



RESEARCH AND IMPLEMENTATION OF ANDROID BASED OPTICAL MUSIC RECOGNITION

Sofia Papadopoulou

SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENT FOR THE DIPLOMA OF
ELECTRONIC AND COMPUTER ENGINEERING
OF TECHNICAL UNIVERSITY OF CRETE.

October 2015

Thesis Committee

Ass. Professor I. Papaefstathiou, Thesis Supervisor
Professor M. Zervakis
Professor A. Dollas

“Music gives a soul to the universe, wings to the mind, flight to the imagination and life to everything.”

Plato

Abstract

As stated on Gerd Castan's page about OMR,

“Doing OMR is hard. Very hard. If you follow the OMR science links you will find much more theses about OMR than there are public available programs”.

The motivation behind this thesis was to bring music to life. It is a great opportunity to have the software that can instantly produce a song from a sheet music in a course book. Presently, the respective commercial software is mainly available on a desktop platform, which requires a high-performance computer and a scanner to scan a sheet music. This project will survey the different techniques that have been used to perform OMR on printed music scores and an application by the name of AOMR will be developed. AOMR project renovates this kind of software by applying it on a mobile platform instead. This application, which highly involves image processing techniques and artificial intelligence applications, can recognize a scanned image of sheet music to be interpreted and exported as an audible representation via MIDI synthesis while handling resource utilization on an Android mobile phone platform. Limited processing performance and memory capacity, including the lack of image processing and other related APIs, are major issues that cause the algorithms used in the application to be different from traditional approaches applied in software on a PC platform. Such program is meant not only to be a simple OMR application but also the starting point for a future work about specialists having an interactive software to rely on for both as a practical tool for work and educational purposes.

Acknowledgements

I would like to thank my thesis supervisor, Professor Ioannis Papaefstathiou, for his guidance, constructive remarks, as well as for giving me the opportunity to expand my knowledge in the field of Optical Recognition and Android development. Moreover, I would like to thank Stamatis Andrianakis for his support. Furthermore, I would like to thank Professors Michalis Zervakis and Apostolos Dollas for their contribution.

Last but not least, great thanks to my friends; Stauroula, Vasia, Marietta, Vivi and Padelis for their love, support and encouragement.

I would also like to thank my parents and my sister for all their support. Without them, I would not be here right now.

Contents

Introduction.....	9
1.1. Objective and Contribution.....	9
1.2. Thesis Outline	10
Background.....	11
2.1 Music theory and terminology.....	11
2.2 Android project	13
2.2.1. Android revolution	13
2.2.2 Architecture Android Software.....	15
2.2.2.1 Application Framework	16
2.2.2.2 Process Handling	17
2.3 OMR Overview	18
2.4 Artificial Neural networks	19
2.4.1 Feed-forward Neural Networks.....	21
2.4.2 Training of Feed-Forward Network	22
2.4.2.1 Backpropagation Algorithm	23
2.5 Discrete Fourier Transform (DFT)	24
2.6 RLE encoding.....	25
2.7 Projections	26
2.8 Music File Formats	27
2.9 Related Works	28
2.9.1 PC Platform	28
2.9.2 Android Platform	29
2.9.3 OpenOMR.....	30
Development Tools	31
3.1 Java.....	31
3.2 Java Development Kit	31
3.3 Eclipse IDE	32
3.4 Eclipse ADT.....	32
3.5 Android Studio	32
3.6 Java and Android API	33
Modeling and System Architecture	34
4.1 System architecture	34
4.2 System Modeling	37

4.2.1 Image Selection Process	37
4.2.2 OMR ALGORITHMs & FLOW PROCESS	37
4.2.2.1 Image pre-processing	36
4.2.2.2 Musical Object Location	39
4.2.2.3 Musical feature Classification	43
4.2.2.4 Audio Synthesis	46
4.3 Application Design	49
4.3.1 Functional Requirements	49
4.3.2 Supporting Different Devices	49
4.3.3 Design principles and application's interface	51
4.4 AOMR	51
Implementation	52
5.1 Programming Language Choice	52
5.2 APIs Structure	52
5.2.1 AndroidManifest.xml	52
5.2.2 Basic Components	53
5.3 Fundamental implementation hurdles	60
5.4. Performance issues	64
5.4.1. Performance guideline	64
5.4.1.1 Optimizing Image Handling	63
5.4.1.2 Improving responsiveness	63
5.5. Allotment time	66
Results	68
6.1 OMR Evaluation	68
6.1.1. Basic Symbol Recognition Evaluation Results	69
6.2 Application Resource Utilization	77
6.2.1. Local Server	77
6.2.2. Android framework	79
Future Improvements	82
Conclusion	84
Bibliography	85

List of figures

Figure 2.1: Musical Semantics	11
Figure 2.2: Example of a key signature designating the highlighted note to be played one semitone higher	11
Figure 2.3: Examples of time signatures.....	11
Figure 2.4: Worldwide smartphone OS market share (2010-2015)	12
Figure 2.5: Android versions (August 2015)	13
Figure 2.6: Google Android Architecture.....	14
Figure 2.7: A graphical representation of a simple perceptron.....	18
Figure 2.8: Multilayer perceptron.....	20
Figure 2.9: Magnitude Image of DFFT.....	23
Figure 2.10: RLE encoding.....	24
Figure 4.1.: System architecture diagram.....	33
Figure 4.2.: Recognizer architecture diagram.....	34
Figure 4.3: Image pre-processing – UML diagram.....	36
Figure 4.4.: FFT Implementation – UML diagram	37
Figure 4.5: “Song for Guy” by Elton John, Skewed	37
Figure 4.6: FFT of figure 2.10	39
Figure 4.7.: Musical object Detection – UML diagram	41
Figure 4.8: Level 0 segment	41
Figure 4.9: Level 1 segment	42
Figure 4.10: Level 2 segment	43
Figure 4.11: Neural Network – UML diagram.....	44
Figure 4.12: Backpropagation training – Flow chart.....	45
Figure 4.13: Midi generation – UML diagram.....	46
Figure 5.1: API – UML diagram	51
Figure 5.2: Main application menu in landscape and portrait orientation of the screen	53
Figure 5.3: Display screen destination address option to import an image.....	54
Figure 5.4: Display dialog box to define FFT size	54
Figure 5.5: Warning Message for FFT size features.....	55
Figure 5.6: Display Dialog FFT Process.....	55
Figure 5.7: Display recognition Parameters before starting recognition process	56
Figure 5.8: Display Dialog Recognition Process	56

Figure 5.9: Display recognized score	57
Figure 5.10: View while MIDI is generated.....	57
Figure 6.1: A.1- Recognized	70
Figure 6.2: A.2-Recognized	71
Figure 6.3: A.3-Recognized	71
Figure 6.4 : OpenOMR CPU Utilization	75
Figure 6.5: OpenOMR Memory Utilization	75
Figure 6.6 : AOMR CPU Utilization.....	77
Figure 6.7 : AOMR Memory Utilization	78

List of Tables

Table 6.1: FFT results	67
Table 6.2: Note head results.....	68
Table 6.3 : Pitch results.....	68
Table 6.4: Neural Network Results	73

Introduction

1.1. Objective and Contribution

In terms of human past knowledge, music may be one of the few things which we are certain that follow us since pre-historic times. As a way of representation, musical score notation has always been the main source of musical expression for non-hearing systems. Optical music recognition (OMR) – a computer system that can “read” printed music, has been under intensive development for over five decades (first published OMR work was carried out in the mid of 1960s by Pruslin [1] and while numerous achievements have been made, there are still many challenges to be faced before it reaches its full potential. OMR has still much a promise: you could listen to a piece of written music without any training in musical notation; a clarinetist could scan a tune and have it transposed automatically; a soloist could have the computer play an accompaniment for rehearsal; a music editor could make corrections to an old edition using a music notation program; or a publisher could reduce an orchestrated work to a piano part with little fuss, and convert the piece to Braille with almost no extra work. This is the aim of Optical Music Recognition (OMR): to convert optically scanned pages of music into a versatile machine-readable format. OMR applications based on the computer’s software focus on the performance and the accuracy of the result regardless of resource utilization since a computer desktop has a high processing power and a large memory size, which are upgradable and configurable easily. However, to build this kind of system on a mobile phone is quite a challenge. Apart from the problems of limited computational resources and electrical power, there are other limitations to be concerned in terms of technical software development as well. The system puts priority on CPU and memory utilization over the accuracy in order to maintain system stability and battery power on a mobile phone. I adapted best practice for

performance design from the Android official developer guide to make the process consume system resources in an optimal level while providing an acceptable performance. Compared to its expensive non-portable counterparts, a portable mobile music recognition system would serve as a convenient and cost-efficient alternative for the average music enthusiasts.

1.2. Thesis Outline

Finishing the first chapter, which was originally a small introduction to presenting the Optical Recognition System, the purpose and contribution of this thesis, it follows a brief description of what will contain the following chapters. Chapters two three incorporated to present the concepts and the technology used, in a more theoretical base, while in the following chapters we will analyze in depth the design and the system's architecture implemented. Finally the results of the project will be presented and discussed. In detail chapter:

2. Background: The basic theory to be used for the design and implementation of the project is presented, including OMR and Android Software fields. Related published work in the OMR field will be briefly presented, as well.

3. Development Tools: The tools that helped us in the implementation of our system. The Java language is also presented and a briefly comparison with the first and Android API.

4. Modeling and System Architecture: In depth description of the application's and OMR system modeling and architecture.

5. Implementation: a technical description of the various components and the way they interact with each other is provided and the actual presentation of the application.

6. Results: The results in terms of OMR accuracy and application's performance.

7. Future Improvements: Some useful conclusions we extracted of the work carried out for this project and the future development of the OMR application implemented in this project.

Background

This chapter provides a brief introduction into the field of music notation before we delve into the more analytical area of Optical Music Recognition (OMR). The goal of this chapter is to build the basic foundations needed in order to understand how an OMR system works along with the theory used for the implementation of the optical music recognition system. We will also present the most popular commercial OMR systems available. Finally, some underlying theory which is used in the design and implementation of the OMR system is then presented. In the following chapters, we will see how the above concepts are used in the various components of our system.

2.1 Music theory and terminology

In order to understand how OMR systems work, it is important to have a notion of the underlying concepts in music theory and its associated terminology. Music notation emerged from the combined and prolonged efforts of many musicians. They all hoped to express the essence of their musical ideas by written symbols. Music notation is a kind of alphabet, shaped by a general consensus of opinion, used to express ways of interpreting a musical passage. The fundamental music element in a music score is the staff. The staff consists of five horizontal lines on which musical notes lie. The lines and the spaces between the lines represent different pitches. Lower pitches are lower on the staff and higher pitches are higher on the staff. We use Clefs to tell us which notes correspond to which lines or spaces. The most common clefs are the Treble Clef (also known as the G clef) and the Bass Clef (or F clef) (Fig. 2.1). From here we can figure out the other note names simply by going forward or backward through the musical alphabet: A, B, C, D, E, F, G. If we need more notes above or below the staff we add Ledger Lines, which extend the range of the staff. Usually, a note consists of a note head and a connected stem. In certain cases, where two or more notes are played at the same time, all stems but one are

omitted. The appearance of the note head gives information on the duration of a particular note, which is shown in Fig. 2.2. Positioned near the clef sign in most cases, a time signature affects neither the time nor the pitch value of notes. While the numerator specifies the number of beats per bar, the denominator indicates the note value of one beat. A bar is referred to as space between two neighboring bar lines that are drawn perpendicular to the staff lines. The letter C (see Fig. 2.3) indicates the common time which would otherwise be represented by the 4/4 meter. Common children's tunes, which are subject to this work, have been written in a 3/4 or 4/4 time signature while other musical genres like Jazz or classical music show a higher variability in this particular property.

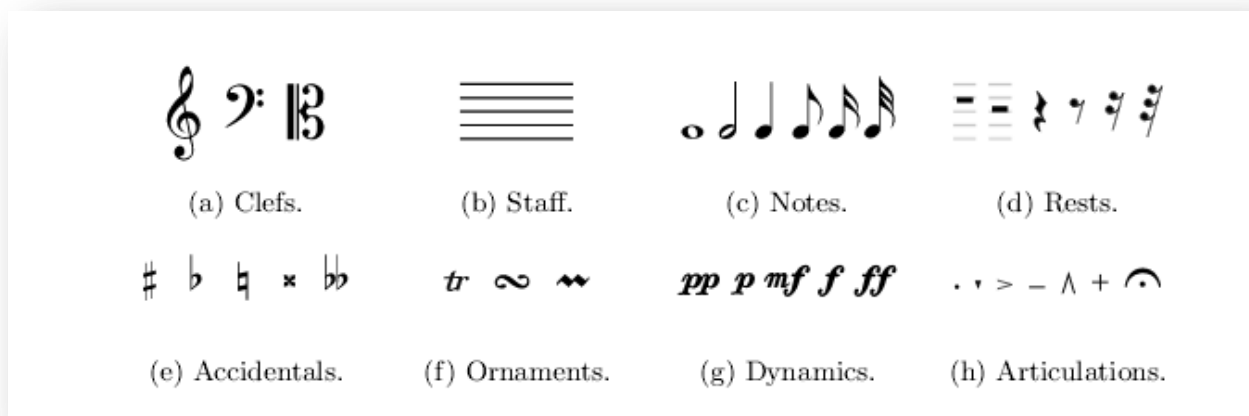


Figure 2.1: Musical Semantics



Figure 2.2: Example of a key signature designating the highlighted note to be played one semitone higher



Figure 2.3: Examples of time signatures

2.2 Android project

2.2.1. Android revolution

Android, one of the most popular operating systems today, is an open source project developed by the Open Handset Alliance and held by Google Inc. It is often wrongly attributed to the operating system based on Linux kernel alone, but in fact it contains a middleware and a variety of additional applications. For these reasons it is fairer to say that Android is a software stack for mobile devices. Inheriting all the security features of Linux and all memory management techniques and processor features, Android becomes a fairly reliable operating system. But one of the most appreciated aspects of Android is its openness. The source code has been revealed to the public, enabling many developers around the world not only to have better understanding of what is happening in the background of the system, but also to actively contribute to the project. It makes Android more flexible, allowing new cutting-edge technologies to quickly incorporate in the system and developers to adapt the system to their own needs and in their own material. It has been eight years now from the moment the first Android phone has been released to the public (November 2007). Within this period of time Android has come a spectacular way from an early stage project with prototype devices to being the most popular OS in the smartphone market in the world.

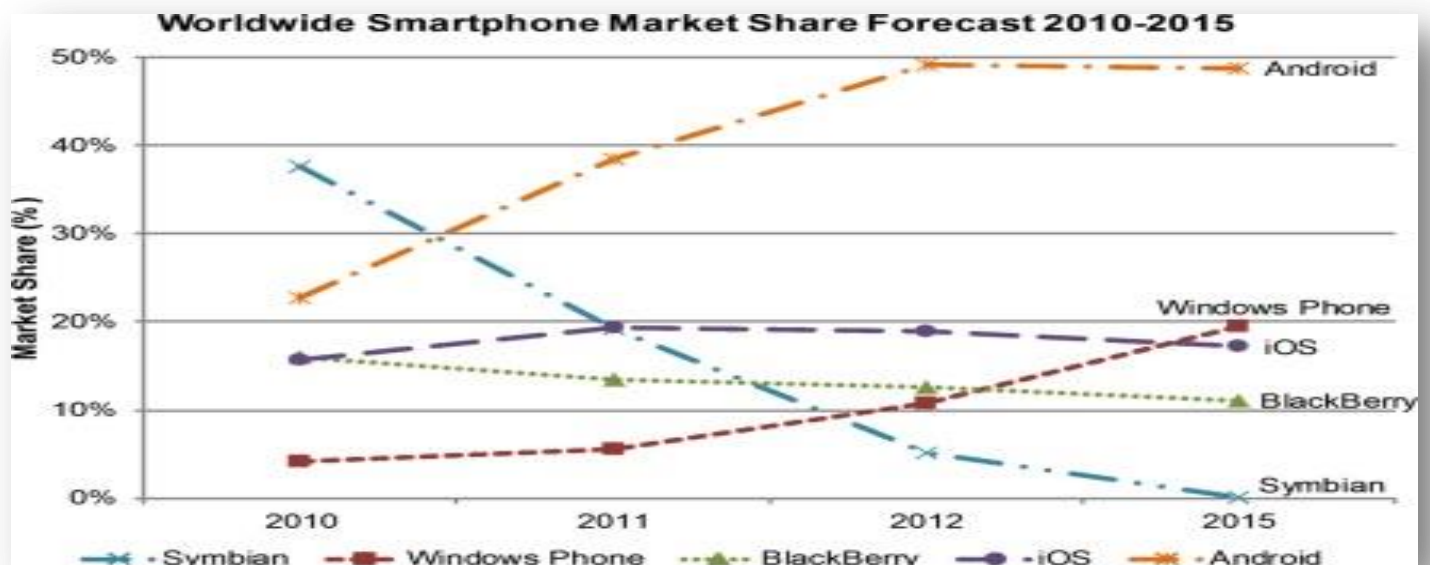


Figure 2.4: Worldwide smartphone OS market share (2010-2015), source: <http://www.biomedcentral.com/1472-6947/12/67/figure/F5?highres=y>

Naturally, Android itself is not a single operating system. It comes in many versions with new major updates being released every half year or even more frequently. So far all new updates have kept a total backward compatibility. On April 30 2009, the official update version 1.5 for Android was released, and the latest to date version is 5.0 with code name Lollipop. The most up-to-date chart (figure 2.5) from Android Developer web site shows a distribution of Android versions.

