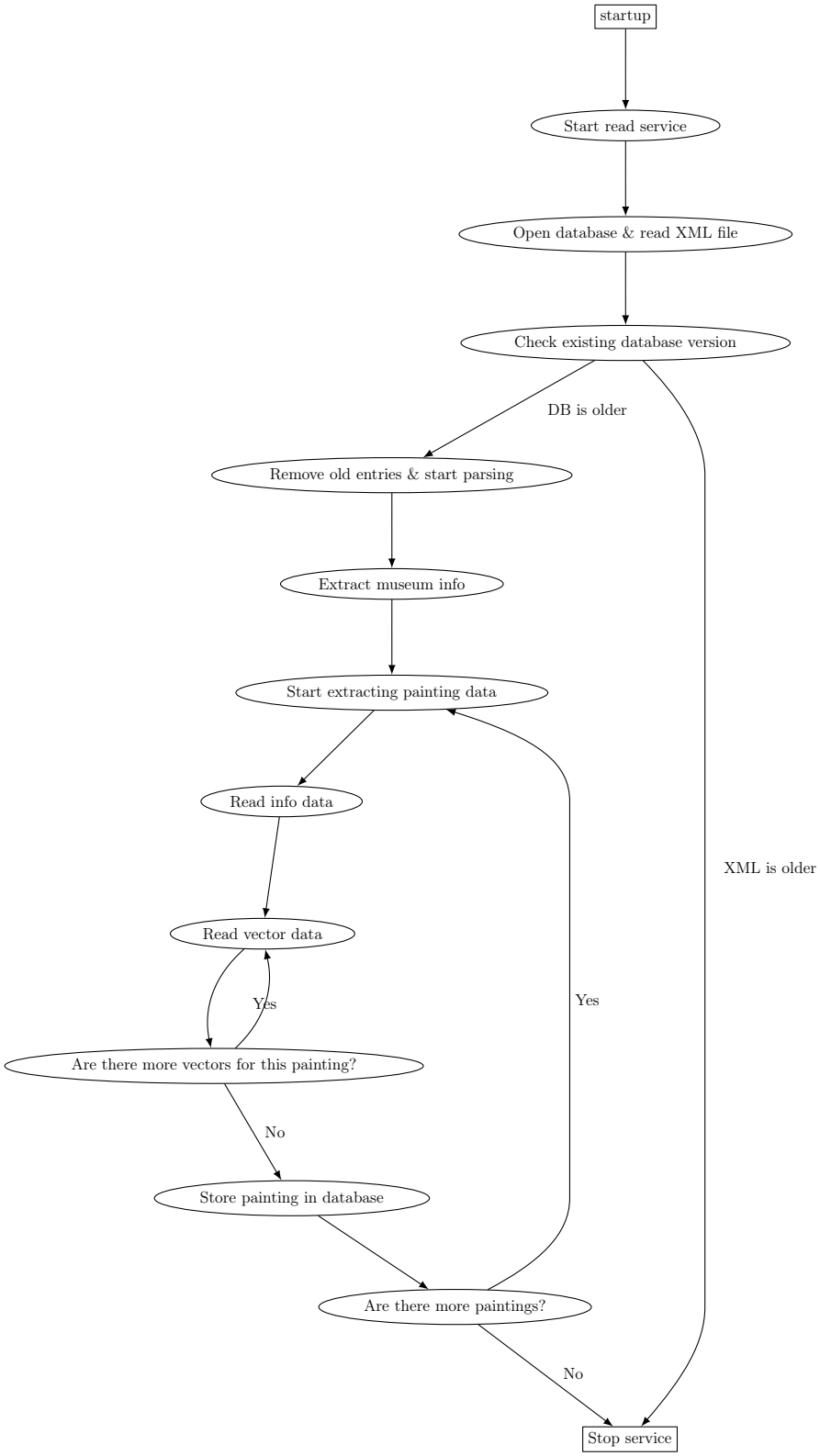


Figure 4.3: *Reading database during startup of the application*

4.3 Capturing images

After the user selected the "Start tour" option in the main menu, he starts the *Tour*-activity which presents a live preview of the camera image. We based our code on Google's CameraPreview.java sample code [4]. Of course we had to expand the code, because we don't only want a preview, we want to take a picture by clicking a "Take picture" button. To do this, we had to make an *OnClickListener* which calls the *takePicture* procedure of our camera when the button is clicked. As arguments to the *takePicture*-call, we pass a newly created *jpegCallback*. Android has 2 possible callbacks for pictures: JPEG and RAW. However, RAW is not supported yet, so we have to use the JPEG-option. This callback function first decodes the JPG-data into a *Bitmap*-object, and passes the bitmap to the *GPDS1*-object for calculation. The calculation isn't started yet. A *Handler* is created so the *Tour* object can get notified when the calculation is done, and then it's time to start executing our image recognition code using the *GPDS1*-object.

```
1  public class Tour extends Activity {
2    // Create our Preview view and set it as the content of our activity.
3    mPreview = new Preview(this);
4    mLL = new LinearLayout(this);
5    mLL.setOrientation(LinearLayout.VERTICAL);
6
7    bGPDS1 = new Button(this);
8    bGPDS1.setText("Take picture");
9    bGPDS1.setVisibility(View.VISIBLE);
10   bGPDS1.setClickable(true);
11   bGPDS1.setOnClickListener(mGPDS1Listener);
12
13   mLL.addView(bGPDS1);
14   mLL.addView(mPreview);
15
16   setContentView(mLL);
17 }
18
19 @Override
20 protected void onResume() {
21   super.onResume();
22   bGPDS1.setClickable(true);
23 }
24
25 OnClickListener mGPDS1Listener = new OnClickListener() {
26   public void onClick(View v) {
27     bGPDS1.setClickable(false);
28     mPreview.mCamera.takePicture(shutterCallback, null, jpegCallback);
29   }
30 };
31
32 Camera.PictureCallback jpegCallback = new Camera.PictureCallback() {
33   public void onPictureTaken(byte[] _data, Camera _camera) {
34     Bitmap bmpTmp = BitmapFactory.decodeByteArray(_data, 0, _data.length);
35     oGPDS1 = new GPDS1(TourActivity,bmpTmp);
36     Handler wait4calc = new Handler() {
37       public void handleMessage(Message msg) {
38         Result = oGPDS1.getResult();
39         AMuGu.getPhoto().setInv(Result[0],Result[1],Result[2]);
40         Intent i = new Intent(TourActivity,ShowInfo.class);
41         i.putExtra("KEY_PID", AMuGu.getPhoto().findMatch());
```

```
42          startActivity(i);
43        }
44      };
45      oGPDS1.execute(wait4calc);
46    }
47 };
```