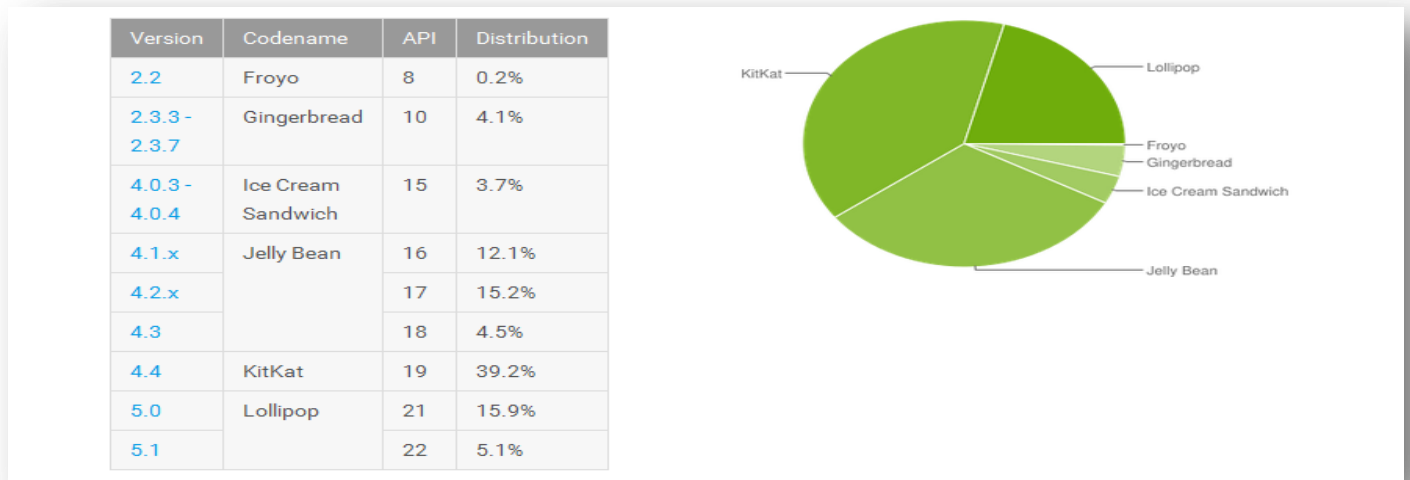


Figure 2.5: Android versions (August 2015), source: *Android Developers*
<http://developer.android.com/resources/dashboard/platform-versions.html>

The actual distribution of Android versions on the global market can definitely be regarded as the long term trend, showing the speed of the upgrade process. This is a very important observation for developers before they release applications that support from a certain version onwards. When designing a new Android application, one has to take under consideration not only a variety of platform versions but also a target device's specification. The most important feature is probably the screen resolution. Oppositely to Apple's iPhone, Android is not limited to a single device and Google does not manufacture its own phones. As it was mentioned beforehand, Android is a software stack and thus it can be installed on practically any device that satisfy a minimal set of requirements and it was meant to support a variety of resolutions or even screen orientations.

2.2.2 Architecture Android Software

This section covers the fundamental aspects of designing an Android application which are necessary in order to understand the following discussion about AOMR *app*. The Android-powered as shown in the figure below, can divide into five levels: The Linux Kernel, parent libraries (native libraries), the Android Runtime, the Application framework and Applications. Kernel used is the Linux 2.6 line, modified because of particular needs in energy management, memory management and operating environment. Because Android is for devices with little main memory and low power processors, libraries for intensive work of the CPU and GPU translated into modified device parent libraries. Basic libraries such as libc and lipm were developed specifically for low power consumption. The Android Runtime consists of the Dalvik virtual machine and libraries core Java. Dalvik virtual machine is byte code compiler who has converted from byte code to Java byte code Dalvik. Framework is written in the Java and provides deductions from parent libraries and capabilities in Dalvik applications. Android apps run their own Dalvik virtual machine and can contain various items such as those will be analyzed below.

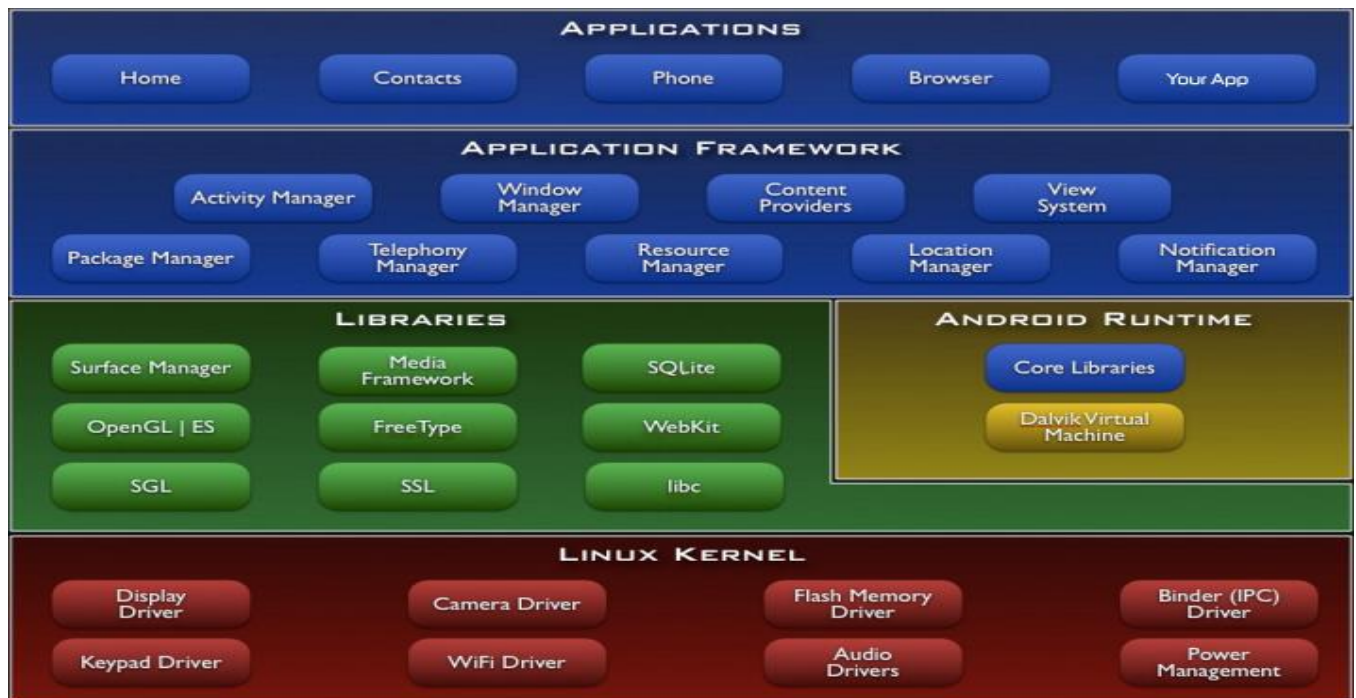


Figure 2.6: Google Android Architecture, *source*: <http://developer.android.com/about/index.html>

2.2.2.1 Application Framework

Every Android application must have a single manifest file in a root directory (named *AndroidManifest.xml*). This file contains essential information about the application which is required by the system to run it. All new components have to be registered there along with permissions needed to perform actions requested by the application. The file is also used to inform the system if a developer wishes to expose the application's data to other *apps*. It specifies the application's reaction to various system events such as incoming calls or photo capturing. Within the Manifest file developers can specify if they want one of their activities (described below) to become a *launch* activity. Applications which contain a launch activity are added to the Android's menu screen so they can be launched independently from other apps.

All Android applications contain zero or more of the following components:

- **Activity** – a single piece of user interface. In a system it runs independently from other elements although users have a smooth experience of different activities appearing on the screen in a sequence. Activities usually contain some graphical elements (customized or taken from a pre-defined set) and constitute a specific action that users can perform like writing an email or selecting contact from a list.
- **Service** – a separate part of the application without a graphical representation (running in a background). Service itself does not start a separate thread because it runs in context of the activity or the broadcast receiver which has started it. However, it usually should use a working thread to perform some complex computation or lengthy I/O operations.
- **Content Provider** – a mechanism that allows applications to share data between each other or to simply persistently save some information. It is basically an interface which provides standard methods to access data like query, insert, update, delete. No specific type of data structure is imposed by the system, it can be a single file or a SQLite database. Content provider is uniquely identified by its *authority* and can contain many types of data objects (many tables in a database). The most common method of accessing an object is to query Content Provider with a specific *content Uri* which has a generic form of:

content://<authority>/<type path>/<id>/

- **Broadcast Receiver** – a part of the application which responds to actions broadcasted by the system itself (dimming the screen, capturing a new photo) or by other parts of the application (the same one or not). Broadcast Receiver does not have its own graphical representation but it can initiate certain actions to keep users informed about its work. This could be done for instance by creating a notification in a status bar or updating a desktop widget.

As it was mentioned previously Android, is an event-driven system. These events are called *Intents*. Intent API is a very powerful mechanism that manages interactions between application components across the whole system. Developers do not have to write any additional lines of code in order to integrate their application with other ones as long as they know their set of supported Intents. Also an application on Android has other elements such as: strings, styles, colors, drawable, xml. Important files containing topics and other data that has to do with displaying application.

2.2.2.2. Process handling

One of the key issues when designing an operating system for mobile devices is memory management. Android is trying to do as much as possible in the background without bothering anybody, but at the same time there is remarkable space for alterations and modifications. Therefore, developers should at least be aware of the most basic aspects of process handling not to create applications that would be 'arrogant' or even 'hostile' to the system. Very often Android has to deal with situations of memory shortage. The system has to reclaim resources in order to allow new processes to run. In such situations Android ranks all active processes according to the following order:

- **Foreground process** – the process which hosts a foreground activity or a service that is bound to a foreground activity. It basically means the part of the system that the user is currently interacting with, so at any given time there are only few such processes. These processes are killed only in absolute critical situations as not terminating them will most likely cause lack of responsiveness (or even displaying error messages to the user).
- **Visible process** – a process that does not have any active components, but still is visible to the user. A good example is a process which hosts an activity that

launched another not-full-screen activity. Visible processes are considered important and will only get killed if the system cannot find enough memory for foreground processes.

- **Service process** – a process which basically hosts a service and at the moment it is neither a *foreground* nor a *visible* process. Those processes however may be doing some important tasks for the, user like playing music in a background, or downloading some data from the Internet.
- **Background process** – a process which handles an activity that is currently not visible. These processes are likely to get killed at any given time so activities should be prepared for that.
- **Empty process** – a process without any of the application's components. The only reason why the system holds these processes alive is for caching purposes.

Developers should constantly be aware of the fact that their processes could get killed and reestablished by the system in the background even if they did not expect such a situation to happen in the first place. To allow a smooth navigation between different screens, Android introduces a number of helping methods to save and restore the state of activities. It is good practice not only to use these methods whenever there is a need to, but also to do it wisely. For instance, no lengthy operations should ever be performed within them. A good example is an activity where users can edit a new text message being interrupted by an incoming call. The screen is captured by the new activity, decreasing the priority of the old one. Therefore, the process which handles the message-editing activity is more likely to get killed. Users would probably expect to be able to continue editing the text of the message after finishing the call. However, this would be impossible if the text had not been persistently saved beforehand.

2.3 OMR Overview

Breaking down the problem of transforming a music score into a graphical music-publishing file in simpler operations is a common but complex task. This is consensual among most authors that work in the field. A typical framework for the automatic recognition of a set of music sheets encompasses four main stages:

1. Image preprocessing
2. Musical object location
3. Musical feature reconstruction and classification

4. Construction of a musical notation model representation (MIDI, MusicXML, etc.)

In the image preprocessing stage, several techniques – e.g. enhancement, binarization, noise removal, blurring, de-skewing – can be applied to the music score in order to make the recognition process more robust and efficient. The reference lengths staff line thickness (`staffline_height`) and vertical line distance within the same staff (`staffspace_height`) are often computed, providing the basic scale for relative size comparisons. Then we can proceed in the staff line detection/removal to obtain an image containing only the musical symbols. The output of the image preprocessing stage constitutes the input for the next stage, the recognition of musical symbols. This is typically further subdivided into two parts: symbol primitive segmentation; and symbol recognition. In this stage the classifiers usually receive raw pixels as input features. In the fourth and final stage (final representation construction), a format of musical description is created with the previously produced information. The system output is a graphical music-publishing file, like MIDI or MusicXML.

2.4 Artificial Neural networks

Artificial neural networks or simply neural networks are often seen as implying an intention to duplicate the information processing of the nervous system. Early work in the field was indeed inspired by biological studies of the brain function, and [2] and [3] described systems of neurons and learning rules derived from biological models. The relationship between a modern neural network and the brain of an animal lies in the idea of performing computations by using the parallel interaction of a very large number of simple entities. Neural networks have been used in connection with many different applications. The tasks for which they are used are generally problems of pattern recognition/classification or function approximation. The crucial feature of neural networks is their ability to learn how to make the desired mapping from inputs to outputs without explicitly having to be told the rules for doing so. Instead, they adjust their internal connections based on a number of examples of the required mapping, and are then used to generalize from the given examples to others that they have not previously seen. Thus they are used for capturing patterns in sets of data, and use these captured patterns to perform the required computations and so neural networks have proven to be accurate for Optical Character Recognition applications as a classifier. Most of the discussion to follow is based on [4] and [5].

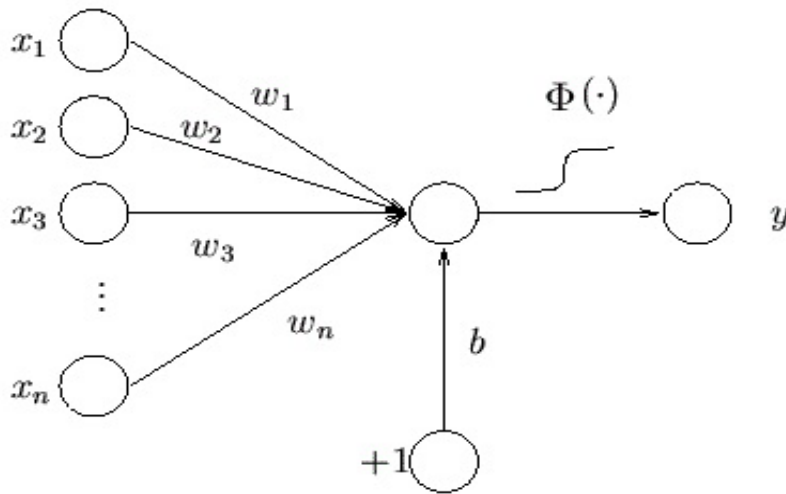


Figure 2.7: A graphical representation of a simple perceptron

The artificial neuron is the basic unit from which a neural network is constructed. It can be viewed as a many-to-one function that will either fire or not fire based on the summed value of its input(s). Figure 2.7 shows a diagram of an artificial neuron. Here y is the output signal, Φ is the activation function, n is the number of connections to the perceptron, w_i is the weight associated with the i th connection and x_i is the value of the i^{th} connection. The b in the figure represents the threshold [6]. The perceptron neuron can take several weighted inputs and summarize them, and if the combined input exceeds a threshold it will activate and send an output. Which output it sends is determined by the activation function and is often chosen to be between 0 and 1 or -1 and 1. Since the derivative of the activation function is often used in the training of the network, it is convenient if the derivative can be expressed in terms of the original function value, as few additional computations are needed to calculate the derivative in this case. The equation for a neuron can be written as:

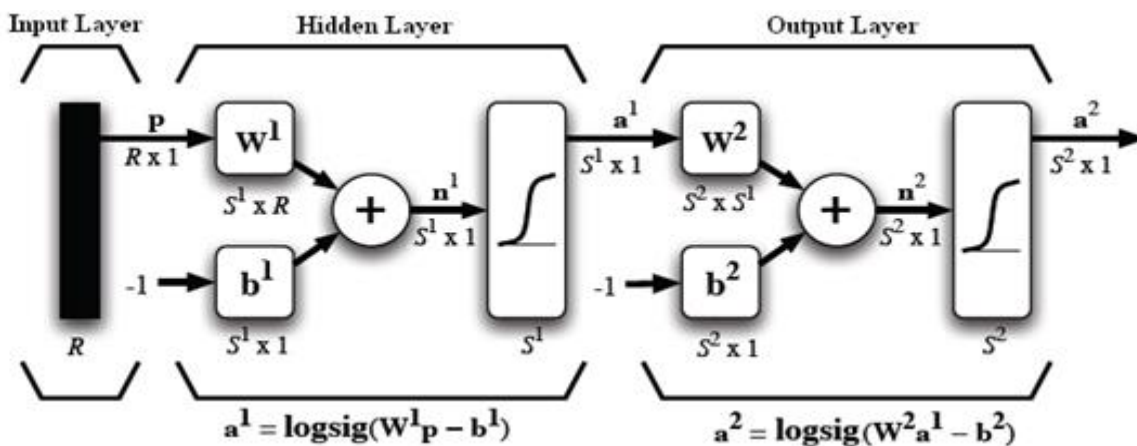
$$y = \Phi \left(\sum_{i=1}^n w_i x_i + b \right)$$

The threshold b is a neuron with a constant value of -1. By allowing the network to modify the weight associated with b , a dynamic threshold for when the perceptron activates is achieved. This is a very simple design, and its strength can be shown when several neurons

are combined and work together. The neurons are often organized in layers, where each layer takes input from the previous, applies weights and then signals to the next layer if appropriate. (See figure 2.8)

2.4.1 Feed-forward Neural Networks

For the feed-forward network type, units are arranged into layers. It consists of L processing levels, where the first level $l = 0$ is the input layer and the last level $l = L$ is the output layer. All communication with the outside world is done through these two layers. Usually intermediate layers of units which cannot be reached from the outside world are present. They are called hidden layers and the units within them are called hidden units. The hidden layer closest to the input is called the first hidden layer, the next layer second hidden layer and so on. The last hidden layer is therefore the one closest to the output layer. If it does not have any hidden layers, the network is called a simple perceptron otherwise it is called a multi-layer perceptron or X-layer perceptron where X is the number of hidden layers added by 1, representing the output layer, the input is usually not counted. Simple perceptron model attracted lots of attention when initially introduced until it was proven that the perceptron was only capable of solving linearly separable problems. In a feed-forward network the information passes from a lower layer forward to a higher. This means that units from a layer may only be connected to units in higher layers, no feedback or interconnections are allowed. A typical multi-layer feed-forward network *with* the notational conventions is shown in figure 2.8.



$$a^k = f^k(W^k a^{k-1} + b^k) \quad \text{For } k = 1 \dots M$$

Figure 2.8: Multilayer perceptron, *source*: Parizeau [7].