Listing 4.8: *Capturing images*

The *GPDS1*-object starts by storing the *Handler* for the *Tour* object in a class variable. Then it creates a *ProgressDialog*, configures and starts it. A seperate *Thread* is created to do the calculation. If we didn't start a new thread, it was impossible to have user interaction and calculation at the same time. This would mean the calculation would take up 100% of the processor, and the user wouldn't see any progress on his display, the device would look frozen. By separating the calculation from the user interface using threads, the user sees an updating progress bar. The new calculation thread is actually the *GPDS1*-object itself being executed. This is possible because we made it implement *Runnable*. So, when the thread is started, the *run*-function of *GPDS1* is called which does 2 things: first it starts the calculation itself (explained in more detail in section 4.4) and calling an internal *Handler* which dismisses the *ProgressDialog*. When the calculation is busy, the *ProgressDialog* gets updated by the *GPDS1*-thread. When the *calculatebatch*-function returns, the resultvector is known and the *ProgressDialog* is dismissed.

```
1  public class GPDS1 extends Object implements Runnable{
2    /* The GPDS1-class will generate the invariant color moment of a given image by
3     * using the GPDS1-algorithm.
4     */
5    private Bitmap bmpImage = null;
6    private Float[] Result = new Float[3];
7    private int pdupdatenrs = 25 ; //updates of progressbar per image
8    private ProgressDialog pd;
9    private Context cntParent;
10   private Handler ParentHandler;
11   private Thread GPDS1thread;
12
13   public GPDS1(Context _cnt, Bitmap _bmpImage) {
14     super();
15     bmpImage = _bmpImage;
16     cntParent = _cnt;
17   }
18
19   public Float[] getResult() {
20     return Result;
21   }
22
23   public void execute(Handler _refHandler) {
24     ParentHandler = _refHandler;
25     pd = new ProgressDialog(cntParent);
26     pd.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
27     pd.setTitle("Working...");
28     pd.setMessage("Processing image");
29     pd.setMax(pdupdatenrs);
30     pd.show();
31     GPDS1thread = Thread.currentThread();
32     Thread thread = new Thread(this);
33     thread.start();
34   }
```

```
35
36    private Float[] calculatebatch() {
37      ...
38      return Result;
39    }
40
41    public void run() {
42      Result = calculatebatch();
43      handler.sendEmptyMessage(0);
44    }
45
46    private Handler handler = new Handler() {
47      public void handleMessage(Message msg) {
48        pd.dismiss();
49        ParentHandler.sendEmptyMessage(0);
50      }
51    };
52  }
```

Listing 4.9: *Implementation of GPDS1 object illustrating the usage of threads*

Now we know the vector of the image we can notify the *Handler* of our *Tour* activity. From there the matching is started, as further explained in section 4.5.

The full code for capturing images can be found in the appendix in files tour.java and gpds1.java.

4.4 Optimization of code

After our Matlab-experiments, we went on to transform the algorithms to Java-code. In order to generate our test-results of 475 photos in Matlab at a reasonable speed, we had to optimize the Matlab-code as much as possible. This was mainly accomplished by trying to avoid for-loops and replacing them by matrix-operations. By doing this, our code became more efficient, however, not as easily portable to Java. In other words, we started from the optimized Matlab-code and basically had to undo most of the optimizations we created. This extra step wasn't completely useless, by doing this we had a better view on exactly what happened in the code.

Original version

Our first Java-implementation of the algorithm we called "GPDS1" looked something like this:

```
1  private Double[] calculate() {
2    Double[] dblResult = new Double[]{0.0d, 0.0d, 0.0d};
3    Double dblTmp;
4    int i;
5    for (i=0; i<3; i++) {
6      dblTmp = gcms(0,0,1,i);
7      dblResult[i] = gcms(0,0,2,i)*gcms(0,0,0,i)/(dblTmp*dblTmp);
8    }
9    return dblResult;
10 }
11 private Double gcms(int p, int q, int scale, int rgb) {
12   Double m = new Double(0.0d);
13   int iPixel;
14   double i,j;
15   Integer[] iRGB = new Integer[]{0,0,0};
16   double[] dRGB = new double[]{0,0,0};
17   int height = bmpImage.getHeight();
18   int width = bmpImage.getWidth();
19
20   for (i=0; i<height; i++) {
21     for (j=0; j<width; j++) {
22       iPixel = bmpImage.getPixel((int)j,(int)i);
23       iRGB[0] = Color.red(iPixel);
24       iRGB[1] = Color.green(iPixel);
25       iRGB[2] = Color.blue(iPixel);
26       dRGB[0] = iRGB[0].doubleValue()/255;
27       dRGB[1] = iRGB[1].doubleValue()/255;
28       dRGB[2] = iRGB[2].doubleValue()/255;
29       m += ( Math.pow(j,p)
30            * Math.pow(i,q)
31            * Math.pow(dRGB[rgb],scale)
32          );
33     }
34   }
35   return m;
36 }
```

Listing 4.10: *Original GPDS1-implementation for Android*

Running this code in the Android-emulator took about 1m43s for a single photo on the computer we used for testing. Because we didn't have any Android-hardware available, these simulated timing results offer a realistic measure. However, we did realise that the mobile phone would basically be slower compared to the PC because it has less MIPS, but on the other hand, the Android-phones can accelerate the Java-bytecode in their processors. The general approach we used was: if we can speed it up, do it.