2.4.2 Training of Feed-Forward Network

Learning can be divided into two types: supervised and un-supervised, depending on how much information is provided to the network. The learning paradigm considered here is supervised learning, which means that the network is provided with examples of both input pattern and output pattern. The desired output pattern is called the target and is denoted $\mathbf{T} = \{ t_{L1}; t_{L2}; \dots ; t_{LNL} \}$. Learning involves two important components: a measurement of the network's performance during learning and an algorithm that improves the performance. The measurement is normally done by an error function that gives a qualitative value of the error. The most commonly used measurement is the Root Mean Square Error (RMSE) function, given by:

$$RMSE = \frac{1}{2} \sum_{p=0}^P \sum_{i=1}^M (T_i^p - V_{Li}^p)^2$$

where M is the number of output units and P is the total number of patterns.

There are several alternative error functions such as the relative entropy. A further description of this can be found in the references under the heading "*LEARNING AND ERROR FUNCTIONS*".

The MLP network is trained by altering the weights in each artificial neuron in such a way that the difference between the desired output value and the output we get is minimized. When the units in the network use transfer functions that are differentiable like sigmoid or Gaussian functions and an error function that is also differentiable it is possible to use a more mathematical approach. Training algorithms are often based on techniques for mathematical optimization, such as the principle of Gradient Descent, and consist of repeatedly presenting the network with patterns from the entire set of patterns, usually hundreds of times. After each presentation of an input pattern, the network adjusts its weights so that if the same pattern is presented again, it will be more likely to produce the correct output pattern. One full presentation of all patterns in the training set is normally called an epoch.

The sigmoid function, commonly used as an activation function is defined as follows:

$$p(c_j) = \frac{1}{1 + e^{-c_j}}$$

Where c_j is the combination function and e is Euler's number. Furthermore when computing the derivative of the sigmoid function, we observe that its derivative is computationally easy to perform.

$$\frac{dp(c_j)}{dc_j} = p(c_j)(1 - p(c_j))$$

Over-fitting

One problem with the ANN approach is over-fitting of the data, which happens when the classifier becomes too good at recognizing the training examples, at the expense of not being able to recognize a general input. This can be avoided by cross-validation, where the network is trained on one set of data, and then evaluated on a separate one.

When the error starts rising in the validation set, the network might be over-fitted. If previous networks are saved, the network can then be rolled back to the one which gave the smallest error. [8]

2.4.2.1 Backpropagation Algorithm

One of the most famous algorithms to train a neural network is the backpropagation algorithm and was introduced by Rumelhart, Werbos and Parker in the late 1980's. The backpropagation algorithm works as follows:

- i. Initialize the network by randomly assigning weights ranging from -0.5 to $+0.5$ for all neurons in each node.
- ii. Test the neural network with the input data set. After running through all the input data sets, an error term is calculated and weight changes occur at each node.
- iii. If a certain terminating condition is met, training stops or else step 2 is repeated.

2.5 Discrete Fourier Transform (DFT)

Discrete Fourier transform is a linear mapping that operates on N-dimensional vectors in the same way that the Fourier transform operates on functions in R. As the image is of finite size, we approximate the Fourier transform by a finite number of algebraic operations performed on a finite set of data. Given a discrete sequence, $x[n]=0,...n-1$

DFT:

$$X[k] = \text{DFT}\{X[n]\} = \sum_0^{N-1} x[n] \cdot e^{-i(2\pi kn/N)}$$

$k=0,1,..., N-1$

IDFT:

$$X[n] = \text{IDFT}\{X[n]\} = \frac{1}{N} \sum_0^{N-1} x[k] \cdot e^{i(2\pi kn/N)}$$

A two-dimensional FFT can be computed as two discrete Fourier transforms in one dimension.

$$F[u, v] = \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} I[m, n] \cdot e^{-i2\pi(\frac{um}{M} + \frac{vn}{N})}$$

$F(u, v)$ is a complex number and is represented by a magnitude and phase rather than a real and imaginary component $I(m, n)$ is a function representing the pixels of the image at coordinates (M, N) and has a real value [9].

$$\text{Magnitude}(F) = \sqrt{\text{real}(F)^2 + \text{imaginary}(F)^2}$$

$$\text{Phase}(F) = \tan^{-1} \left(\frac{\text{imaginary}(F)}{\text{real}(F)} \right)$$

The magnitude image of the Discrete Fourier Transform has the following properties (See fig.2.9). The u-axis represents the horizontal frequency component and runs from left to right through the center of the image. The v-axis represents the vertical frequency component and runs from bottom to top through the center of the image. The center of the image corresponds to the origin of the frequency coordinate system.

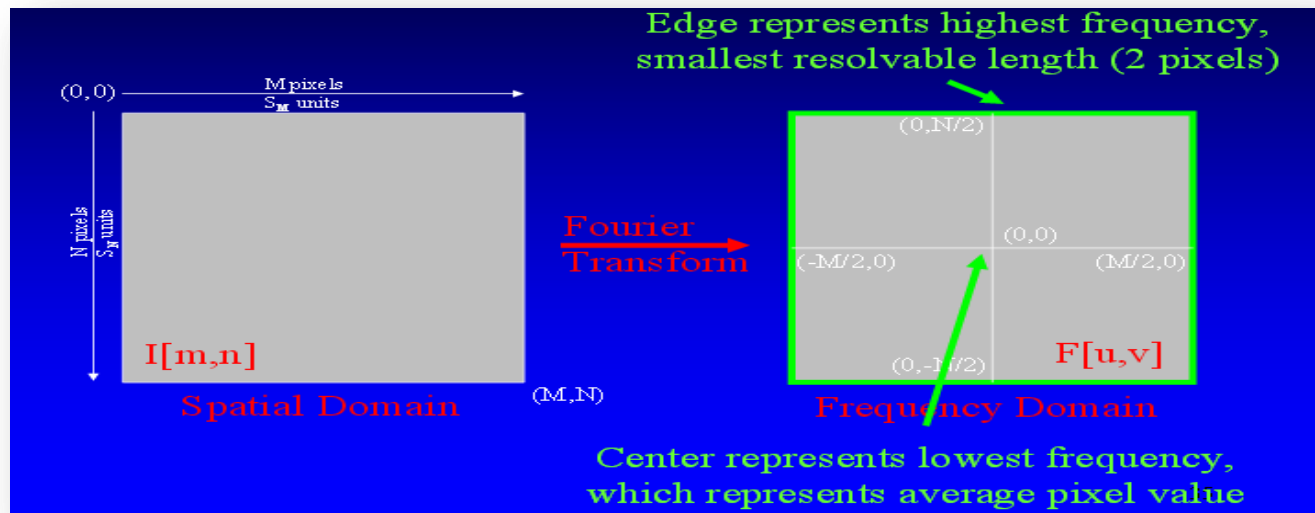


Figure 2.9: Magnitude Image of DFFT, source: Seul et al, *Practical Algorithms for Image Analysis*, 2000, p. 249, 262.

2.6 RLE encoding

Run-length encoding (RLE) is a very simple form of data compression in which *runs* of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. In a binary image, we can detect only two values: one and zero. In such a case, the RLE algorithm is even more compact, because only the lengths of the runs are needed.

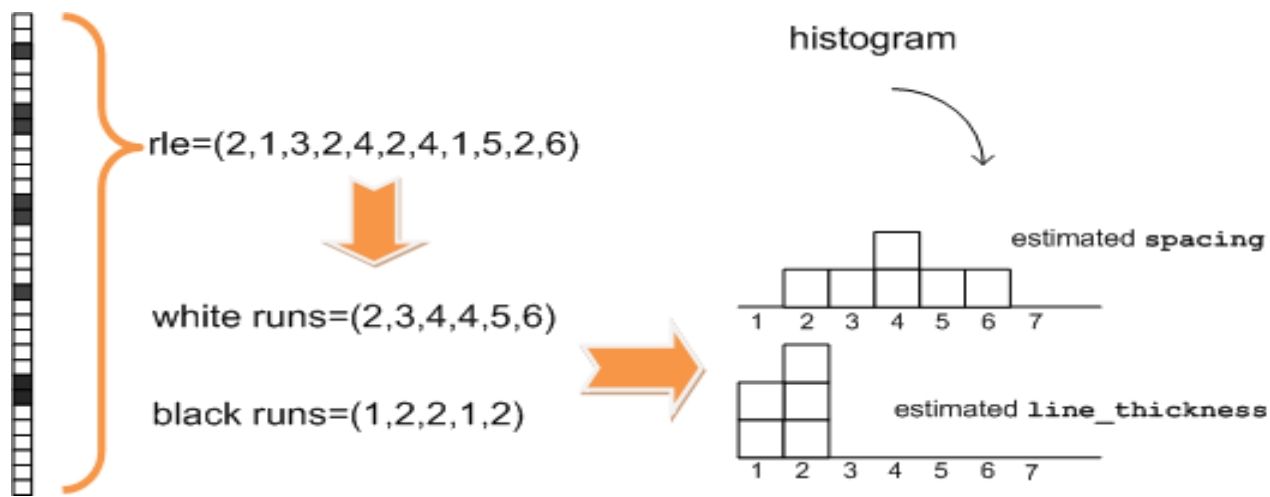


Figure 2.12: RLE encoding ,source: Pinto et al.[10]

As we can see from figure 2.12, the Run Length Encoding Algorithm will be used, in this project, to determine the thickness of the stave lines in the score and the distance between two stave lines, two important reference values in the staff line detection.

2.7 Projections

Image projections are one-dimensional representations of the image contents, usually computed parallel to the coordinate axis. In this case, the horizontal projection of an image $f(r,c)$, for all r and c is defined as: [11]

$$P_f(r) = \sum_c f(r, c)$$

And the vertical:

$$P_f(c) = \sum_r f(r, c)$$

The horizontal projection $P_f(r)$ is the summation of the pixel values in the image row r and has length N corresponding to the height of the image. On the other hand, vertical projection $P_f(c)$ of M length is the summation of all the values in the image column c . In the case of a binary image with $f(r,c)$, the projection contains the count of the foreground pixels in the corresponding image row or column. However, projection can be taken along any direction. For example, projection of $f(r,c)$ along a direction θ measured with respect to r -axis can be given by:

$$P_{f,\theta}(d) = \sum_r f(r, r, (\tan \theta) + d)$$

, for all d .

Projections in the direction of the coordinate axis are often utilized to quickly analyze the structure of an image and isolate its component parts.

2.8 Music File Formats

By the end of the 1970s, electronic musical devices were becoming increasingly common and affordable in North America, Europe and Japan. Thus the need for a system which would allow direct communication between equipment from different manufacturers rose. Sequential Circuits engineers and synthesizer designers Dave Smith and Chet Wood devised a universal synthesizer interface (MIDI), which was proposed at the Audio Engineering Society show in November 1981.^{[12]:4} MIDI (short for Musical Instrument Digital Interface) is a technical standard that describes a protocol, digital interface and connectors and defines a way for music to be represented in a digital format and can be played by computers. The SMF format can store sequences of time-stamped MIDI events (note on, note off, polyphonic pressure, etc.) for future playback or editing. It differs from an audio file in that it does not contain encoded sound, which means it can be significantly more compact. MIDI became a popular format for the interchange of music notation, although it had not been primarily designed for music notation purposes and it suffers certain drawbacks. [13] Standard MIDI files cannot, for example, distinguish between D# and Eb, because these notes are realized by pressing the same key on a piano keyboard. There is also no mechanism for explicitly representing rests within MIDI files. Several extensions to MIDI have been suggested to overcome the various limitations of MIDI for a general musical and notational information interchange. So far, none of the extensions have been widely accepted, therefore, the basic problems remain.

Another example of a binary file format is the Notation Interchange File Format (NIFF). Developed by several major music software companies, NIFF was envisioned to become a standard file format for music notation and interchange. It is extremely comprehensive and, thanks to the great attention to detail, it found use primarily in optical music recognition. However, it never really caught on and its support was dropped. [14]

One of the younger file formats is MusicXML, which was released in 2004. As the name suggests, MusicXML is based on the Extensible Markup Language (XML), which was designed to be easy to use, both human- and machine readable, and Internet-friendly. MusicXML is not open source but it is available under a royalty-free license from Recordare. [15]

This thesis only deals with playing of a scanned music score. Thus, MIDI format will be adopted.

2.9 Related Works

2.9.1 PC Platform

The investigation in the OMR field may began in the 1960's by Pruslin and Prerau but it was only in 1980's decade, when the equipment of digitalization became accessible that work in this area was expanded and the first commercial products appeared in the early 90s. Currently The most advanced recognition products are:

- Midiscan in Finale¹,
- Smartscore²,
- Sharp-eye³,
- Photoscore in Sibelius⁴,
- Capella-scan⁵ (Bellini, Bruno, & Nesi, 2007).

In 1993, Musitek Corporation released “music scanning and notation software” named as MIDISCAN which is known as the “world's first commercially available music-scanning software for Windows” [16]. In 1998 it was renamed to SmartScore (for Windows and Macintosh). Meanwhile, Graham Jones developed SHARPEYE “Music Reader” which is used to convert printed sheet of music notes into a MIDI file [17]. In September 1998, the first version for Windows of PhotoScore (now simply called Sibelius) was released by Neuratron. In 2003, Capella-scan was presented in Frankfurt Music Exhibition [18].¹

Remarks

Capella-Scan as well as SHARPEYE is released in windows version only. No current program deals with handwritten music. You need to be working with printed music. PhotoScore has a limited handwritten capability but that requires you writing the music in a particular PhotoScore-friendly style. Although advertisements for commercial OMR packages claim very good recognition rates, users of such systems agree that they are still too error-prone to be of much practical use.

¹ <http://www.finalemusic.com/>

² <http://www.neuratron.com/photoscore.htm>

³ <http://www.musitek.com/>

⁴ <http://www.music-scanning.com/>

⁵ <http://www.capella.de/us/>

2.9.2 Android Platform

The digital revolution of the 2010s introduced widespread use of mobile networked devices, mobile telephony and tablet computers. Tablet computers in particular introduced new opportunities for digital sheet music through their usable digital display [35-36]. Consequently, many iOS and Android apps were introduced in the sheet music market, offering digital a range of apps processing sheet music and on various mobile platforms.

This section provides a description of two commercial OMR applications on App store. Those apps were tested and a description of how they performed those apps was tested and a description of how they performed in comparison to one another is provided. In section 7.1, we will thoroughly discuss how OMR systems can be evaluated. Therefore this section only provides a user's perspective of how well each application performed.

SnapNPlay¹ :

Published on April 29th, 2013 by David Johnson. SnapNPlay, allows you to take/scan a picture of a piece of sheet music, and play it back like a recording. Next, you are not taking a simple picture of the whole piece of sheet music. You are only snapping one staff at a time. Even so, you are not snapping the whole staff, just the notes, not the clef symbol and key signature. You enter the key signature, manually. You repeat this for each staff you want to play back. You can organize your individual snaps into folders. There is no way to play through a song with multiple staves, but you can swipe from one staff to another within a folder. SnapNPlay has an attractive feature which enables the user to modify the recognized score. There is an editing module that enables correction of any misrecognized notes and rests. You can also delete and add notes as you see fit. Lately, an iOS version has been released. The cost of this app is \$4 and it is rated as 3.7 out of 5.

iSeeNotes²:

Published in 2014 by Gear Up AB. iSeeNotes uses device's camera to scan musical notes, automatically recognizes the notes and plays back the music. This app reads multiple staves at a time and can recognize bass and treble clef as well. Lately, an iOS version has been released. The cost of this app is \$4 and it is rated as 3.2 out of 5.

Remarks

For both of the apps mentioned there are, admittedly, a few problems with execution. As with any scanning and OCR app, the key is getting a great image and this depends basically on the device's camera and the lighting, the possibility of user to shake while snapping and the distance between object and camera. And most importantly none of those can guaranty much accurate results. As mentioned in the description of the iSeeNotes app by the developer: "Music recognition is a difficult problem. This app does not support the complete set of notation, for instance whole notes and the alto clef are not recognized. The app may have trouble recognizing the notes and symbols correctly - do not expect perfect results!"

2.9.3 OpenOMR

OpenOMR [19] is a 2006 open source project, submitted by Arnaud F. Desaedeleer in fulfillment of MSc Degree in Advanced Computing of the University of London. OpenOMR is an optical music recognition (OMR) tool written in Java for printed music scores. It allows a user to scan a printed music partition and play it through the computer speakers. A segmentation based approach was used for the design of this project and was based on the Music Sheet Reader [20] an OMR system by Yong Li at the University of Birmingham project.

It is being published as free software under the terms of the GNU General Public License (GPL). OpenOMR is described as pre-alpha project which means it is at a very early experimental stage and will be very unstable.

¹ <https://play.google.com/store/apps/details?id=com.snapnplay.android>

² www.iseenotes.com

Development Tools

3.1 Java

Java is a programming language first released by Sun Microsystems back in 1995. It can be found on many different types of devices from smartphones, to mainframe computers. It is a general-purpose computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. You can use it on your desktop PC and even on the Raspberry Pi. Java doesn't compile to native processor code but rather it relies on a "virtual machine" which understands an intermediate format called Java bytecode. Each platform that runs Java needs a virtual machine (VM) implementation. The job of these virtual machines is to interpret the bytecode, which is really just a set of instructions similar to the machine code found in CPUs, and executes the program on the processor. The VMs use a variety of technologies including just-in-time compilation (JIT) and ahead-of-time compilation (AOT) to speed up the processes. The Oracle implementation is packaged into two different distributions: The Java Runtime Environment (JRE) which contains the parts of the Java SE platform required to run Java programs and is intended for end users, and the Java Development Kit (JDK), which is intended for software developers and includes development tools such as the Java compiler, Javadoc (documentation generator), Jar (Java archiver), and a debugger.

3.2 Java Development Kit

The Java Development Kit (JDK) is an implementation of either one of the Java SE, Java EE or Java ME platforms released by Oracle Corporation in the form of a binary product aimed at Java developers on Solaris, Linux, Mac OS X or Windows. The JDK includes a private JVM and a few other resources Machine and all of the class libraries present in the production environment, as well as additional libraries only useful to developers, such as

the internationalization libraries and the IDL libraries.to finish the recipe to a Java Application. Since the introduction of the Java platform, it has been by far the most widely used Software Development Kit (SDK). The JDK also comes with a complete Java Runtime Environment, usually called a *private* runtime, due to the fact that it is separated from the "regular" JRE and has extra contents. It consists of a Java Virtual

3.3 Eclipse IDE

Eclipse is an integrated development environment (IDE). It contains a base workspace and an extensible plug-in system for customizing the environment. Eclipse is written mostly in Java and its primary use is for developing Java applications, but it may also be used to develop applications in other programming languages through the use of plugins. Functions such as maintaining publications, changes to the code sharing between various members of the development team, virtual servers for testing correctness operating web applications and others provided in the environment itself.

3.4 Eclipse ADT

Android Development Tools (ADT) is a Google-provided plugin for the Eclipse IDE that is designed to provide an integrated environment in which to build Android applications. ADT extends the capabilities of Eclipse to let developers set up new Android projects, create an application UI, add packages based on the Android Framework API, debug their applications using the Android SDK tools, and export signed (or unsigned) .apk files in order to distribute their applications. It was the official IDE for Android but was replaced by Android Studio (based on IntelliJ IDEA Community Edition).

3.5 Android Studio

Android Studio is the official integrated development environment IDE for Android application development. It provides a number of improvements to assist you in debugging and improving the performance of your code, including an improved virtual device management, inline debugging, and performance analysis tools. Android Studio allows you to track memory allocation as it monitors memory use. Tracking memory allocation allows you to monitor where objects are being allocated when you perform certain actions. Knowing these allocations enables you to adjust the method calls related to those actions to optimize your app's performance and memory use. Android Studio allows you to work with layouts in both a Design View and a Text View. Allows you to open it and create new virtual devices for running your app in the emulator. The AVD Manager

comes with emulators for Nexus 6 and Nexus 9 devices and also supports creating custom Android device skins based on specific emulator properties and assigning those skins to hardware profiles. Finally it gives you the opportunity to choose an external device, usb connected to the computer, as a virtual machine, a feature really useful.

3.6 Java and Android API

While most Android applications are written in Java-like language, there are many differences between the Java API and the Android API, and Android does not use a Java Virtual Machine but two other ones called either Dalvik or Android Runtime (ART). There is no Java Virtual Machine in the Android platform. Java bytecode is not executed. Instead Java classes are compiled into a proprietary bytecode format and run on Dalvik, a specialized virtual machine (VM) designed specifically for Android. Unlike Java VMs, which are stack machines, the Dalvik VM is a register-based architecture.

Dalvik has some specific characteristics that differentiate it from other standard VMs:²³

- The VM was designed to use less space.
- The constant pool has been modified to use only 32-bit indexes to simplify the interpreter.
- Standard Java bytecode executes 8-bit stack instructions. Local variables must be copied to or from the operand stack by separate instructions. Dalvik instead uses its own 16-bit instruction set that works directly on local variables. The local variable is commonly picked by a 4-bit 'virtual register' field.

Because the bytecode loaded by the Dalvik virtual machine is not Java bytecode and due to the specific way Dalvik loads classes, it is not possible to load libraries packages as jar files. A different procedure must be used to load Android libraries, in which the content of the underlying dex file must be copied in the application private internal storage area before it is loaded.²⁴ Developing an Android app is more than just Java, you need to understand how the Android UI is constructed (using XML), and how to access the different Android subsystems.

Modeling and System Architecture

This chapter analyzes in depth the modeling, android system and OMR system architecture, as well as the rules followed to design and implement our system. The adopted Optical Music Recognition framework will be discussed along with the various stages and the respective algorithms will be explained in detail. Similar to other implementations, the framework used in the prototype follows the same unidirectional pipeline that has been presented in section 2.3. A detailed description of the Java package structure of the project is given with the presented UML at the respective stages for better understanding of system's architecture.

4.1 System architecture

Composing the individual parts, the complete system architecture is created. The parts are communicating and exchanging data in accordance with the logic of our application as shown in the Figure 4.1. A more thorough presentation about the interaction between these components and sub-modules is in the following sections.

The user launches the AOMR application and is then asked to decide on an image to go into the recognize process. By the time of selection the image goes through the Pre-Processor for image manipulation. The processed image is then input to the Recognizer module where all music symbols are located. These symbols are then passed into a neural network as this is defined by the features extracted from its local host component. Finally musical rules are integrated to the classified objects. Finally, pursuant these musical object information a Midi file is created and comes as output of the application.

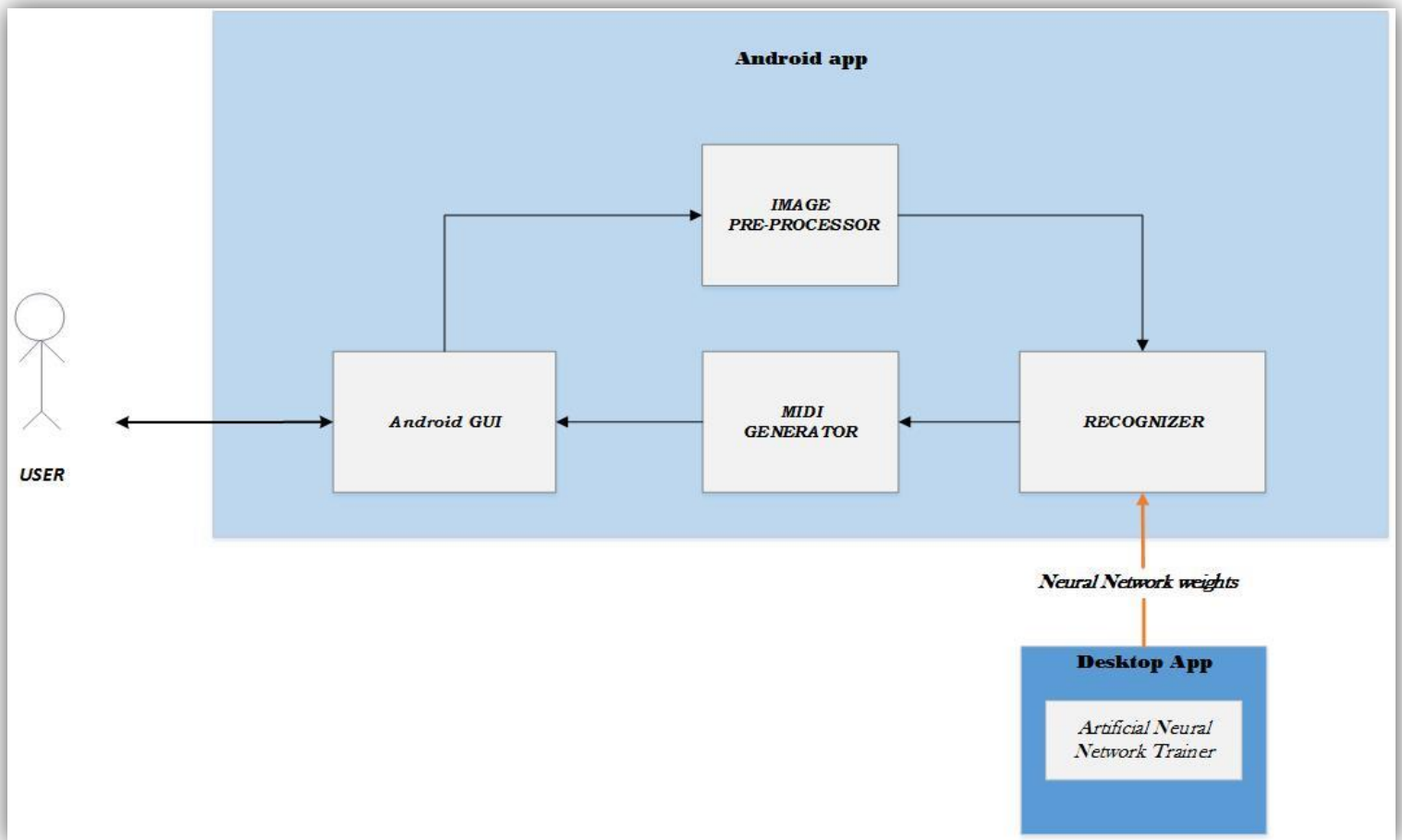


Figure 4.1.: System architecture diagram

The Recognizer sub-module gets the pre-processed image and the result given is input to MIDI generator, as shown in the above diagram. This sub-module is responsible for the process of music recognition and contains the corresponding individual functions as shown in figure 4.2.

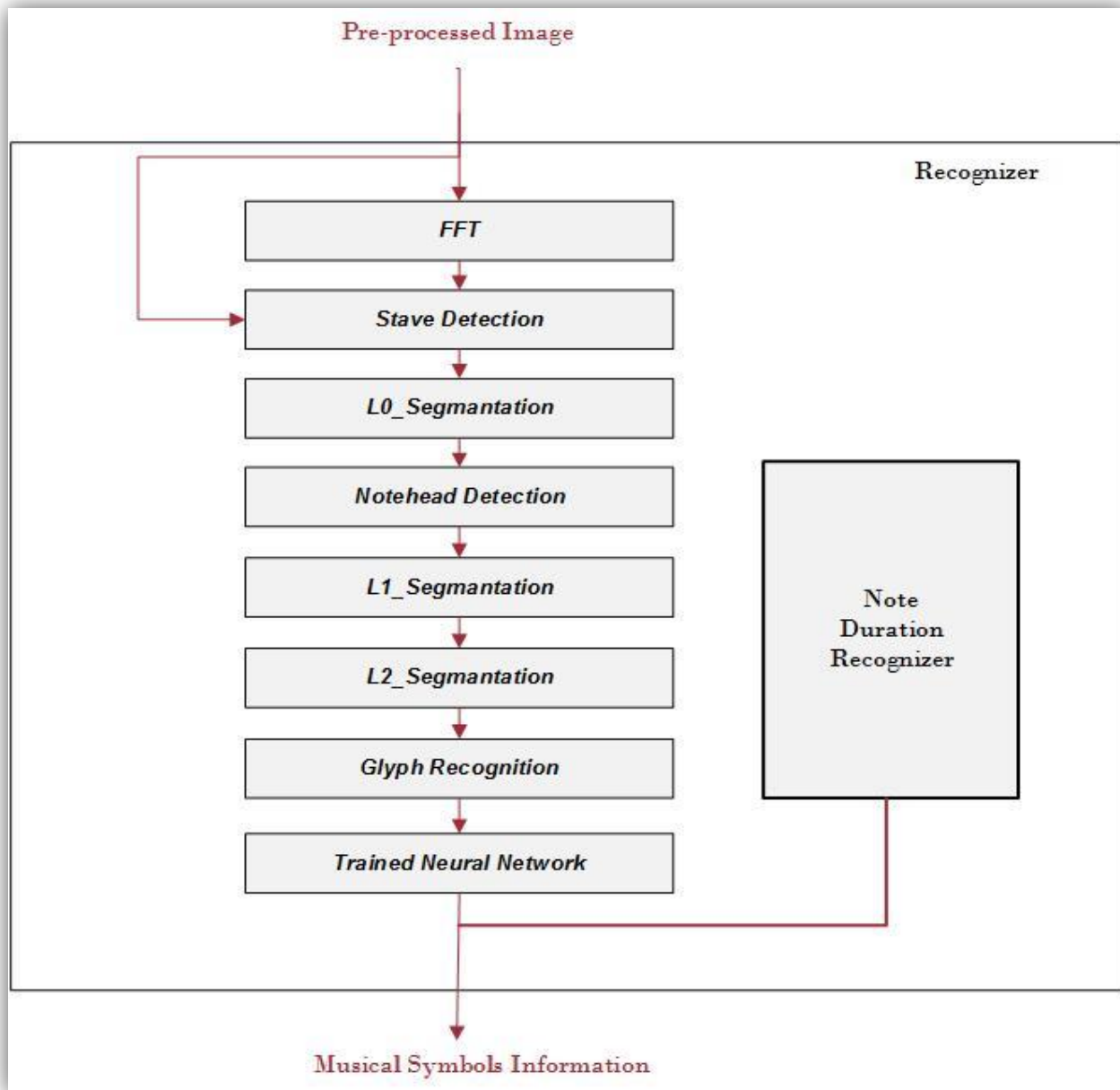


Figure 4.2.: Recognizer architecture diagram

4.2 System Modeling

4.2.1 Image Selection Process

Commercial applications that deal with image manipulation distinguish two cases: capturing photos through the in-built camera or selecting a locally stored image from Google Photos application and the long-known Gallery application. The issues taken into consideration are:

- The lighting environment and the photographer's ability to capture photos greatly affect the quality of the image. Even two images taken under the same environment can be different.
 - Pictures need to be taken at a minimum resolution of 300 dpi. This seems to be satisfactory for standard piano music or instrumental parts that have eight to ten staves per page. This resolution value, however, is not fine enough for orchestral scores or miniature scores; a resolution of 600 dpi is needed. In terms of use of our application an image resolution of 300dpi seems fine enough. In order to succeed this and given the size of a common notation sheet (DIN A4, 8.27x 11.69”), we would need a camera with a capability to work at about 9 megapixels.
- Thus we strongly suggest user to generate a picture under the terms of a scanner or software and then import this in device’s memory in order to obtain satisfactory results.

4.2.2 OMR ALGORITHMS & FLOW PROCESS

Similar to other implementations, the framework used in the prototype follows the same unidirectional pipeline that has been presented in the previous chapter. In the following, the various stages will be discussed and the adopted techniques will be explained in detail.

4.2.2.1 Image pre-processing

In digital image processing, as in all signal processing systems, different techniques can be applied to the input, making it ready for the detection steps. The motivation is to obtain a

more robust and efficient recognition process. Enhancement, binarization, morphological operations and de-skewing are the techniques applied in this stage.

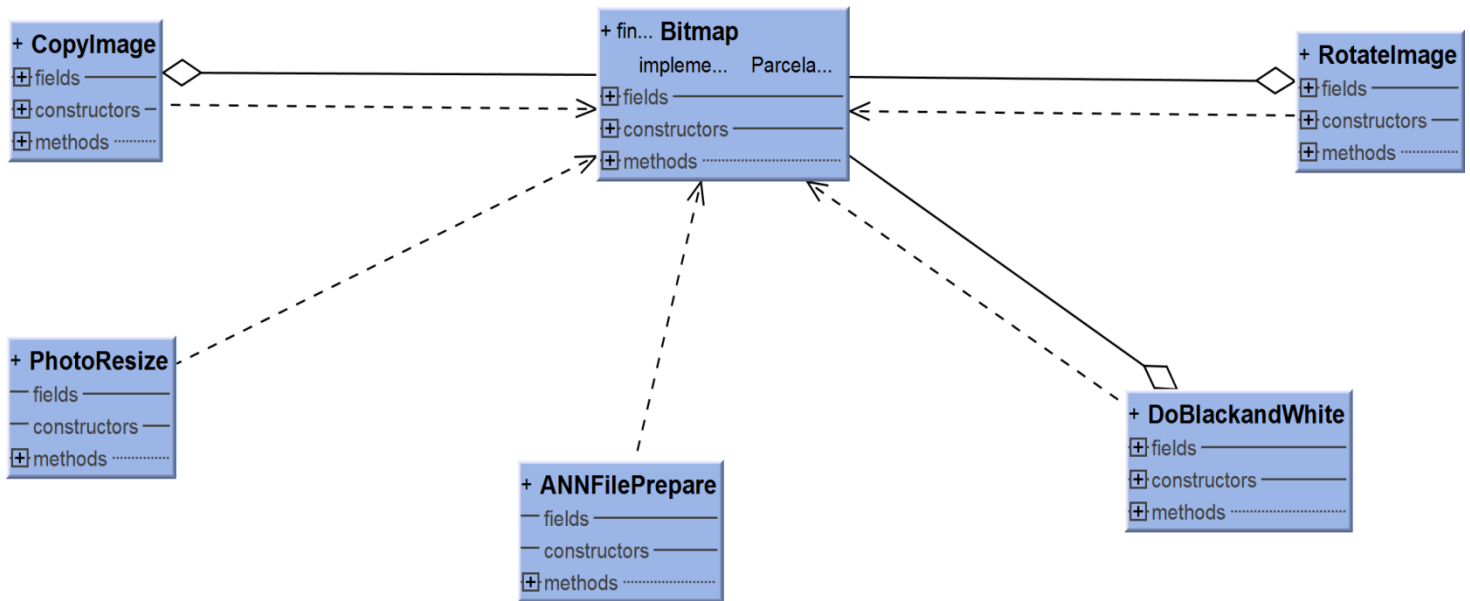


Figure 4.3: Image pre-processing – UML diagram

Binarization

The image is first divided into chunks whereby the number of eight rows and columns has proven best for the image dataset. Iterating through the pixels of each image partition, the number of foreground and background pixels are counted, based on a threshold value T which has the initial value of 0. Then, the mean value both of all foreground and background pixels is computed. After each iteration, T is assigned the average of the two means, resulting in a maximization of variance of the two classes once the algorithm has reached its end. Finally, the threshold T is applied to every pixel within the current window. This procedure is a fair compromise, overcoming deficits of the global thresholding but remaining computationally efficient.

Skew-Correction

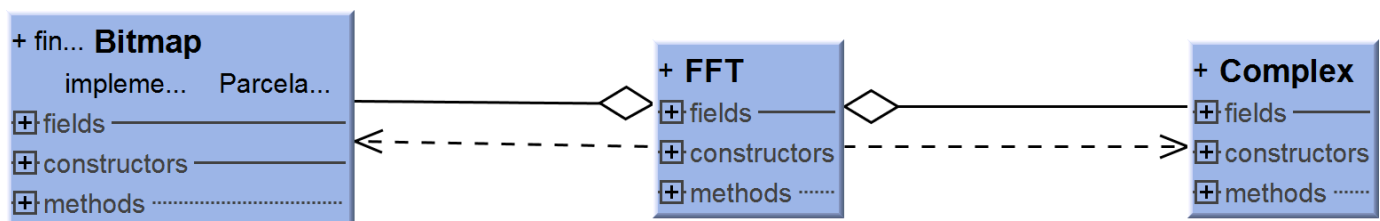


Figure 4.4.: FFT Implementation – UML diagram

The correction of distortion caused by scanners or digital cameras has received considerable attention in the OMR literature. Distortion correction is generally combined with staff line detection, which is usually performed by either projection or vertical scan lines, which means it is elementary to properly align the image in order to obtain satisfactory results. Furthermore, prior work generally assumes that the deformation can be corrected by rotation, partly because music has traditionally been scanned on a flat-bed layer where rotation is the most likely error. In order to compensate for the rotation of the mobile phone relative to the notation sheet, we apply the DFFT algorithm as discussed in section 2.5.