First optimization

When looked more closely at the code in (Listing 4.10), one might notice *gcms* is called 9 times. Each *gcms* scans the entire image. In order to optimize this we reduced the number of full-image scans to 1. This was done by creating a *gcmsbatch* function, which accepts an array of GCMS options as parameter.By doing this, multiple operations can be executed on a pixel while only scanning the image once. Of course, the *calculate* function has to create the array of GCMS commands. We also cancelled out the *Math.pow*-function because we only used exponents 0, 1 and 2. Exponent 0 means the result is always 1, exponent 1 means the result is the original value, and exponent 2 means value*value. Calculating the square of a value is a lot faster using a simple multiplication than using the *Math.pow*-function, as described in [7]. The *calcPow* function was our answer to this problem. Furthermore, the datatype for our calculations was done in double-precision, which probably isn't needed. Floats are faster computable. After adjustments our code looked like this:

```
1  private Float[] calculatebatch() {
2    Float[] floResult = new Float[]{0.0f, 0.0f, 0.0f};
3    Float floTmp;
4    int i,offset;
5
6    GPDS1.GCMScommand[] gcmsc = new GPDS1.GCMScommand[9];
7    //insert commands for batch processing
8    for (i=0; i<3; i++) {
9      offset = i*3; //offset
10     gcmsc[offset+0] = new GPDS1.GCMScommand(0,i);
11     gcmsc[offset+1] = new GPDS1.GCMScommand(1,i);
12     gcmsc[offset+2] = new GPDS1.GCMScommand(2,i);
13   }
14
15   //execute commands
16   gcmsc = gcmsbatch(gcmsc);
17
18   //post-processing: combining individual gcms-results into RGB-results
19   for (i=0; i<3; i++) {
20     offset = i*3; //offset
21     floTmp = gcmsc[offset+1].getResult();
22     floResult[i] = gcmsc[offset+2].getResult()*gcmsc[offset+0].getResult()/(floTmp*
           floTmp);
23   }
24   return floResult;
25 }
26
27 private GPDS1.GCMScommand[] gcmsbatch(GPDS1.GCMScommand[] gcmsc) {
28   int iPixel;
29   int i,j,k;
30   Integer[] iRGB = new Integer[]{0,0,0};
```

```
31    Float[] fRGB = new Float[]{0.0f,0.0f,0.0f};
32    int height = bmpImage.getHeight();
33    int width = bmpImage.getWidth();
34    int numpixels = height*width;
35    int pdupdate = numpixels/pdupdatenrs;
36
37    for (i=0; i<height; i++) {
38      for (j=0; j<width; j++) {
39        iPixel = bmpImage.getPixel(i,j);
40        iRGB[0] = Color.red(iPixel);
41        iRGB[1] = Color.green(iPixel);
42        iRGB[2] = Color.blue(iPixel);
43        fRGB[0] = iRGB[0]/255;
44        fRGB[1] = iRGB[1]/255;
45        fRGB[2] = iRGB[2]/255;
46
47        for (k=0; k<gcmsc.length; k++) {
48          gcmsc[k].addToResult(calcPow(fRGB[gcmsc[k].rgb],gcmsc[k].scale));
49        }
50    }
51    return gcmsc;
52  }
53
54  private class GCMScommand extends Object {
55    //commands
56    int p   = 0;
57    int q   = 0;
58    int scale  = 0;
59    int rgb  = 0;
60    //result
61    float m  = 0.0f;
62
63    GCMScommand(int _scale, int _rgb) {
64      scale = _scale;
65      rgb = _rgb;
66    }
67
68    GCMScommand(int _p, int _q, int _scale, int _rgb) {
69      p = _p;
70      q = _q;
71      scale = _scale;
72      rgb = _rgb;
73    }
74
75    public void addToResult(float d) {
76      m += d;
77    }
78
79    public float getResult() {
80      return m;
81    }
82  }
83
84  private float calcPow(float a, int b) {
85    if (b==1) {
86      return a;
```

```
87   } else if (b==2) {
88     return a*a;
89   } else {
90     return 1.0f;
91   }
92 }
```

Listing 4.11: *First optimization: using gcmsbatch*

Above code managed to improve the time from 1m43s to 0m31s. Although this means a possible speed-up with a factor 3.3, we still weren't completely satisfied. On one hand we wanted to make it faster and on the other hand add extra functionality: a progress bar so the user knows something is happening. The Android-platform already contains a *Progressbar* object so this shouldn't be too hard.

Second optimization

In order to improve speed, we looked at the case where *calcPow* returns a*a. It's not incredibly slow because we don't use the *Math.pow* function, but the value of "a" is likely to occur more than once in the image. We thought we could improve by making a kind of primitive cache. An extra advantage of this approach was eliminating divisions. In (Listing 4.11, lines 43-45) the integer values for R, G and B were divided by 255 for each pixel. Now, the integer value is passed to the cache-function, and used as an index of the storage array. Also, "*GPDS1.GCMScommand(0,i)*" ran 3 times for different values of *i*, but the result was always the same. So, it's a good idea to only request 7 instead of 9 *GCMScommands*.

```
1  GPDS1.GCMScommand[] gcmsc = new GPDS1.GCMScommand[7];
2  //insert commands for batch processing
3  for (i=0; i<3; i++) {
4    offset = i*2; //offset
5    gcmsc[offset+0] = new GPDS1.GCMScommand(1,i);
6    gcmsc[offset+1] = new GPDS1.GCMScommand(2,i);
7  }
8  //performance: GCMScommand(0,0,0,i) is equal for every i,
9  //so execute it only once and put it as the last element of the array
10
11 gcmsc[gcmsc.length-1] = new GPDS1.GCMScommand(0,0);
12 //execute commands
13 gcmsc = gcmsbatch(gcmsc);
14
15 //post-processing: combining individual gcms-results into RGB-results
16 for (i=0; i<3; i++) {
17   offset = i*2; //offset
18   dblTmp = gcmsc[offset].getResult();
19   dblResult[i] = gcmsc[offset+1].getResult()*gcmsc[gcmsc.length-1].getResult()/(dblTmp
         *dblTmp);
20 }
```

Listing 4.12: *Second optimization: 7 GCMScommands instead of 9*

As for the progressbar we ran into some trouble. Even though the progressbar objects were supplied in the Android library, using them was a bit difficult. The idea of a progressbar is that you see progress while a lenghty operation is executing. The calculation of our invariant moments had to be split from the graphical user interface by using multiple threads. We had

some difficulties getting the threads to run while keeping the screen updated. The progressbar itself also introduced some extra computation time. First we tried to update the bar once every horizontal line in the image, but this proved quite inefficient. Then we introduced a variable to specify the number of updates in general for an image. A value of 25 proved to be a good compromise between speed and responsiveness.

With these optimizations, the processing time was reduced to 19 seconds.

Third optimization

The cache introduced in the previous optimization was re-written. Each time a value was needed, an *if* structure checked if the cache knew the value. This was a waste of time, and a lookup-table (LUT) was a better solution. Even more, we decided to fill the LUT at the start of the calculation using OpenGL|ES-instructions. In the Android library there are a couple functions to calculate the product of two 4x4 *Float* matrices, among others. Using this, we could use diagonal matrices and calculate 4 values in 1 instruction. Because a value was in the range of 0-255, we had to execute this 64 times. Instead of really caching results, we now calculate all possible squared values at initialisation. We risk calculating some values we don't need, but it's an acceptable risk. We don't have to check each time if the calculated value already exists or if it needs to be computed.

Instead of executing the *getPixel* function for each pixel in *gcmsbatch*, an array of integers is built which gets all values for the bitmap.

```
1 int[] iPixels = new int[width*height];
2 bmpImage.getPixels(iPixels, 0, width, 0, 0, width, height);
```

Listing 4.13: *Third optimization: getPixels*

Some *for*-loops could also be written in an optimized form. Using "`for (int item : iPixels){`" is more efficient than "`for (i=0; i<height; i++){ for (j=0; j<width; j++){`". This construction is called the "enhanced for loop" [12]. The code now looks like:

```
1 for (int item : iPixels) {
2     iRGB[0] = Color.red(item);
3     iRGB[1] = Color.green(item);
4     iRGB[2] = Color.blue(item);
5
6     for (k=0; k<gcmsc.length; k++) {
7       gcmsc[k].addToResult(calcPow(iRGB[gcmsc[k].rgb],gcmsc[k].scale));
8     }
9 }
```

Listing 4.14: *Third optimization: the enhanced for loop*

In the end, one image now gets processed in 8 seconds, more than 12 times faster than our original implementation. A graphical comparison of execution times is shown in table 4.1. The fully optimized code can be found in the appendix.