As the images we are dealing with are digital images, the signals are discrete. Then, the relevant Fourier transform is the discrete Fourier transform. There exists a fast method to approximate the DFT and this is known as the Fast Fourier Transform (FFT). While the Discrete Fourier Transform is computed in $O(N^2)$ time, the FFT can be computed in $O(N \log N)$ time. Therefore an algorithm to compute the FFT was used in order to improve the efficiency of this component.

This package contains the class which calculates the Fast Fourier Transform of a Bitmap. The detected rotation angle of the image is then input to the pre-processing package so as to rotate at the detected angle and then replace it.



Figure 4.5: “Song for Guy” by Elton John, Skewed

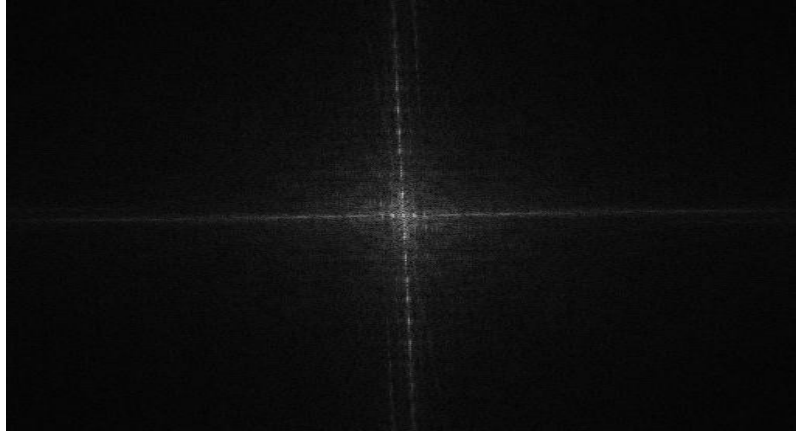


Figure 4.6: FFT of figure 2.10

Figure 4.6 shows the graph of the magnitude components when the Fourier Transform of the image in figure 4.5 is taken. When looking at figure 4.6 we notice that there exists an almost vertical line tilted by a few degrees to the right. From this, the angle by which the staves are rotated can be calculated by the following algorithm[Rot.A]:

1. The Fourier Transform is symmetric to the origin. That is: $F(-x, -y) = F(x, y)$
2. The most distinct feature in the Fourier Transform of a music score is the line consisting of bright white spots at an angle to the vertical axis. The Fourier components on this line arise from the rotated stave lines.
3. The angle between the strong components in the Fourier transform can be determined by locating two strong components, for example the brightest white spots, which are the furthest away from one another. The angle can thus be calculated as follows given two coordinates (x_1, y_1) and (x_2, y_2) :

$$\theta = \sin^{-1} \frac{y_1 - y_2}{x_1 - x_2}$$

4.2.2.2 Musical Object Location

The OMR system developed in this section is based on the segmentation architecture as this was implemented in OpenOMR project discussed in section 2.9.3 and we converted it to Android API. In the UML of this module only the most characteristic classes were implemented to facilitate its presentation.

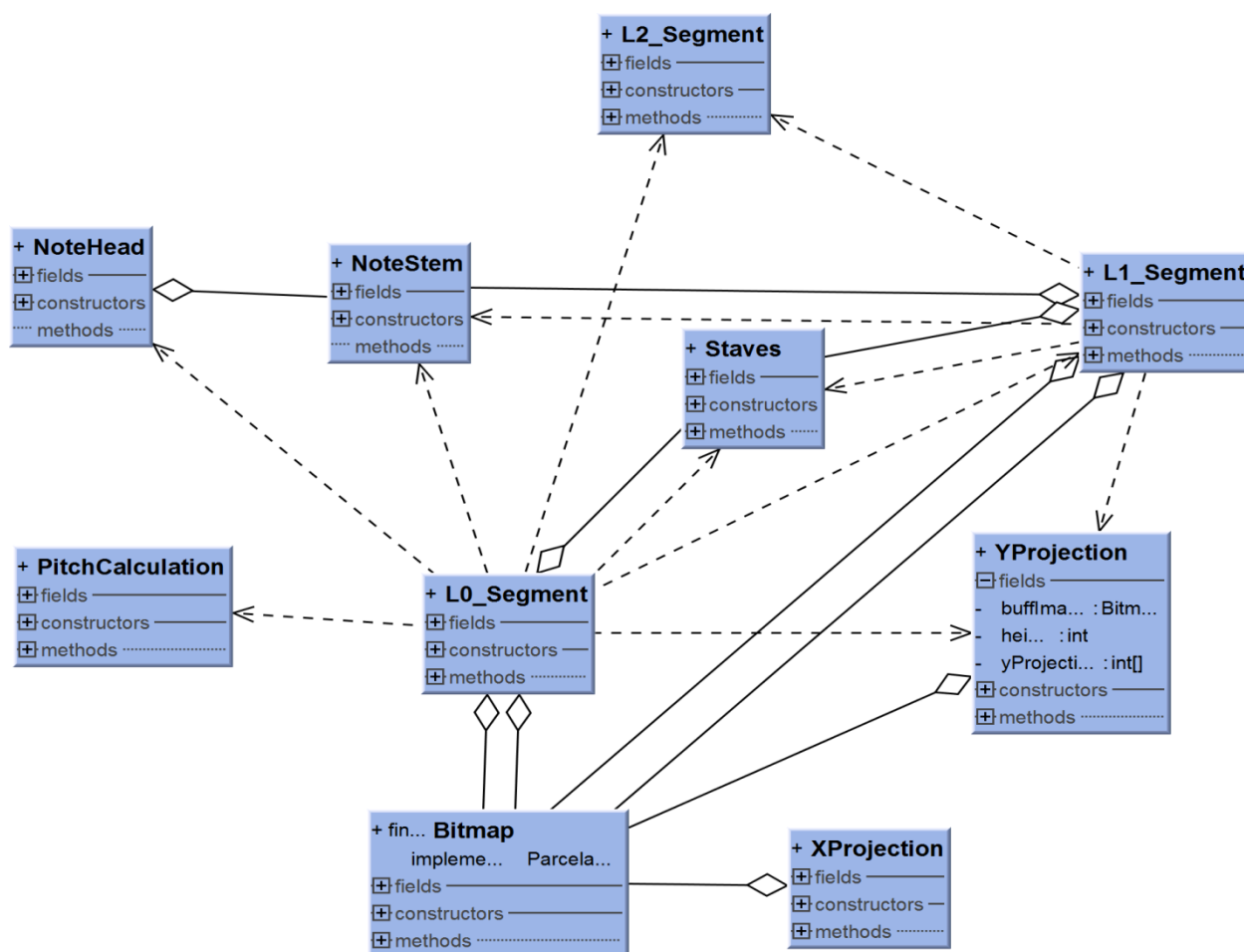


Figure 4.7.: Musical object Detection – UML diagram

Stave Detection

Further process of the image is based on two reference values: staff space height (d) and staff line height (n), which are determined after RLE encoding applied (See section 2.7). While the first measure describes the distance in pixels between each of the four staff lines per stave, the latter stands for the thickness of staff Lines. The importance of these values becomes self-evident in the following chapters. Traversing through the elements,

black and white runs are counted and the most-common ones are stored as staff space height and staff line height, respectively. Based on this information, we can proceed to the stave detection, knowing that the height of one stave can be approximated by:

$$\text{Height} = 5d + 4n$$

Given a valid result (staff height > 0 and staff space > staff height) from the RLE measuring process, we are now in the position to detect staff lines, a critical task in music recognition, when it comes to isolating musical symbols. Since staves and staff lines are supposed to be horizontally aligned after the de-skewing process, the position of staff lines can easily be derived from the local maxima of the y-projections. We obtain a one-dimensional array whose contents are the summation of all black pixels along the y-axis. Then we “run” the array, looking for five equidistant peaks in range $1/3$ to $3/2$ of the value found of the distance between two stave lines each of approximately the same value. The peaks therefore do not have to have a value above a certain threshold but rather must be within a certain percentage of their neighboring peaks. This therefore enables staves which may not take up the entire width of the image to be found.

Hierarchical Segmentation

The extraction of music symbols is the operation following the staff line detection. The segmentation process consists of locating and isolating the musical objects to identify them. Groups of symbols are defined as being groups of notes that are attached to one another with beams. The main purpose of this step is to reduce the search space of the image in order to reduce the processing time required for the subsequent phases.

Image Segmentation (Level 0)

The method used to locate groups of symbols relies on the x-projection. The x-projection for each stave is calculated and stored in an array for post processing. A minimum threshold T_0 is defined :

$$T_0 = 5 * (\text{max_staveline_height})$$

to indicate segments that comprise music symbols than just empty staves. We choose maximum stave line height here as we want to try to set a minimum possible threshold. The array containing the x-projection is then scanned and as soon as we find values above the minimum threshold, we start a counter. The process then continues until a value

below the minimum threshold is found. The counter then defines the width of the new segment that was found and its coordinates are stored in a data structure.

Note Head Detection

The purpose of this step is to detect and label any segments containing filled note heads. It does so by analyzing the y-projection of the Level 0 segments. In the particular example shown below, we notice that the Level 0 segment contains a total of six note heads. Similarly, different musical notes are detected. At this stage, the stems of the notes are also located. All filled note heads have a stem attached to them. We are concerned with finding the pitch and duration for each note. The pitch for each note is relatively simple to find as we already know their x and y coordinates from the note head detection module. We now need to determine the duration of the note and this is done based on which side of the note the stem is located. If the stem is located on the left of the note, we need to look at the glyph directly below the note and if the stem is on the right, we need to look at the segment to the top right of the node. This follows the way that music scores are written in general. All note heads which were found during this segmentation process along with the position of the stem (left or right) are stored in a data structure which is contained within the level 0 segment data structure.



Figure 4.8: Level 0 segment

Segmentation (Level 1)

Level 0 segments that after To thresholding was applied were detected to comprise a note head will be further processed by Level 1 segmentation module. In this module Level 0 segments symbols are vertically separated. Level 0 segments that have been Level 1 segments have the height of the respective level 0 segments and the width of the note head or the distance from the start until the start of the first note head, if Level 0 does not start with a note head.

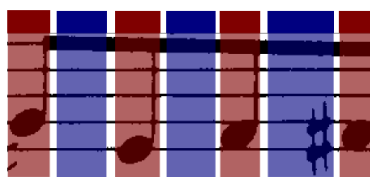


Figure 4.9: Level 1 segment

Segmentation (Level 2)

The goal of this module is to separate note heads from other symbols. Level 2 segmentation module horizontally separates each of the level 1 segments. A level 1 segment is input into the level 2 segmentation module and the first filter is applied to the y projection which removes the identified stems. The y-projection is then passed through a second filter which removes the stave lines. Having applied these two filters, we can now search for glyphs in the projection. Any sequence of non-zero values in the projection which are separated by less than the $\frac{1}{3}$ the value found of the distance between two stave lines consecutive 0 values in the projection are considered to be one glyph. The relevant coordinates of each glyph found are then stored for processing by the neural network.

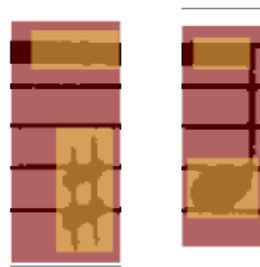


Figure 4.10: Level 2 segment

4.2.2.3 Musical feature Classification

Neural Network

The importance of neural networks is that they offer a very powerful and very general framework for representing non-linear mappings from several input variables to several output variables, where the form of the mapping is governed by a number of adjustable parameters. The advantages of using Artificial Neural Networks undergo by [21]:

- Are extremely powerful computational devices
- Massive parallelism makes them very efficient
- They can learn and generalize from training data – so there is no need for enormous feats of programming
- They are particularly fault and noise tolerant, so they can cope with situations where normal symbolic systems would have difficulty.

The neural network software implementation is based on JOONE [Joone], as it appears to be the best offering among the freely available neural network development tools for the Java language. The Java Object Oriented Neural Network (JOONE) is an open source project that offers a highly adaptable neural network for Java programmers. The JOONE project source code is covered by a Lesser GNU Public License (LGPL). JOONE can be downloaded from: <http://joone.sourceforge.net/>.

It consists of a component-based architecture based on linkable components that can be extended to build new learning algorithms and neural networks architectures. Components are plug-in code modules that are linked to produce an information flow. New components can be added and reused. Beyond simulation, Joone also has to some extent multi-platform deployment capabilities. Joone also has a GUI Editor to graphically create and test any neural network, and a distributed training environment that allows for neural networks to be trained on multiple remote machines. JOONE framework implements Multi-Layer Perceptrons (MLP) Backpropagation Neural Network's architecture. System uses the simple back propagation neural network engine from as a base with a modification of the stopping criterion during the training, where we use the sum of squared error to determine. MLP network will output a percent confidence [22] for each output node when interrogated, which is one of the main reasons for choosing the MLP network for the design of our application. If the percent confidence of the classified symbol is below a certain percentage, it is rejected (See fig.4.11). As we saw in section 2.4.2, the MLP network can be trained in supervised/unsupervised mode.

A detailed discuss about the training environment of the network will be discussed in the following chapter.

Our proposed system for android, supposes that the neural network's training is prosecuted off-line on a PC using a separate desktop application and saving the respective weights/Symbol Confidence of neural network arcs as a data file once the training is finished. Every time the AOMR application is loaded, it looks for this file in order to restore the saved state of the neural network. The class which is responsible for this is ANNInterrogator. The implementation of neural network will be discussed more thoroughly in the next chapter.

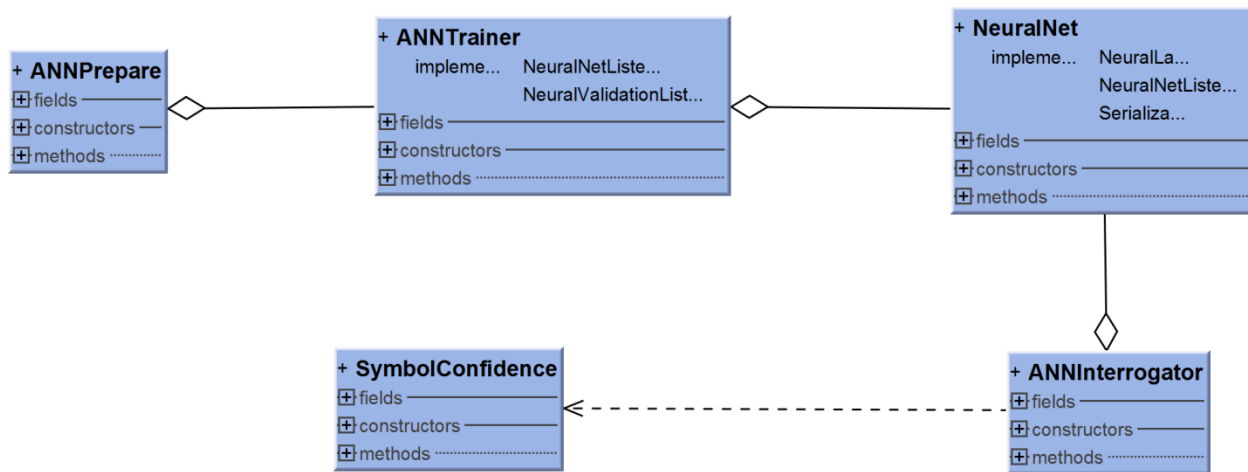


Figure 4.11: Neural Network – UML diagram

Normalising Image Segments

Inputs nodes to neural network are fixed and so each image segment must be normalized as the image segments produced in the Level 2 segmentation phase will have different sizes. One pixel per input node is used. In order to input the image into the neural network, we can convert the 8 by 16 image into a single array by concatenating each column of the image one after another. HSV (Hue, Saturation and Value) colour model is used to represent each pixel of this image. The Value of the HSV model represents the brightness of the pixel and this is what is used as input into each node of the neural network.

The images are read from the 'neuralNetwork' directory and the `Bitmap.createScaledBitmap()` method of the Java Bitmap class is used to scale each image in accordance to the demands of the neural network.

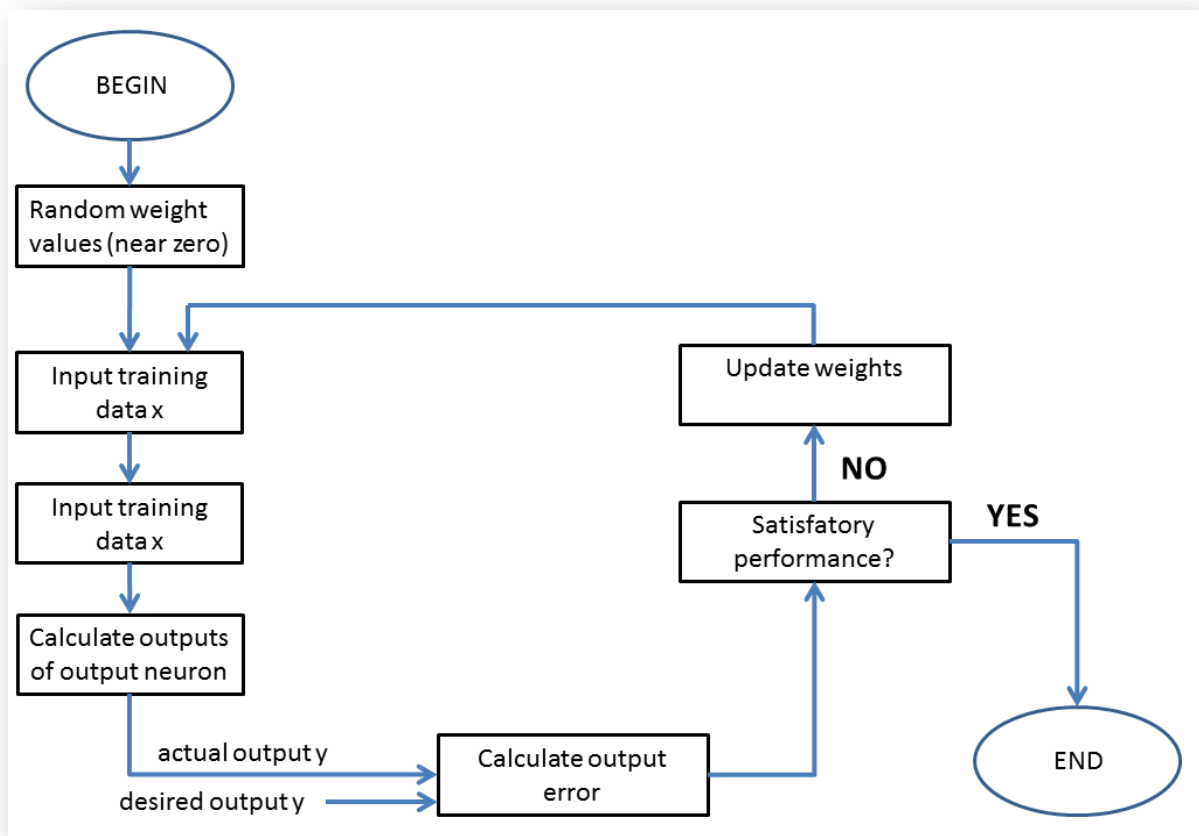


Figure 4.12:Backpropagation training – Flow chart

The neural network software implementation is based on JOONE [23], as it appears to be the best offering among the freely available neural network development tools for the Java language. The Java Object Oriented Neural Network (JOONE) is an open source project that offers a highly adaptable neural network for Java programmers. The JOONE project source code is covered by a Lesser GNU Public License (LGPL). JOONE can be downloaded from: <http://joone.sourceforge.net/>.