Optimization method	Time [s]	
None	103	
First optimization	31	
Second optimization	19	
Third optimization	8	
		0 30 60 90 120

Table 4.1: *Comparison of execution times*

4.5 Matching

In the previous section, we explained how the invariants of a new picture are computed. The next step is matching these invariants to the database of pictures. Figure 4.4 shows the different steps of the matching process. As we've told in section 4.3, when a picture is taken, its results are stored in a *MuseumPhoto* object. Based on these values, we try to find a match for this *MuseumPhoto* object. This is done by the *findMatch()* method, which executes most of the steps of figure 4.4.

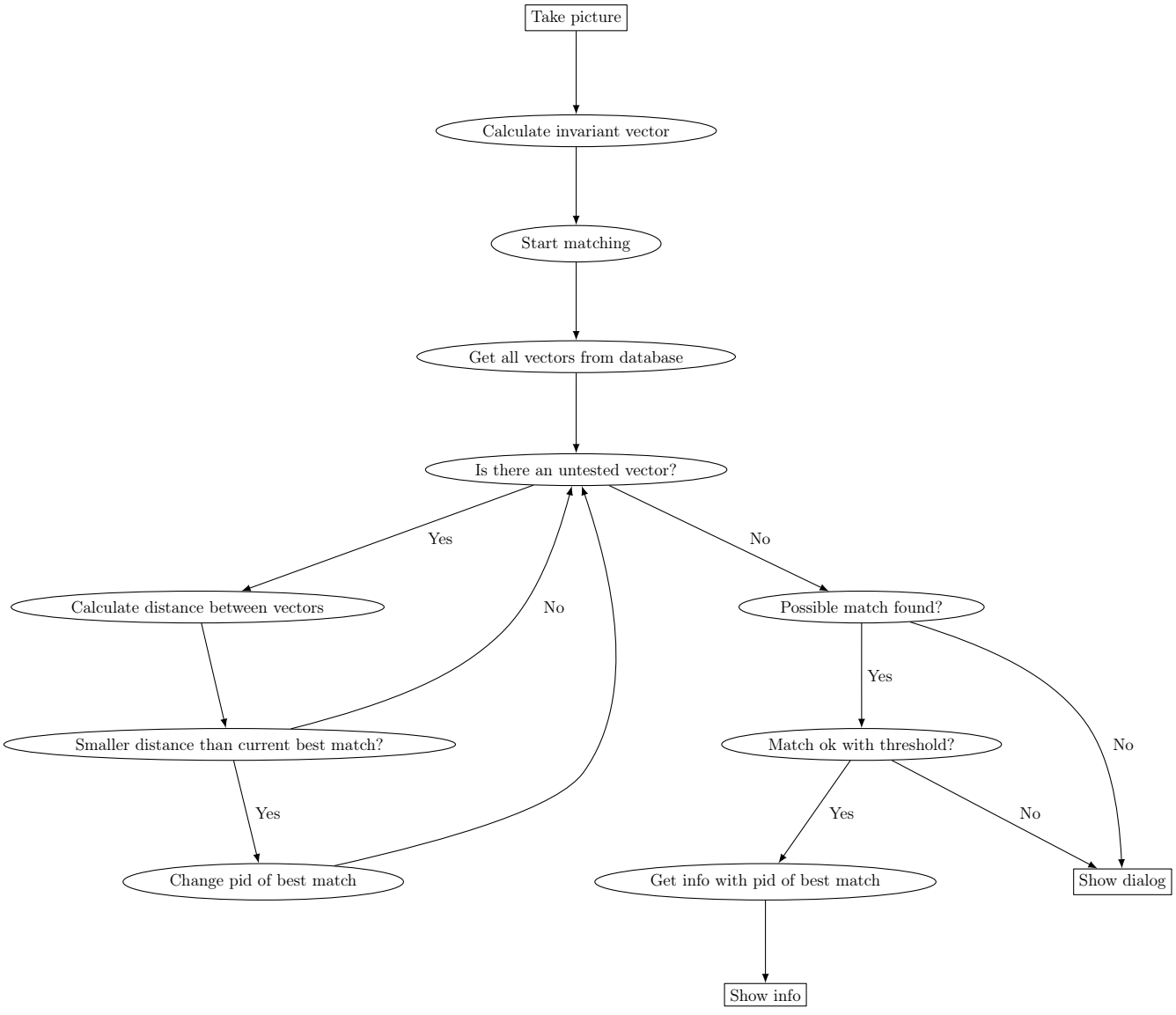


Figure 4.4: *Matching process*

First the vectors in the database are loaded. This is done the same way as loading the *info* table entries in the *commitPainting()* method used in section 4.2. Each vector's distance to the vector of the taken picture is calculated. If the calculated distance is smaller than the distance of the previous best match, we've found a better one and therefore update the *pid* value of the best

match. All this is done by the code in listing 4.15.

```
do {
  // calculation
  m_inv1 = c.getFloat(c.getColumnIndex(DbAdapter.KEY_INV1));
  m_inv2 = c.getFloat(c.getColumnIndex(DbAdapter.KEY_INV2));
  m_inv3 = c.getFloat(c.getColumnIndex(DbAdapter.KEY_INV3));
  m_pid = c.getInt(c.getColumnIndex(DbAdapter.KEY_F_PID));

  dist = (float) Math.sqrt((inv1-m_inv1)*(inv1-m_inv1)
       + (inv2-m_inv2)*(inv2-m_inv2) + (inv3-m_inv3)*(inv3-m_inv3));
  if (dist<smallest_dist) {
    Log.v("MUSEUMPHOTO", "Best␣match␣changed");
    smallest = m_pid;
    s_smallest_dist=smallest_dist;
    smallest_dist=dist;
  }
} while (c.moveToNext());
```

Listing 4.15: *Calculate distance to all database vectors*

Notice that we also keep track of the smallest and second smallest distance we've found. There's a good reason for that. In section 2.2.2 we explained that if there's an image in the database we'll always find a smallest distance and thus a best match. This however doesn't mean that it's a correct match. So we decided to use a threshold value to reduce the number of mismatches. In section 3.5 we've explained that 0.65 is a good threshold value for the GPDS1 invariant. So before returning the *pid* of the best match, we first test if it matches the threshold specifications. This is shown in listing 4.16.

```
if (smallest==0) { //No vectors were loaded from the database
  Log.v("MUSEUMPHOTO","No␣Match␣Found");
  return -1;
} else if (smallest_dist/s_smallest_dist < THR) { // if best match is ok (qualifies
    with the threshold)
  Log.v("MUSEUMPHOTO","Match␣Found");
  return smallest;
} else {
  Log.v("MUSEUMPHOTO","Bad␣Match");
  return 0;
}
```