It consists of a component-based architecture based on linkable components that can be extended to build new learning algorithms and neural networks architectures. Components are plug-in code modules that are linked to produce an information flow. New components can be added and reused. Beyond simulation, Joone also has to some extent multi-platform deployment capabilities. Joone also has a GUI Editor to graphically create and test any neural network, and a distributed training environment that allows for

neural networks to be trained on multiple remote machines. JOONE framework implements Multi-Layer Perceptrons (MLP) Backpropagation Neural Network's architecture. System uses the simple back propagation neural network engine from as a base with a modification of the stopping criterion during the training, where we use the sum of squared error to determine. A detailed discuss about the training environment of the network will be discussed in the following chapter.

In our case, the training data was composed of musical glyphs coming from different music scores. Those were collected by cutting out different glyphs and saving them into separate files. In chapter 6, we will make a more detailed presentation of the testing environment of the neural network.

4.2.2.4 Audio Synthesis

The MIDI Protocol

Being widely used on various platforms and applications, the multi-purpose Musical Instrument Digital Interface (MIDI) protocol has proven to be a viable route for storing and interpreting musical information for this prototype. As opposed to a plain audio format, MIDI enables the developer/user to edit notational information at a later point. Moreover, the amount of data is a magnitude lower than even a compressed audio file.

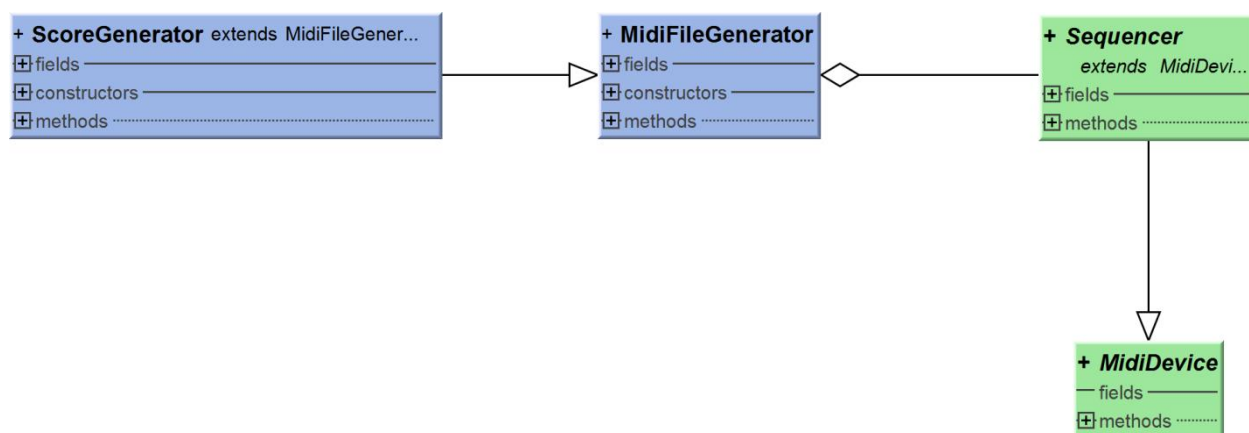


Figure 4.13: Midi generation – UML diagram

The goal of this module is to generate an audio file which can be played through the device's speakers. In essence, this module takes all the classified symbols (level 2 segments), interprets their musical semantics and constructs a music file representation of them based on some rules. We are concerned with finding the pitch and duration for each note. The pitch for each note is relatively simple to find as we already know their x and y coordinates from the note head detection module. We now need to determine the duration of the note and this is done based on which side of the note the stem is located. If the stem is located on the left of the note, we need to look at the glyph directly below the note and if the stem is on the right, we need to look at the segment to the top right of the node. This follows the way that music scores are written in general.

4.3 Application Design

4.3.1 Functional Requirements

Basic functional requirements of AOMR Android application:

- Device with an android operating System. The prototype has been targeted for Android version 4.4 and later and likewise, testing was carried out on several other handsets that revert to recent versions (API levels 14-21).
- A notation sheet image stored in the "Gallery". (see 4.2.1)
- Minimum memory size available: 3,90 MB

4.3.2 Supporting Different Devices

The Android operating system runs on a wide range of devices all around the world, which may come in many shapes and sizes. An Android app needs to adapt to various device configurations in order to be as successful as possible. Some of the important variations that you should consider include different screen sizes and versions of the Android platform.

- **Supporting Multiple Screens**

Although the Android system provides a consistent development environment across devices and performs scaling and resizing to make an application work on different screens, different APIs should be implemented for adjustment on different screen sizes and densities with the aim to create an application that displays properly and provides an optimized user experience on all supported screen configurations, using a single .apk file.

Android divides the range of actual screen sizes and densities into:

- a set of four generalized sizes: *small*, *normal*, *large*, and *extra-large*
- A set of six generalized densities:
 - *ldpi* (low) ~120dpi
 - *mdpi* (medium) ~160dpi
 - *hdpi* (high) ~240dpi
 - *xhdpi* (extra-high) ~320dpi
 - *xxhdpi* (extra-extra-high) ~480dpi
 - *xxxhdpi* (extra-extra-extra-high) ~640dpi

In our application the respective configuration qualifiers³ were implemented that support all the range of screen sizes and, most importantly all the range of densities, to ensure that the imported image is at the optimum resolution, as our app is based on image recognition algorithms.

³ A configuration qualifier is a string that you can append to a resource directory in your Android project and specifies the configuration for which the resources inside are designed.

- **Supporting Different Platform Versions**

Continuous evolution of the Android platform is an advantage and a challenge for the programmer who must follow developments and uses new possibilities offered by the each version without excluding support in earlier versions of Android. This is a significant problem because some new features are not supported by older versions and thus make it impossible to use in an earlier version of Android.

The AndroidManifest.xml file (See section 5.2.1) describes details about the developed app and identifies which versions of Android it supports and identifies the lowest API level with which an app is compatible and the highest API level against which app is designed and tested.

4.3.3 Design principles and application's interface

The design has become one of the key issues in developing Android applications. Nevertheless, apart from properly implementing the core functionality, the design is the second most important issue in developing an application. With the rapidly growing number of available apps, users expect them to be at the same time intuitive and elegant. It ultimately means that developers should constantly find a balance between making their interface clear but rich and making it simple.

AOMR application consists of three different views, designed to exhibit consistent user interface conventions, which are based on some design pre-defined rules:

- A few buttons and options on each screen.
- Ease and simplicity to navigate between screens.
- Distinct data and information

4.4 AOMR

The aim of this project is to design an optical music recognition (OMR) system a system which will have a more dynamic access to user's everyday life. This would be achieved by implementing OMR on Android devices. Focusing on these features the name of the application AOMR came up after a composition of the word Android and the acronym for optical music recognition, OMR.

Implementation

This chapter discusses several considerations taken into account when the AOMR system was implemented. In the respective fields a representation of AOMR launched application will follow. A technical description of the various components and how they interact with each other is then provided. Notably justifications as to why the Java programming language was used are provided. Great attention will be to the major implementation issues faced and the way we tackled them. Finally an approximate time diagram of the respective stages for the fulfillment of this thesis is presented.

5.1 Programming Language Choice

The official language for Android development is Java. Large parts of Android are written in Java and its APIs are designed to be called primarily from Java. It is possible to develop C and C++ app using the Android Native Development Kit [24], however it isn't something that Google promote. Using native code on Android generally does not result in a noticeable performance improvement, but it always increases your app complexity.

Java programming language was chosen for several reasons. The Java programming language also offers an extensive range of classes which are part of the Standard Developers Kit providing the programmer with flexibility and ease.

5.2 APIs Structure

5.2.1 AndroidManifest.xml

AndroidManifest.xml is the foundation file upon which every application operates. It is placed in app's root directory, with precisely that name. The manifest file presents essential information about your app to the Android system, information the system must have before it can run any of the app's code. Manifest declares predefined data types,

most important of which are: the name of the application, declaration of components (Activities, Services, Broadcast Receivers), Intents to be handled, libraries to be linked.

5.2.2 Basic Components

Subsequently in Figure 5.1 the basic layout of the system that was implemented is presented. In outline, one can see all the individual parts of the application and the connection between them. In the following the components will be described more thoroughly. Green color represents the Java classes and red the OpenOMR classes.

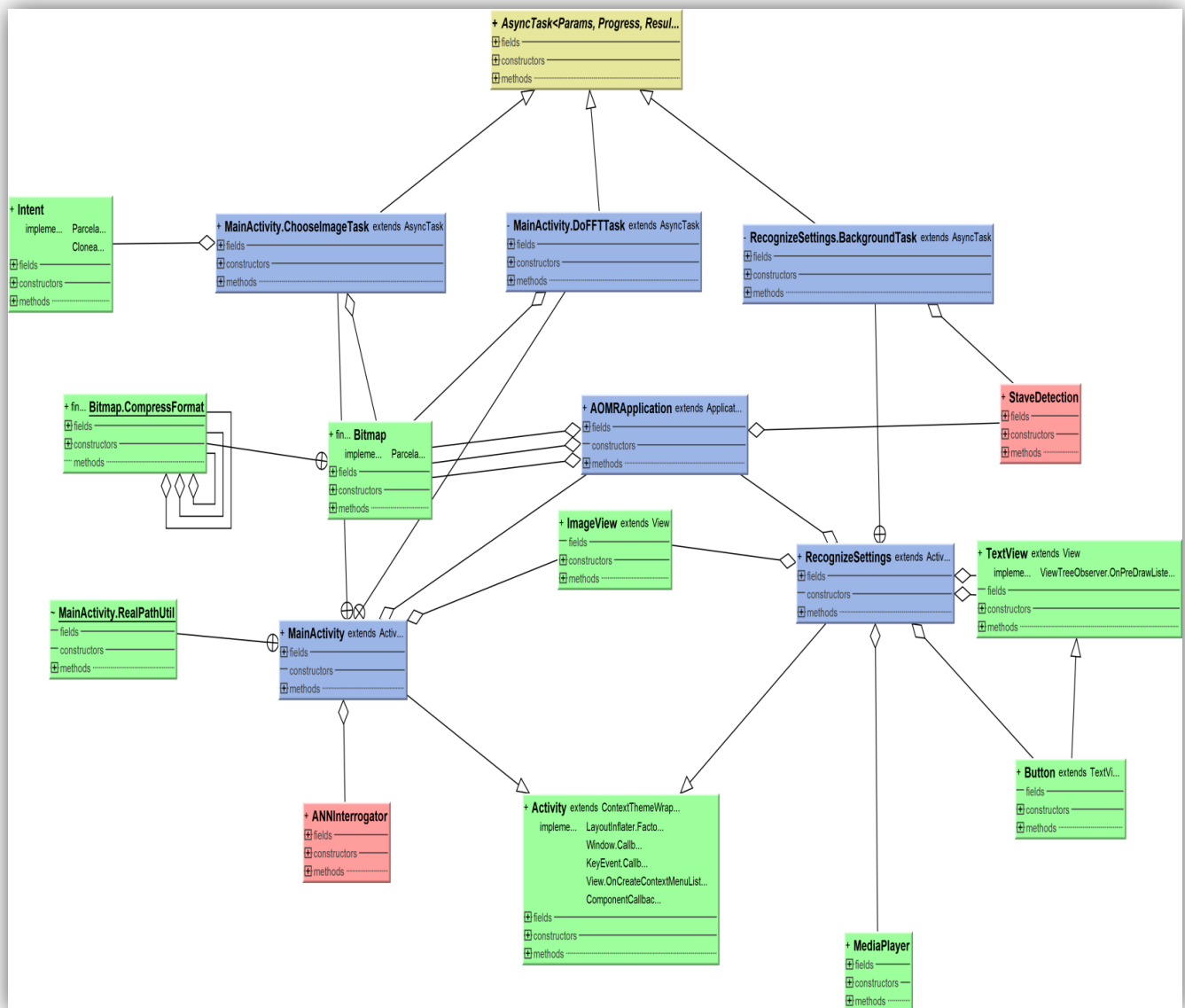


Figure 5.1: API – UML diagram

INTENTS

As we mentioned in section 2.2.2.2, the core components of an application (activities) are activated by intents. For the purposes of this application there were created intents for importing an image and for generating a MIDI file.

- For the selected image to be imported, intent is created which calls the appropriate android activity. As a result, the requested image is returned and then forwarded to the recognition routine by invoking a new intent that encapsulates the `MediaStore.Images.Media` class. It needs, however, to be stated that only a reference to the respective image by means of a Uniform Resource Identifier (URI) is used within the new intent call. The URI of the image chosen, is stored inside the Intent through its `putExtra()` method.
- After the MIDI file has been generated and stored an instance of (Another prominent call) `android.media.MediaPlayer` class provides for the playback routine.

AOMRApplication

As runtime configuration changes, such as a screen orientation or layout change, Android destroys and restarts the running activity with the new configuration. This results to processing all images again causing an unsmooth and slow experience to the user. This problem is overcome by using the global variable space.

In detail, an Application object (extends Application), is created so that we can declare the input and the recognized images as global variables for possible later use of every activity launched in our application. This technique is preferred as it is memory efficient.

5.2.2.1 MainActivity

The activity that is launched with the starting of the application is the MainActivity, which generates the central menu of our application (See figure 5.2). The user should, at first, press the “**Import Picture**”, which leads to a dialog window in order to show the folder’s directory in order to choose a music sheet image to be musically recognized. Upon selecting a file, the image of the score opened will be displayed in the main window. By importing an image, pre-processing techniques (See section 5.2.2.2) applied to the image by calling the respective functions.

Furthermore, in case that user suspects that imported image is skewed, there is an “**FFT**” button which indicates the possibility of a Fast Fourier transform to be applied on image so that a possible rotation angle of the imported image could be detected and corrected (See figure 5.4). Then the prior is replaced by the adjusted one. Finally the neural we need to

interrogate the neural network for potential use. The state of the neural network is loaded and the neural network is created. These activities are reflected by the code below:

```
fft.doFFT();  
rotAngle = fft.getRotationAngle();  
buffImage = rotImage.tilt();  
imageView.setImageBitmap(buffImage);
```

```
neuralNetInterrogator = new ANNInterrogator();
```

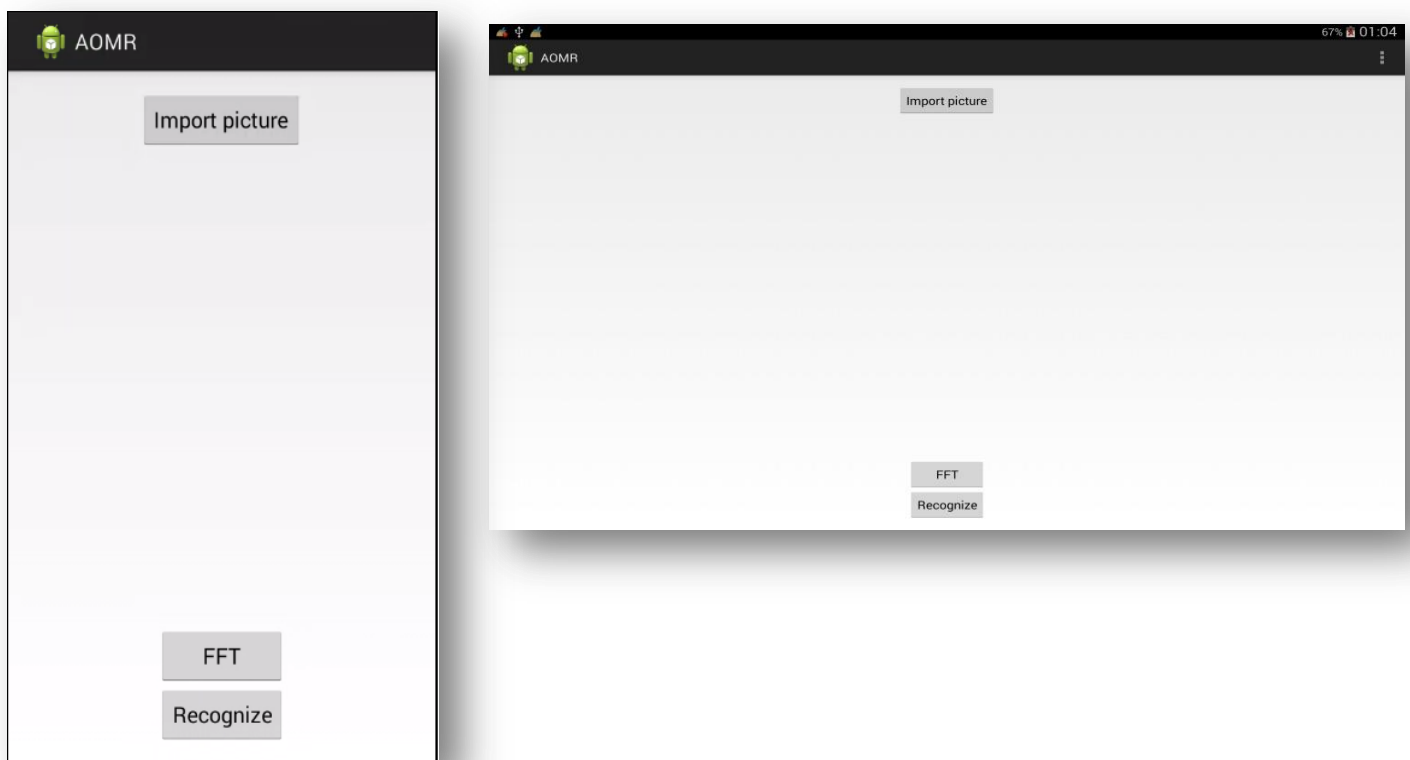


Figure 5.2: Main application menu in landscape and portrait orientation of the screen

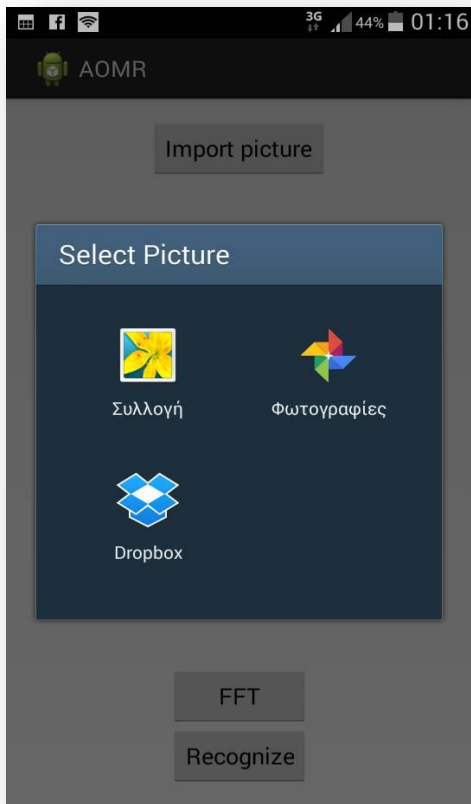


Figure 5.3: Display screen destination address option to import an image

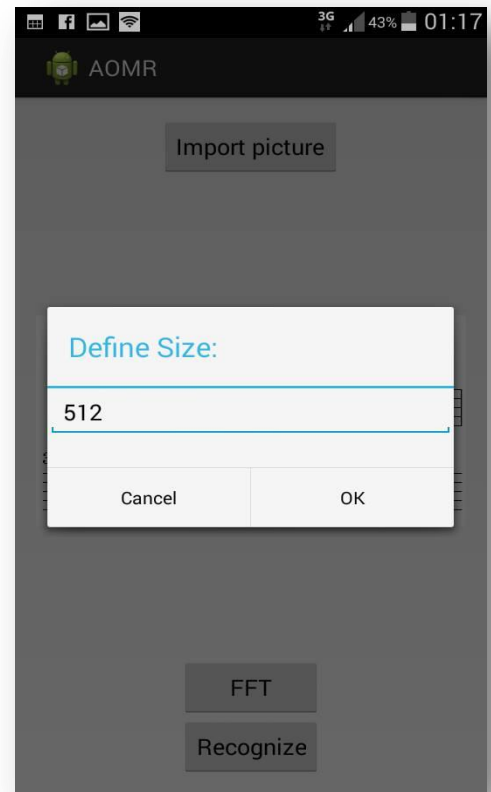


Figure 5.4: Display dialog box to define FFT size

The window will prompt you for the window size of the FFT and the input must be a power of 2 {512, 1024, 2048, ...}.

The FFT size defines the number of bins used for dividing the window into equal strips, or bins. Hence, a bin is a spectrum sample, and defines the frequency resolution of the window. This relationship can be modified proportionally with the oversampling factor. The number of bins of the windows can be increased via the Oversampling pop up menu FFT size zone in the dialogue window. By default, the FFT size is the first equal or superior power of 2, which corresponds to an oversampling factor of 1. The frequency resolution is improved. With an oversampling rate of 2, we have twice more bins in the window, and the frequency resolution is twice more precise. If user types a number that is not a power of two then we get a warning message and the user must re-enter the FFT size (See figure 5.5).

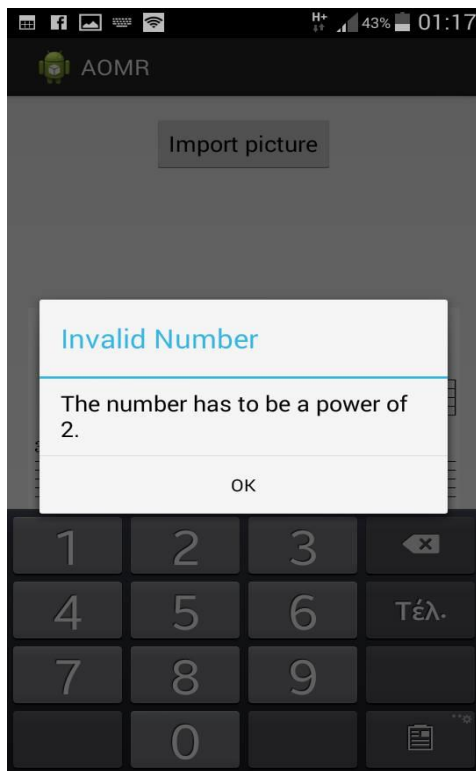


Figure 5.5: Warning Message for FFT size features

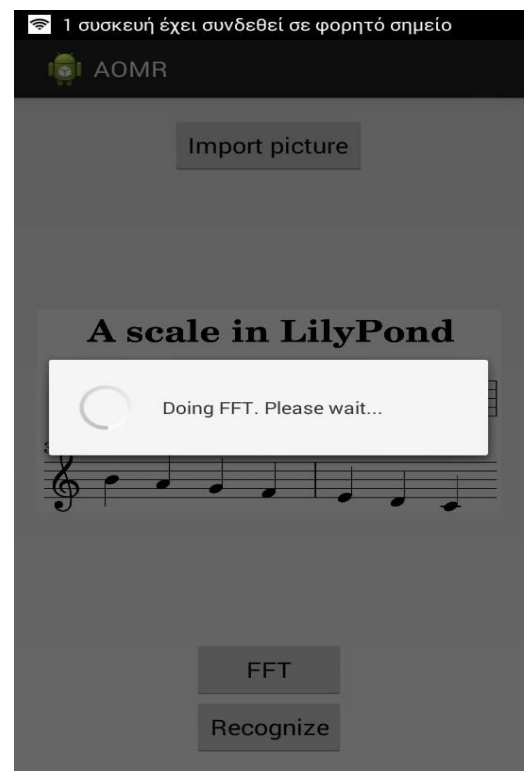


Figure 5.6: Display Dialog FFT Process

Since the initial processes have been accomplished, user should press the “**Recognize**” button for further process of the image and the RecognizeActivity Activity is created.

5.2.2.2 RecognizeActivity

After RecognizeActivity is launched the user will be prompted with a box as shown in figure 5.6. There are shown some predefined parameters that are used as threshold values in the image recognition process (specifically in Level 2 segmentation process). Users should first try recognising the score with the default parameters. The next step is to recognise the score and this can be done by clicking on the “**Recognize Score**” button in the toolbar.

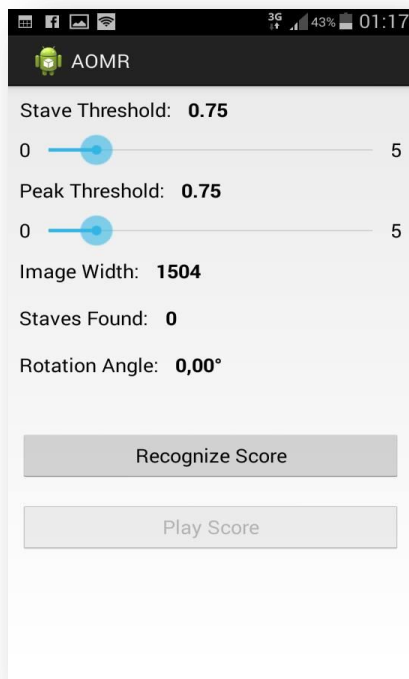


Figure 5.7: Display recognition Parameters before starting recognition process

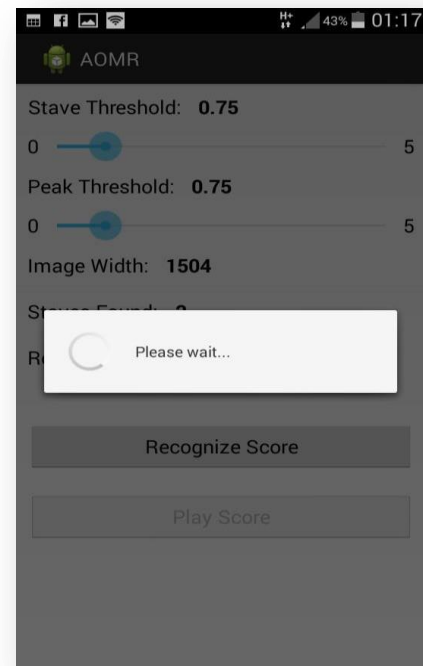


Figure 5.8: Display Dialog Recognition Process

After recognition process is fulfilled, the segmented/recognized image is displayed (See figure 5.7) along with the number of the staves founded and the rotation angle of the prior imported Image, which serve for verification purposes.

If system results in a wrong number of staves or in a misrecognized image, the user may adjust the parameters in the input bars and restart the recognition process with different parameters.

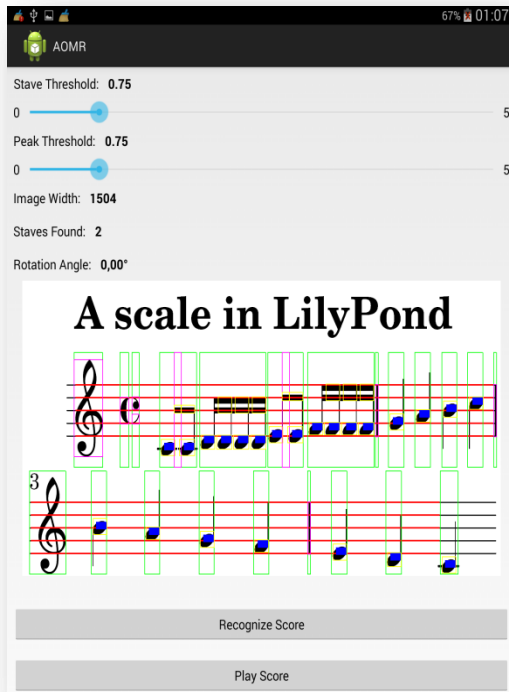


Figure 5.9: Display recognized score

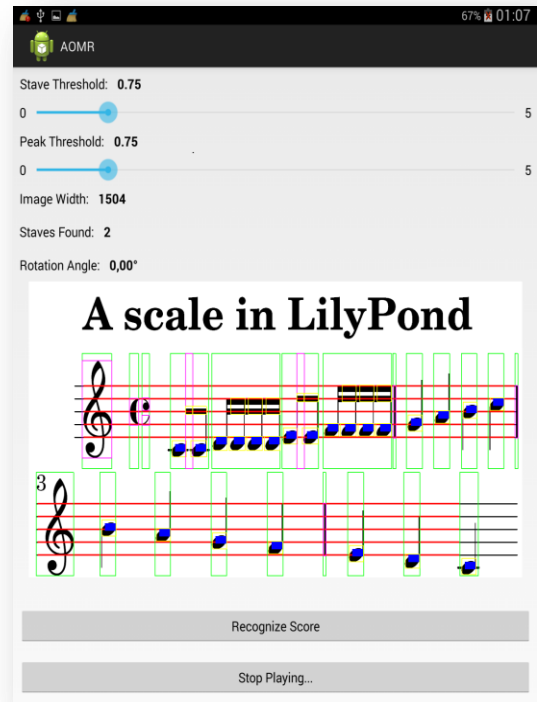


Figure 5.10: View while MIDI is generated

The next step is to play the score and this can be done by clicking on the “**Play Score**” button in the toolbar (See Fig. 5.9). User has the option to control the generation of the music by clicking “**Stop Playing...**” button (See Fig. 5.10).

Through RecognizeActivity all the methods (implemented in helper classes) required for the recognition of the score, are sequentially activated.

The code below reflects the main activities of the recognition process:

➤ *Take the y-projection of image*

```
YProjection yproj = new YProjection(bitmap);
yproj.calcYProjection(), bitmap.getHeight(), 0, bitmap.getWidth());
```

➤ *Calculate the stave line parameters*

```
StaveParameters pars = app.getStaveParameters ();
pars.calcParameters ();
```

➤ *Find all the staves*

```
StaveDetection staveDetection = new StaveDetection(yproj, pars);  
staveDetection.locateStaves ();  
staveDetection.calcNoteDistance ();
```

➤ *Segment image (Level0, 1 and 2) and recognize all symbols*

```
DetectionProcessor detection = new DetectionProcessor (bitmap, staveDetection);  
detection.processAll();
```

Midi creation and generation will be thoroughly discussed in the following section.

5.3 Fundamental implementation hurdles

Getting systems like Optical Music Recognition into an Android platform introduces a number of new challenges to do with image quality. Providing output on a mobile device in a timely fashion is also difficult because of relatively small local processing capacity. In this section the major limitations and costs that we have faced and successfully overcame, associated with this porting are explained as well as the way tackling them.

➤ *Image processing*

During the step of image processing one major step is binarization. In this manner Otsu's method [25], one of the most popular methods, to perform binarization was first implemented. Its advantage is that there is a low computational effort as only zero- and first order cumulative moments are utilized. Unfortunately, we came across some problems such as uneven illumination of the music sheet in certain environments as well as the inability to deal with strong illumination gradients. This led us to conclude that this approach is not sufficient for the field of mobile photographs. In order to overcome this difficulty in this project, we adopted a local thresholding method. (See section 4.2.1.1)

Furthermore one other major issue in the step of image processing was that image processing techniques effected images differently on the PC platform than the android API. Using GIMP platform⁴ we compared output images by their respective pixel values

⁴ 1. <http://www.gimp.org/>

2. <http://docs.oracle.com/javase/7/docs/api/java/awt/image/BufferedImage.html>

3. <https://msdn.microsoft.com/en-us/library/windows/desktop/ms534420%28v=vs.85%29.aspx>

4. https://en.wikipedia.org/wiki/Gamma_correction

and lead to different results. This step lead to the conclusion that accuracy issues in the recognition process on the android API have arised. This is caused by a bug in the `BufferedImage`² class of Java that does not exist in the `Bitmap`³ class which is supported by Android SDK. After carefully searching bibliography we came across that the method `"BufferedImage.getRGB (int x, int y)"`, unlike to `"Bitmap.getpixel (int x, int y)"`, applies a color conversion of the image's color model in the RGB space and a gamma correction⁴, which improves the quality of the image by default. Thus a gamma correction algorithm was manually implemented in our system and optimum gamma input values had to be decided after some experiments. During the color conversion a gamma correction algorithm⁴, which improves the quality of the image is also applied by default.

➤ **Neural Network Implementation**

To implement large and effective software neural networks, considerable processing and storage resources need to be committed. Neural network systems will often need to simulate the transmission of signals through many of the connections and their associated neurons – which must often be matched with incredible amounts of CPU processing power and time.

In an initial phase, the training database was created using SQLite API, recommended by the Android documentation, being ideal for structured data. But it became completely inadvisable for our purpose when the time to launch for the first time or train the neural network was too much time to dispense, in user's perspective way. Another disadvantage was the problem of division of works; in case of deleting the data application crashes. We needed a simpler way to use and edit the database information then the SQLite system.

Furthermore, we came across one very interesting point about Joone implementation to note. That is the single thread verses multithread performance. JOONE supports multithreaded implementation, as it is mentioned in its official release. Multithreaded allows the benchmark to take advantage of a multicore to take advantage of a multi core CPU. Though running JOONE in multithreaded mode we observed that this change caused our system to slow it down on a multicore machine. We found an article that reaffirms this Joone behavior, while benchmarking both versions. [26]

Our proposed system for android, supposes that the neural network's training is prosecuted off-line on a PC using a separate desktop application and saving the respective

weights of network arcs as a data file once the training is finished. Then we uploaded that file onto a mobile phone and imported it into the neural network in the Android system before performing a feed-forward method during real-time classification. This feature gives the application two advantages of high importance:

- A. Reduces processing time and mobile phone system resource usage for the neural network training significantly.
- B. Seeing the output result of the training process of our implemented neural network as an independent file it means that the file is portable and universal recognized.
- C. With a common interchange format these data sets could be built cumulatively. As new pieces of music are recognized and corrected, or if concluding to an optimized training output data, this work can be saved and used to train other systems with no further interventions. For example, an archive that has provided a data set of examples from a 16th-Century Italian music printer can make this data set available, allowing other systems to immediately re-train their recognition systems to take advantage of this new data and increase their accuracy on this particular repertoire.

➤ **MIDI GENERATION**

The result obtained from the recognition process will be integrated and converted to be a set of intermediate objects before interpreting it and exporting it as MIDI file. Several operating systems for mobile phones provide means of interpreting MIDI commands and of course, Android OS is no exception. The major limitation founded was that the task of real-time audio synthesis from MIDI, was complicated by the fact that respectively to “javax.sound.midi” library supported by java there is no native MIDI library on Android platform making synthesis of MIDI commands a challenge¹. Using a library from other computer languages such as C++ is inconvenient and error-prone, so we have to come up with a way to generate a MIDI file. The most common adopted solution is:

- ❖ Android API connects to a server for processing on another language. MIDI file is generated by mapping the pitches of the detected notes to corresponding MIDI numbers based on a MIDI table. Each note in the MIDI format requires a number identifying its pitch, a start time and end time, as well as its volume. Each pitch key is associated with a unique frequency. By matching note with its MIDI file music is generated for whole sheet. Different instruments are also provided. Upon

completion of the entire processing flow, the server will return a MIDI file to the Android device for playback with a click of a button.