Listing 4.16: *Check threshold requirements*

For showing the info related to the matching painting, we've created a new Java class *ShowInfo* which extends an Android Activity class. We launch this activity when the matching process is completed. In *ShowInfo* we load the related info from the database in an *Info* object using its *setInfoFromPID()* method.

```
public void setInfoFromPID(int _pid) {
  Cursor c;
  AMuGu.dbHelper.open();
  c = AMuGu.dbHelper.executeSql("select␣*␣from␣info␣" +
                  "inner␣join␣painting␣on␣info._iid=painting.iid␣" +
                  "where␣painting._pid="+_pid+";");
  if (!c.moveToFirst()) return;
  iid  = c.getInt(c.getColumnIndex(DbAdapter.KEY_P_IID));  //retrieve index of the iid
      -column
```

```
9   name = c.getString(c.getColumnIndex(DbAdapter.KEY_NAME)); //retrieve index of the
        name-column
10  artist = c.getString(c.getColumnIndex(DbAdapter.KEY_ARTIST)); //retrieve index of
        the artist-column
11  img  = c.getString(c.getColumnIndex(DbAdapter.KEY_IMG));  //retrieve index of the
        img-column
12  optinfo = c.getString(c.getColumnIndex(DbAdapter.KEY_OPT));  //retrieve index of the
         opt-column
13  AMuGu.dbHelper.close();
14 }
```

Listing 4.17: *setInfoFromPID(): Initialize Info object to be shown in ShowInfo activity*

As you can see in listing 4.17, we don't use one of the predefined SQLite queries. We join the *info* table and the *painting* table. Since only the rows where the *iid* of the *info* table equals the *iid* of the *painting* table are relevant, we join them based on this equality. Out of the resulting rows we select the one where the *pid* is the same as the one we found in the matching process. Now we've filled all the data fields of the *Info* object, we can use it to construct our GUI. We'll describe how this is done in section 4.6.

4.6 Graphical user interface

As most other applications, our application needed a graphical user interface or GUI to allow user interaction. This section first explain how the creation of GUIs in Android works. Then it describes how we created our GUIs and we'll show the result.

4.6.1 General idea

Basically one can design and control his GUI with only the Java programming language. However, it's common knowledge that it's better to split design and development. Therefore Google provided the possibility to design your GUI in XML files. These XML files provide more possibilities than Java created GUIs. It is possible to create a layout for each activity in an application by using different XML files. In these files you can define elements like buttons, TextViews, WebViews, ... using XML tags. Attributes in these tags are used for further configuration of these elements. One of the configuration possibilities is defining an ID.

The Android Developer Tools automatically generate a file called *R.java*, which contains addresses to all the resources (for example layouts, layout elements, strings and drawables) defined in the file. It acts as an ID-address mapping table. This allows to address and control layout elements in Java-written Activity classes through their IDs.

```
1  public final class R {
2    public static final class attr {
3    }
4    public static final class drawable {
5      public static final int icon=0x7f020000;
6    }
7    public static final class layout {
8      public static final int main=0x7f030000;
9    }
10   public static final class string {
11     public static final int app_name=0x7f040001;
12     public static final int hello=0x7f040000;
13   }
14 }
```

Listing 4.18: *Example of an R.java file*

4.6.2 Our GUI

Main menu

Figure 4.5: *Main menu*

Our main menu is a pretty basic layout. All parts of the layout are included in the main.xml file (listing 4.19). It has one TextView and two buttons. We ordened these elements in a TableLayout.

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
  <TableRow android:id="@+id/TableRow01"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:layout_margin="10pt">
    <TextView android:id="@+id/title"
          android:layout_height="wrap_content"
          android:text="@string/app_name"
          android:textSize="8pt"
          android:textColor="#1FD343"
```

```
14            android:typeface="normal"
15            android:layout_width="wrap_content"
16            android:layout_weight="1"/>
17    </TableRow>
18    <TableRow android:id="@+id/TableRow02"
19          android:layout_width="wrap_content"
20          android:layout_height="wrap_content"
21          android:layout_marginTop="20pt"
22          android:layout_marginRight="10pt"
23          android:layout_marginLeft="10pt">
24       <Button android:id="@+id/start"
25            android:layout_height="wrap_content"
26            android:layout_weight="1"
27            android:text="Start␣Tour"
28            android:layout_width="fill_parent"/>
29    </TableRow>
30    ...
31 </TableLayout>
```

Listing 4.19: *A part of main.xml: XML layout of the main menu*

As we've told, we can address our layout and its elements because of the *R.java* file. In our main menu activity, called *AMuGu.java*, we want the layout described in *main.xml* to be its content. We also want to react on button clicks with an *onClickListener*. Therefore we first need to address the buttons with *findViewById()*:

```
1 setContentView(R.layout.main);
2 Button start = (Button) this.findViewById(R.id.start);
3 Button read = (Button) this.findViewById(R.id.read);
4
5 ButtonListener bl = new ButtonListener();
6 read.setOnClickListener(bl);
7 start.setOnClickListener(bl);
```

Listing 4.20: *Linking main.xml and its components to the AMuGu activity*

ButtonListener is our own *OnClickListener* class. It checks which button was clicked and then executes the related action, as shown in listing 4.21 and figure 4.1 on page 32.

```
1  public class ButtonListener implements OnClickListener {
2    public void onClick(View v) {
3         if (v.equals(start)) {
4           Intent i = new Intent(con,Tour.class);
5           startActivity(i);
6         }
7         else if (v.equals(read)) {
8         ...
9         }
10   }
11 }
```

Listing 4.21: *ButtonListener: Reacts to buttonclicks*

Tour screen

Figure 4.6: *Tour screen*

The Tour screen has a layout which contains a camera preview. However we would have liked to split split this screens layout and Java code, it wasn't possible because Android currently has nog XML tag equivalent for the camera preview. Thus we've created this simple layout in Java code:

```
1     mPreview = new Preview(this);   // Camera preview
2     mLL = new LinearLayout(this);
3     mLL.setOrientation(LinearLayout.VERTICAL);
4
5     bGPDS1 = new Button(this);
6     bGPDS1.setText("Take picture");
7     bGPDS1.setVisibility(View.VISIBLE);
8     bGPDS1.setClickable(true);
9     bGPDS1.setOnClickListener(mGPDS1Listener);
10
11    mLL.addView(bGPDS1);
12    mLL.addView(mPreview);
13    setContentView(mLL);
```

Listing 4.22: *Adding a camera preview to a GUI using Java code*

As you can see, we first create and configure the layout elements and then add it to the *LinearLayout*. Then we tell the application to use the layout as content.

Figure 4.6 shows what the Tour screen should look like. However, since we don't have an Android mobile phone at the moment and the Android emulator still doesn't support webcams as an alternative for the phone camera, we can't actually see a camera preview. But it should work the way we've programmed it now. The Android emulator also doesn't have a camera button, so for now we created an onscreen button.

Progress bar

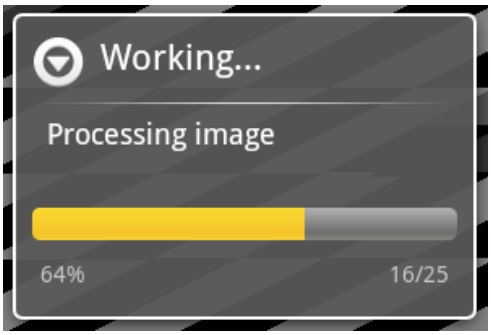


Figure 4.7: *Progress bar*

Calculating the invariant vector of a photo with our GPDS1 algorithm takes some time, so we decided to use a progress bar. It assures that the application is still doing some work, so a user wouldn't panic. How this progress bar works is explained in section 4.3 already.

Info screen

Our last GUI is the screen that shows the info related to the taken picture. As can be seen in listing 4.23, this layout is a little more complicated: it uses tabs, an ImageView and a WebView.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <TabHost xmlns:android="http://schemas.android.com/apk/res/android"
3     android:id="@android:id/tabhost"
4     android:layout_width="fill_parent"
5     android:layout_height="fill_parent">
6     <LinearLayout
7        android:orientation="vertical"
8        android:layout_width="fill_parent"
9        android:layout_height="fill_parent">
10       <TabWidget
11          android:id="@android:id/tabs"
12          android:layout_width="wrap_content"
13          android:layout_height="wrap_content"/>
14       <FrameLayout android:id="@android:id/tabcontent"
15                android:layout_width="fill_parent"
16                android:layout_height="fill_parent">
17             <TableLayout android:id="@+id/content1"
18                          android:stretchColumns="*"
19                     android:layout_width="wrap_content"
```

```
20                  android:layout_height="wrap_content"
21                  android:paddingBottom="5pt"
22                  android:paddingTop="5pt">
23            <TableRow>
24              <LinearLayout android:layout_width="wrap_content"
25                      android:layout_height="wrap_content"
26                      android:orientation="vertical"
27                      android:padding="5pt">
28                <TextView android:id="@+id/Name_lbl"
29                      android:layout_width="wrap_content"
30                      android:layout_height="wrap_content"
31                      android:text="Name:␣"/>
32                <TextView android:id="@+id/Name_val"
33                      android:layout_width="wrap_content"
34                      android:layout_height="wrap_content"
35                      android:textStyle="bold"/>
36              </LinearLayout>
37              <LinearLayout ...
38                ...
39              </LinearLayout>
40            </TableRow>
41            <TableRow>
42              <ImageView android:id="@+id/ImageView01"
43                     android:layout_width="fill_parent"
44                     android:layout_height="wrap_content"
45                     android:layout_span="2"/>
46            </TableRow>
47            </TableLayout>
48          <WebView android:id="@+id/content2"
49               android:layout_width="fill_parent"
50               android:layout_height="fill_parent" />
51      </FrameLayout>
52    </LinearLayout>
53 </TabHost>
```

Listing 4.23: *info.xml: XML layout of the Info screen*

In this layout a couple parts can be distinguished. First there is the TabHost, which is the container for the tabbed window. The TabWidget is the layout element that generates the tabs at the top of the screen. The FrameLayout holds the contents of all the tabs. Which content goes on which tab will be defined in Java code. In our case there are two tabs: one for the general info of a painting and one for the additional info provided in an html file. The general info will be shown in the TableLayout, the additional info in the WebView.

We use this layout in our *ShowInfo.java* class which extends a TabActivity. In the *onCreate()* method of this class, we load all the info data in their corresponding layout elements:

```
1  @Override
2  protected void onCreate(Bundle savedInstanceState) {
3    super.onCreate(savedInstanceState);
4    iPID = this.getIntent().getExtras().getInt("KEY_PID", 0);
5
6    this.setContentView(R.layout.info);
7    Info info = new Info();
8    info.setInfoFromPID(iPID);
9
10   Log.v("SHOWINFO", "Info␣retrieved\n");
```

Figure 4.8: *Info screen*

```
11
12
13  mTabHost= getTabHost();
14  mTabHost.addTab(mTabHost.newTabSpec("content1").setIndicator("Info").setContent(R.id
        .content1));
15  mTabHost.addTab(mTabHost.newTabSpec("content2").setIndicator("Additional␣Info").
        setContent(R.id.content2));
16  mTabHost.setCurrentTab(0);
17
18  wv = (WebView) this.findViewById(R.id.content2);
19  ImageView im = (ImageView)this.findViewById(R.id.ImageView01);
20
21
22  TextView artist = (TextView) this.findViewById(R.id.Artist_val);
23  TextView name = (TextView) this.findViewById(R.id.Name_val);
24  artist.setText(info.getArtist());
25  name.setText(info.getName());
26
27  Log.v("SHOWINFO", info.toString());
28
29  char[] web = new char[20000];
```

```
30    FileReader fr = null;
31
32    try {
33      fr = new FileReader(info.getOptInfo());
34    } catch (FileNotFoundException e) {
35      e.printStackTrace();
36      fr = null;
37    }
38
39    if(fr != null){  //Test if HTML file was found
40      try {
41        fr.read(web);
42        wv.loadData(String.valueOf(web), "text/html", "utf-8"); //Load HTML in WebView
43      } catch (Exception e) {
44        e.printStackTrace();
45      }
46    }
47    else{
48      wv.loadData("<font color=white>No additional information available.</font>", "text
          /html", "utf-8");
49      wv.setBackgroundColor(0);
50    }
51
52    image = Drawable.createFromPath(info.getImg());
53    if(image != null){       //Test if image was found
54      im.setImageDrawable(image);    // Load image in ImageView
55    }
56  }
```

Listing 4.24: *Filling each element of Info screen with related data*

First we get the *pid*, which is passed by the calling activity. This is of course the *pid* of the best matching painting. Now we set the content of this activity to *info.xml*. We then fill the *Info* object with *setInfoFromPID()* method, which we have explained in section 4.5. Now that all the data is loaded, we can start building up our screen. With *getTabHost()* we automatically select the TabHost defined in *info.xml*. Then we add two tabs to it and define their content. Now we create Java objects for each other layout element and fill them with the right content. The WebView and ImageView are loaded with data found at the paths in respectively the *optinfo* and *img* variables of the *Info* object. However if a museum wants to use some content found on the internet in their html file, access to the internet has to be granted to this application. This is done by adding the following line to *AndroidManifest.xml* (see section 2.1.3):

```
1   <uses-permission android:name="android.permission.INTERNET" />
```

Listing 4.25: *Additional line in AndroidManifest.xml to allow internet access*

4.7 Results

Developing an application comes down to creating and testing each part separately, followed by combining all parts together. Google provided the Android emulator for testing applications under development. Our application consisted of the parts described in each section: creating and using the database, capturing images, computation of the feature vectors and matching the captured images.

When creating the database, we had to be sure all paintings were actually added when parsing the XML files. The only possible way of testing this, was actually reading all the data contained in the database. This test also ensured that our technique for reading from the database was correct. When executing this test, we noticed that all the data we had created in the XML file was present in the database output. Therefore we were sure that our database worked correctly.

The next step was capturing an image from a camera. We immediately noticed the camera preview didn't actually show a preview. However, we assumed the emulator would use our webcam for capturing image. When capturing an image, the emulator stored the image on a virtual SD-card. When viewing this image, we noticed that the emulator always uses a predefined picture as result of taking a picture. This picture is shown in figure 4.9.

Figure 4.9: *Result of emulator camera*

Next we tested if our feature vector calculation algorithm resulted in the correct numbers, by comparing the results in the log files to Matlab results on figure 4.9. As we've told in section 4.4, calculating the feature vector of the taken picture proved to be time consuming. Our original implementation required a computation time of 1m43s (see table 4.1 on page 50). Optimization of the code was needed for providing user-friendly computation times. By reverting to floats, rewriting the method for calculating powers and using the "enhanced" for-loops we were able to reduce the computation time to 8s. This result seemed acceptable for our application.

For testing the matching process we combined it with all other parts of the application, as it need the database and a calculated feature vector. Unfortunately, we didn't have access to a real Android mobile phone and the emulator didn't allow taking pictures, so we weren't able to take pictures of real paintings. But by changing the threshold value (see section 2.2.2, 3.5 and4.5) to 1, we were able to test the general program flow of our application. All steps were executed in the right order and a result was shown on the Info screen (see section 4.6.2). Obviously this result was a false positive as it wasn't rejected by the threshold specifications. Afterwards we added figure 4.9 to our database. Having this figure in the database, our application always

returned this image as best match, which is correct.

In general, we developed and implemented an Android application able to compare database images with a newly taken picture in 8s. Although we didn't have access to a mobile phone for testing the recognition rate, our experiments in Matlab showed that 88% of the retrieved images were correct. Moreover we implemented a GUI for each activity of the application, of which one displays info related to a retrieved image.

4.8 Conclusion

Android is very flexible and provides many tools for developing applications. This allowed us to develop our museum guide application in a limited amount of time. We've used a wide range of Android's possibilities. First of all we've used its SQLite database management system. We filled our database using XML files, for which we needed the XML parser. We have created Java classes which extend the Activity and Service classes, both core components of an Android application. We've passed information to these Activities and Services in several ways: using global variables, intents and the database. We've tried to implement computationally heavy calculations, but we noticed it took too long to use them in real situations. So we've optimized our code by, among others, using the OpenGL|ES libraries. After these optimizations, we noticed a serious decrease in calculation time, but we still added a progress bar to make our application more user-friendly. For this, the usage of threads was needed. In our XML-based GUIs we've also used many different elements: LinearLayouts, TableLayouts, Buttons, WebViews, ImageViews, Tabs, ...

Although we haven't been able to test it in a real situation on a mobile phone, we have good hopes about our application. The general program flow and calculation work perfectly in the emulator, so we expect that no or minor changes would be needed for an on-phone implementation.

Chapter 5

Java management interface

We already discussed both the image recognition part and the Android-implementation part of our thesis. As an extra addition to this, we decided to start creating a management interface. The goal of this interface is to allow museum personnel to add and edit the museum specific data. This includes for example the creation of a complete new dataset for a museum, and adding paintings and information. Because we like to program cross-platform, we chose to implement this application in the Java language. As an extra advantage, we can reuse parts of the Android-code. In this chapter we will look at the GUI, so the reader gets an overall impression of our goal.

5.1 Implementation idea

During development of the Android application (see section 4.2) the database of paintings in a museum was added manually. This was done by writing information in an XML file, calculating the invariants by using Matlab and storing extra images in the directory structure on the (virtual) memory card. We only had to input a limited amount of information, so this approach was enough. However, in real-world situations the museum personnel needs a more user-friendly application. Figure 5.1 shows an image resembling a GUI suitable for this goal. The actual implementation of this program has not been finished.

GUI

Basically the GUI for the AMuGu management interface is divided in three main areas. The first is the menu bar. This allows the creation of a new museum dataset, saving the current data, and generating future archive files which could transfer both XML, HTML and multimedia content like images, movies or audio at once to the telephone. This could be useful in a real museum where a user could just download the package and the application.

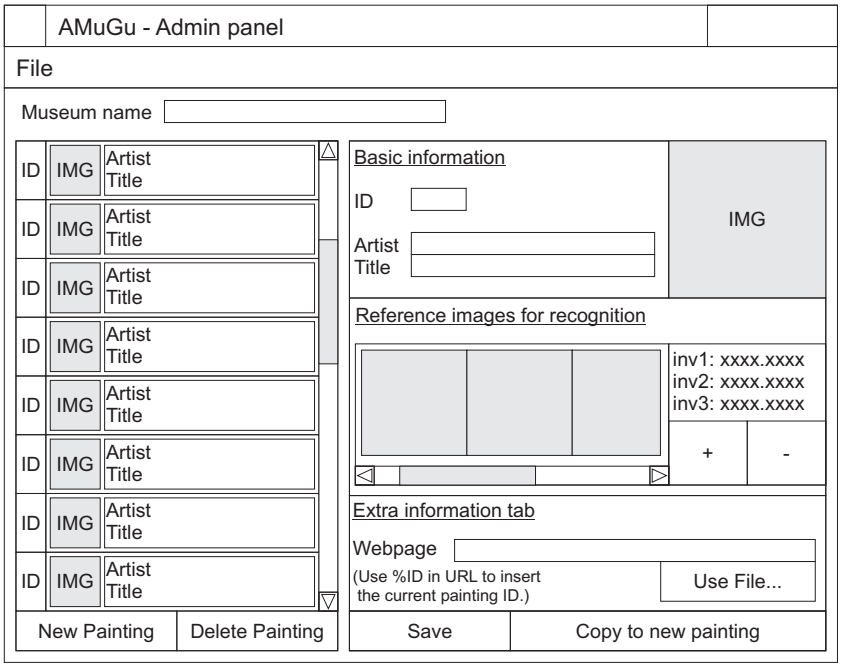


Figure 5.1: *Proposal for a user interface GUI.*

The next part in the GUI is the left-hand area on the main screen. This is a list of paintings in the current database, with an identifier, a thumbnail, artist name and title per painting. Underneath the list the administrator can delete the selected painting or create a new painting. By selecting a painting, the right-hand side of the window is filled in.

The right-hand side of the main windows contains details for the currently selected painting in the list. It consist of three panels and a couple buttons. The first panel is "Basic information" where ID, Artist and Title can be edited. There's also a picture of the painting. When this picture is clicked, the administrator can choose a new image. The image is shown to the AMuGu

user when the painting has been detected. It's completely irrelevant for the image recognition. The next panel is called "Reference images for recognition". This is where actual photos of the current painting have to be added. Preferably, the administrator adds photos made by a cellphone because they resemble actual user input most. Reference images can be added or removed, and when clicked on one, the numbers for the GPDS1-invariant (see section 3.1) are shown. The last panel is about the "Extra information tab". When the webpage field is left empty, AMuGu won't show the tab. Webpages can be either on the internet or intranet by inserting a URL, or refer to a file in the package for the museum. Further, the administrator can save the current painting, or copy the current data to a new painting.

Behind the GUI

Aside from the user interface itself, the management interface needs some work behind the scenes. First, it should implement methods for working with XML data. All information about the paintings is stored inside an XML file, which is opened by the Android application and imported in its SQLite database system. Furthermore, the entire filesystem structure should be defined: where to store the XML file, and how about the reference images? They need to be stored in a folder, but this folder shouldn't be included in the archive which is uploaded to the telephone. The extra information tab also needs a separate folder containing HTML-files, images, movies, sound,... And last but not least, a suitable packaging system should be found. There are quite some options, for example Tar or Zip. When a packaging system has been chosen, it should also be implemented in AMuGu.

Chapter 6

Conclusion

Our thesis originated with "Image recognition on an Android mobile phone". The goal was to create a museum guide application for Android, an open source mobile phone operating system.

The literature study (chapter 2) illustrates our research, where the Android operating system is introduced and explained. Papers about image recognition were read and we decided to go with a method based on global features, called *color moment invariants*. The choice for this method was mainly to reduce computational complexity, as the mobile phone has strictly limited processing power.

The next step was our research done using Matlab (chapter 3). We implemented some of the methods described in our literature study, and went on to compare the results for different invariants (see section 3.2) and found out one of the simplest invariants, which we called GPDS1, was suitable for our application. It was not the best invariant in the test, but it was quite close to the best, without the need for heavy calculations. The matching of vectors of invariants was described in section 3.3, where a threshold was also calculated.

After finding out how to do the recognition, we made an implementation in Android (chapter 4) where we made our AMuGu-application. We started by describing what the application should do, and moved on to the database design. The algorithm was translated to Java code, and we optimized our implementation to reduce computation time. The original implementation in Android required 103 seconds to calculate the invariants for a single image, after optimization it was reduced to 8 seconds. Matching was also rewritten for Android, and our graphical user interface was explained. During our implementation, we discovered Android is a flexible and developer-friendly platform. We didn't have Android hardware available, however we did create an application which works in the Android emulator and returns the same results as our Matlab-code.

As an addition to our thesis, work started on a Java management interface (chapter 5). The goal of this application was to allow museum administrators to add and modify data for the AMuGu-system. During development of AMuGu, we had to manually edit the XML file, which wasn't user-friendly. Therefore we designed a GUI for an application to solve this problem. However, we didn't complete this application due to time constraints.

In general, when looked back at chapter 1 about the goal of the thesis, we can conclude we reached our original goals. We did research about image recognition and Android's possibilities, found a lightweight algorithm and implemented it as an Android application which acts as a museum guide.

Future use-case

When the management interface as we proposed in section 5.1 is completed, extra functionality could be implemented. It would be nice to have one central "AMuGu" website which distributes the AMuGu-application itself. The website could also be used to store a list of museums using the AMuGu system. The AMuGu-application could be distributed with this list, which contains museum names, a summary of what is on display, a picture of the front of the museum, GPS-coordinates etc. Then, a user could walk freely in the city, take his cellphone and let the software search for nearby museums. When he arrives at the museum, he pays the entry fee and is given a short code. The list of museums could per museum contain information on how to get the required AMuGu-data using a wireless network. The user switches on the WiFi on his cellphone, and AMuGu automatically connects to the network, and asks the user for his code. He enters the code in the telephone, and AMuGu downloads the museum's data. Now, everything is ready to start the tour.

Of course, the Java interface would have to be modified for this. It should include a new window with museum information. For user-friendlyness, it could also contain a button which uploads the new information to a central AMuGu project server.

The user only had to install the AMuGu application himself. Another possibility is to distribute SD-cards at the museum which contain both the AMuGu application and the data.

Bibliography

[1] Bay, H., Ess, A., Tuytelaars, T. and Gool, L. V. [n.d.], 'Speeded-up robust features (surf)'.

[2] Bay, H., Fasel, B. and Gool, L. V. [2006], Interactive museum guide: Fast and robust recognition of museum objects, *in* 'Proceedings of the first international workshop on mobile vision'.

[3] Föckler, P., Zeidler, T., Brombach, B., Bruns, E. and Bimber, O. [2005], 'PhoneGuide: Museum Guidance Supported by On-Device Object Recognition on Mobile Phones', *Proceedings of International Conference on Mobile and Ubiquitous Computing (MUM'05)* .
URL: *http://www.uni-weimar.de/medien/ar/PhoneGuide/results.htm*

[4] Google [2009*a*], 'Android API Demos'.
URL: *http://developer.android.com/guide/samples/ApiDemos/index.html (Last consulted: 04/05/2009)*

[5] Google [2009*b*], 'Android Dev Guide'.
URL: *http://developer.android.com/guide/index.html (Last consulted: 04/05/2009)*

[6] Google [2009*c*], 'Official Android Website'.
URL: *http://developer.android.com (Last consulted: 04/05/2009)*

[7] Green, R. [2008], 'Floating Point Java Glossary'.
URL: *http://mindprod.com/jgloss/floatingpoint.html*

[8] Lowe, D. [2004], 'Distinctive Image Features from Scale-Invariant Keypoints'.

[9] Mindru, F., Tuytelaars, T., Gool, L. V. and Moons, T. [2003], 'Moment invariants for recognition under changing viewpoint and illumination'.

[10] *Open Handset Alliance* [2009].
URL: *http://www.openhandsetalliance.com (Last consulted: 04/05/2009)*

[11] Paul, R. [2007], 'Why Google chose the Apache Software License over GPLv2 for Android', *Ars Technica* .
URL: *http://arstechnica.com/news.ars/post/20071106-why-google-chose-the-apache-software-license-over-gplv2.html (Last consulted: 03/05/2009)*

[12] Sun Microsystems, I. [2005], 'The Enhanced For-Loop'.
URL: *http://java.sun.com/developer/JDCTechTips/2005/tt0505.html#2 (Last consulted: 04/05/2009)*

[13] Wikipedia [2009], 'Mahalanobis distance'.
URL: *http://en.wikipedia.org/wiki/Mahalanobis_ distance (Last consulted: 04/05/2009)*

Appendix

All source files, images and other important documents can be found on the supplied CD-ROM.