This software does not support multiple staves, needs a server connection and is not time efficient. However, our application needed to be launched off-line and generating manually a MIDI file with the above technique would cost our application much more processing time. After some research the following methodology was developed: A MIDI library package⁵ designed for Android platforms was implemented. This package provides USB MIDI driver as the same interface as “javax.sound.midi” package. Implementing this package porting the application becomes easier. The issue though remains. The library was designed on terms of cooperation with another library for playing midi on other devices, and not directly to android. This is also confirmed by a package’s method called “MIDISystem.getDeviceInfo ()”, used to detect midi available devices, that did not return any device. Fortunately, after a more extended research, as the feature mentioned below is not exposed for commercial use, Android seems to use Sonivox for MIDI handling in the Media Player. It’s an embedded synthesizer that is integrated in Android. That means that the low-level software stack of android does have the support for it. It’s some kind of wrapper around midi that you can use to “dynamically” playback music.

- ❖ Combining these two possibilities we solved the dynamic midi generation issue as follows: programmatically generate a midi file, write it to the device storage, initiate android media player with the file and let it play. In detail, the file now serves as FileInputStream for our Media Player object. Audio feedback is then provided using a familiar layout, allowing the user to pause and resume play at any moment or at any position. Apart from the purpose described, the file may be used as an input for digital audio workstations or music notation programs, taking advantage of the communication interfaces provided by the Android OS. The piece of code that corresponds to midi creation is in class “RecognizeSettings. Java”, the second activity of the application that is created after you press “Recognize” button.

⁵ 1. <http://www.midi.org/aboutmidi/android.php>

2. <https://github.com/kshoji/USB-MIDI-Driver/wiki/javax.sound.midi-porting-for-Android>

The code that redirects these activities is:

```
ScoreGenerator scoreGen = new ScoreGenerator();  
scoreGen.makeSong();  
scoreGen.start();  
scoreGen.writeFile();  
playScore.setOnClickListener (new OnClickListener () {...});
```

5.4. Performance issues

5.4.1. Performance guideline

Google's products are mostly appreciated for their speed and responsiveness. From the very beginning Android (as another Google branded product) has been designed to provide its users a seamless experience and it has quite strict rules regarding responsiveness of applications. However, performance requirements had to be compromised somehow by the flexibility of the system. In other words: developers can, write fast and responsive applications but they can also do quite the opposite unless they are aware of general design principles and many optimization tricks (other than those imposed by the system).

There are two cases of situations that can be easily described as a "worst case scenarios" for Android apps [27]. The first case is clearly an uncouth exception (like null pointer exception, illegal argument exception etc.) which typically means just a small bug in a code that needs to be fixed. The second one is the ANR (Application Not Responding) error. Android imposes a strict rule that each Activity must respond to any input event in less than 5 second (10 seconds for *BroadcastReceivers*). Otherwise the ANR error is thrown and users are presented a dialog box where they have an option to kill the process (and application) or let it work for another 5-second period of time. In any case this is a clear signal to the users that something is wrong with the application. ANRs are typically caused by executing long running operations (network, I/O) on the UI thread so they can only be fixed by slightly modifying the architecture of certain components (adding a separate thread for instance).

The inner parts of OMR systems are considered long running operations and thus ANR errors were firstly a problem to our application too. To avoid ANRs we had to use the main

(UI) thread only to create/update views and delegate long running operations to separate threads as even accessing local memory can be noticeable [28]. The remaining part of this section will cover only two selected issues of performance and responsiveness improvements implemented in the AOMR application. The first one is related to managing large images and the other focuses on increasing the application's responsiveness by delegating expensive computations to separate threads.

5.4.1.1. Optimizing Image Handling

One of the biggest challenges in this system is scrolling through the large set of data created/used. Naturally, manipulating huge graphic files are also very demanding. The background here is that mobile devices are usually equipped with relatively slow CPUs and a limited amount of internal memory.

The solution adopted by our system tries to additionally minimise the amount of *expensive* operations. One of such operations is the *findViewById()* method which scans the view hierarchy in search for a child view with the given id [28]. Even though this operation cannot be completely avoided, there is no sense in repeating this process each time a *convertView* is populated. A static *ViewHolder* instance can be attached to the view (by tagging) to hold references to all subviews that have to be populated. Whenever there is a not null recycled *convertView* it will now certainly contain the *ViewHolder* instance.

5.4.1.2. Improving responsiveness

In order to keep the application's responsiveness under control one has to be constantly aware of the *Activities* lifecycle. After all *Activities*, as a single Android components which have a graphical representation are the only entry points for any user's action. Therefore all standard methods that are bound to the lifecycle have to be implemented very carefully not to overload the system during transitions between *Activities*.

The solution adopted by our system is using a background thread which removes strain from the main thread so it can focus on drawing the UI. In many cases, using AsyncTask [29] provides a simple way to perform your work outside the main thread. AsyncTask automatically queues up all the execute() requests and performs them serially. AsyncTasks are great for performing tasks in a separate thread, they have however one weakness. While the AsyncTask is in the middle of the work and the screen of device is rotated, the application crashes. This happens because when rotating the device screen a configuration change occurs, which will trigger the Activity to restart. The AsyncTask reference to the Activity is invalid.

In order to address this issue, we will add some logic to keep track of AsyncTask status (running/not running). While the task is running a progress bar dialog (See figure 5.7) would be presented to the user to indicate a background operation. We will dismiss the progress bar dialog when the fragment is detached from activity, and check the status of AsyncTask. If the status is “running”, this means we are returning from a screen orientation and we will just re-create and display the progress bar dialog to show that the AsyncTask is still working.

5.5. Allotment time

- ❖ The majority of the time to develop the AOMR application was needed required for research and comprehensive of the Optical Music Recognition process. In particular, it was important to understand in depth the sense by which the system deconstructed to the most basic of levels and which algorithms contribute to the implementation of each one of them. The time needed for this task was about three months.
- ❖ Also, a large portion of time required for finding the appropriate algorithm (in order to suit requirements of an Android Framework) for the Image pre-processing level as, well as identifying the issue, as mentioned in the previous section (See Section 6.1). The time needed for this task was about two weeks.
- ❖ Unlike, time integrating the neural network was much less as JOONE library on its functions providing simplifies our work considerably. Though, we had to evaluate alternative implementation techniques and methods for optimizing the training process. Time needed for this task was about two weeks.

- ❖ The time to deal with issues regarding the production and generation of MIDI files was of most importance. This part of the project required about three weeks for successful integration.
- ❖ Finally, a great time required for the integration of the OMR system in the Android framework, as well as to improve the user interaction with our application via the interfaces created, as our priority except the recognition rapidity, it was the best possible user interaction with our application three months.

The required times declared above are indicative and are referring to full working days.

Results

In this chapter a brief description of how OMR systems can be evaluated, along with the adopted evaluation method and some issues obtained. For our proposed system we first make a presentation of the techniques implemented to each stage in order to proceed to evaluation. Then, we will indicate the resulted accuracy of our system divided in stages. Finally we will test our overall system's performance in terms of resource utilization. A performance presentation will be made for the OpenOMR (See section 2.9.3) application, a pc platform application, for comparison purposes.

6.1 OMR Evaluation

Another challenging task in Optical Music recognition is to determine how the accuracy of proposed systems to be calculated or comparing competing systems. Different systems place different restrictions on the type of music that can be processed, with incompatible sets of musical features being the most common difference. There is absence of a common methodology and metrics that a system's performance would be based on. This is a more complicated issue than it might seem at first glance, because OMR systems can target different goals (audio playback, score archiving, . . .) and the outputs can be stored in very unlike formats. Trying to summarize the accuracy of an OMR system into a number is the major challenging task. Thus, it makes sense to quote accuracy rates in terms of the following OMR key stages: staff line identification, primitive detection, primitive assembly and musical semantics.

However, performance evaluation and related topics have been also studied in the literature. For example, Szwoch [30] proposed a method able to compare and evaluate the results of recognition systems stored in MusicXML format. More on this topic can be found in [31].

Testing the results of OMR against ground-truth data can be a highly time-consuming task, let alone the difficulty of finding a proper means of comparing this data. An interesting approach was presented in [32] who addresses the problem by measuring differences between the reference score and the tested score, while both objects are stored in the MIDI format. In the field of pattern recognition and classification, the following metrics are commonly defined:

1. C: Correct basic music symbol.
2. F: False basic music symbol.
3. M: Missing basic music symbol.
4. T: Total basic music symbol: $T = C + F + M$
5. R: Recognition rate.

Let $C_i = 1$ if the i th music symbol is correct, otherwise C_i is 0. F_i and M_i are analogously defined.

$$R = \frac{\sum_{0 \leq i < T} C_i}{T}$$

6.1.1 Basic Symbol Recognition Evaluation Results

Adopting previous section described technique, the overall accuracy rate of the OMR system is quoted in terms of the stages utmost importance.

In order to evaluate the OMR system implemented in AOMR application, a test set of images consists of 3 (A_{1-3}) synthetic scores that have been generated by Finale⁶, music notation software.

6. <http://www.finalemusic.com/>

➤ FFT

The FFT module is used to detect the angle by which staves are skewed in an image. This method is of particular interest to us as we want to be able to determine by how much a music score needs to be rotated in order to realign the stave(s) in such a way that they are parallel to the horizontal axis of the image.

Original Angle	FFT Module Angle	error
0°	0°	0 %
0.5°	0.506°	1.2 %
1°	1.0127°	1.27%
2°	2.099°	4.95%
Total_Overall_Error=1.85%		

Table 6.1: FFT results

As we can see in table 6.1, we evaluate the FFT module by rotating an image at different angles with the help of the GIMP image editor. Then the rotated images will be input to the FFT module and then we can compare the angle detected (FFT Module Angle) with the initial applied angle (Original Angle).

The test cases for the evaluation are as follows:

1. Rotate the original image by 0°
2. Rotate the original image by 0.5°
3. Rotate the original image by 1°
4. Rotate the original image by 2°

The test cases were decided by the fact that we refer to slight deformation of an image based on the common sense that indicates that the user tries to snap a picture in as focused as it can possibly be. The method has proven to give accurate results even if manipulating an image with a much greater angle than 2° but for the aforementioned reason we chose to statistically evaluate FFT module in the scale of 0° - 2°.

As we can notice in table 6.1 the statistical result is not referring to a normal distribution of the data set and it indicates much more a quantity result than a quality.

➤ *Stave Detection*

The simplest way to check that the staves were all correctly identified is to paint them over the recognized score. This module will therefore be visually assessed and if the staves do not appear to completely cover the staves in the original image, we will reject it as being correctly identified.

Stave Detection implemented algorithm shows satisfactory results, as we can observe in figures 6.1, 6.2, 6.3, in all cases except the case is skewed and FFT algorithm does not perform accurately.

➤ *Note Heads Detected*

The note heads are painted on top of the original notes as blue square boxes and this will enable us to visually assess how many note heads were correctly and incorrectly identified.

Image	#NoteHeads	#NoteHeads Found	#False Positives	#False Negatives
A.1	371	404	38	5
A.2	15	15	0	0
A.3	165	167	18	16

Table 6.2: Note head results

As we can see in table 6.2 the false positive rate indicates the note heads that are falsely identified by the application whereas the false negative rate gives us the number of note heads that failed to be identified by the application. Furthermore, after processing the image A.2 (see figure 6.2) we have achieved 100% accuracy. Statistically the false positive rate dominates over false negative rate due to the fact that the application misinterprets note heads to other symbols (i.e. text).

➤ *Pitch Calculation*

For each note head found, the program outputs a letter 'A' through 'G' based on the calculated pitch of the note head, for verification purposes. The pitch for each note head on the original score needs to be manually determined and this can then be compared with the pitch found by the program.

Image	% Accuracy
A.1	90.6
A.2	100
A.3	64.7

Table 6.3 : Pitch results

Image A.3, as we can from table 6.2 has shown the smallest accuracy in terms of the misinterpreted note heads and this validates the results presented in table 6.3 that refer on the pitch calculation stage accuracy.

➤ *Note Duration*

For each note head found, the program outputs the duration of the note as an integer having a value from '1' to '3'. A value of '1' represents a semi-quaver, a value of '2' represents a quaver and a value of '3' represents a crotchet. The output value is then compared with the actual duration of each note head in the original score.

The note duration for the scores A.2 (See figure 6.2) obtained 100% accuracy but the scores in figures A.1 and A.2 (See figure 6.1, 6.2) did not perform well at all and this is due to the output from the level 2 segmentation module in the musical object segmentation process.

The image displays a musical score for a song, consisting of ten staves. The score is annotated with various colored boxes and lines. Blue boxes highlight specific notes and phrases. Yellow boxes highlight other notes and phrases. Green boxes highlight the overall structure of the music. Red lines connect notes across staves, indicating harmonic relationships. The score includes a key signature of one sharp (F#) and a time signature of 4/4. The music is written in a standard musical notation with treble clefs. The score is divided into measures, with measure numbers 44, 47, 50, 53, 58, 60, 62, 64, 66, 68, 70, and 74 indicated on the left. The score ends with a double bar line and a 'rit.' (ritardando) marking.

Figure 6.1: A.1- Recognized. Recognized song available at:
<https://www.dropbox.com/s/6g19nzg599yxiqp/A1Sound.wma?dl=0>

A scale in LilyPond

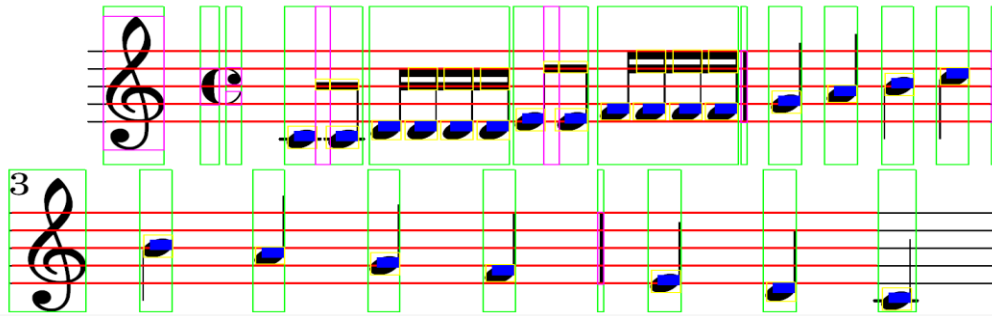


Figure 6.2: A.2- Recognized. Recognized song available at:
<https://www.dropbox.com/s/zjwbof4xdfsr6ct/A2sound.wma?dl=0>

A musical score for a song in LilyPond. The score is written on four staves. The top two staves are in treble clef and the bottom two staves are in bass clef. The key signature is one flat (B-flat). The time signature is common time (C). The score contains a melody in the top staff and a bass line in the bottom staff. The melody is marked with blue stems and blue note heads. The bass line is marked with blue stems and blue note heads. The score is enclosed in a green rectangular box.

Figure 6.3: A.3- Recognized. Recognized song available at:
<https://www.dropbox.com/s/i3cdk39zmo139ij/A3sound.wma?dl=0>

➤ *Neural Network*

There are several issues when training a neural network. First a decision needs to be made regarding an appropriate partition of input/output data sets prepared for a study. Total data sets are usually partitioned into three parts:

- *Training Set*: this data set is used to adjust the weights on the neural network.
- *Validation Set*: this data set is used to minimize overfitting. This data set is not adjusting the weights of the network, it is just verifying that any increase in accuracy over the training data set actually yields an increase in accuracy over a data set that has not been shown to the network before, or at least the network hasn't trained on it (i.e. validation data set). If the accuracy over the training data set increases, but the accuracy over then validation data set stay the same or decreases, then you're overfitting your neural network and you should stop training.
- *Testing Set*: this data set is used to determine the percent accuracy of the neural network and we want to know how many symbols are correctly and incorrectly classified.

In this training attempt, the network was trained with the backpropagation algorithm (See Section 2.4.2) and we adopted the following partition rules regarding data sets:

- *Validation set*: 10% of instances
- *Training set*: 70% of instances
- *Testing set*: 30% of instances that that do not appear in previous two data sets

The data set contains classes of glyphs which are divided in: Bass, crochet, Demisemiquaver line, flat, minim, natural, quaver br, quaver line, quaver tr, semibreve, semiquaver br, semiquaver line, semiquaver tr, treble, sharp.

<i>Glyph name</i>	<i>Average Accuracy</i>	<i>Average Confidence</i>
bass	100%	71.58%
crotchet	100%	84.19%
Demisemiquaver line	100%	99.55%
flat	100%	96.89%
minim	100%	93.89%
natural	100%	93.02%
quaver br	100%	99.51%
quaver line	100%	99.51%
quaver tr	100%	91.82%
semibreve	100%	75.47%
semiquaver br	100%	19.80%
semiquaver line	100%	94.87%
semiquaver tr	100%	71.13%
tremble	100%	99.60%
sharp	100%	84.95%

Table 6.4: Neural Network Results

Supervised neural networks that use a mean squared error (MSE) cost function can use formal statistical methods to determine the confidence of the trained model. The MSE on a validation set can be used as an estimate for variance. This value can then be used to calculate the confidence interval of the output of the network, assuming a normal distribution. A confidence analysis made this way is statistically valid as long as the output probability distribution stays the same and the network is not modified.

Out of the 15 different classes of glyphs trained by the neural network, the only class of data that did not obtain 100% accuracy was the ‘Sharp’, which showed 92.11% accuracy. Neural networks are trained to remember certain patterns and in the case of the sharp its shape is similar to the natural glyph which therefore explains why it is sometimes classifying it as a natural.

In conclusion, the neural network implemented in this application gives satisfactory results but the low performance of level-2 segmentation stage determines the overall low accuracy of the application.

6.2 Application Resource Utilization

In software engineering, profiling is a form of dynamic program analysis that measures, for example, the memory or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization.

The Image which was imported to both systems to create the perspective profiles is score A1 (See figure 6.1), a PNG file which features 864,256 Kbytes.

6.2.1. Local Server

To eliminate the need for testing session configuration we implemented JProfiler⁷ as an eclipse plugin in order to isolate the OpenOMR program performance. This plugin allows collecting “real-time” data from the profiled Java Virtual Machine.

The graphs following represent:

- ❖ CPU usage as a simple percentage of CPU time spent on non-idle tasks.
- ❖ Allocated and used heap Memory on non-idle tasks.

Benchmarking Computer

The handset used during the OpenOMR profiling is an HP Pavilion g6 series that features an Intel Core i-5 (2.3 GHz) processor and 4GB of RAM.

⁷ <https://www.ej-technologies.com/products/jprofiler/overview.html>

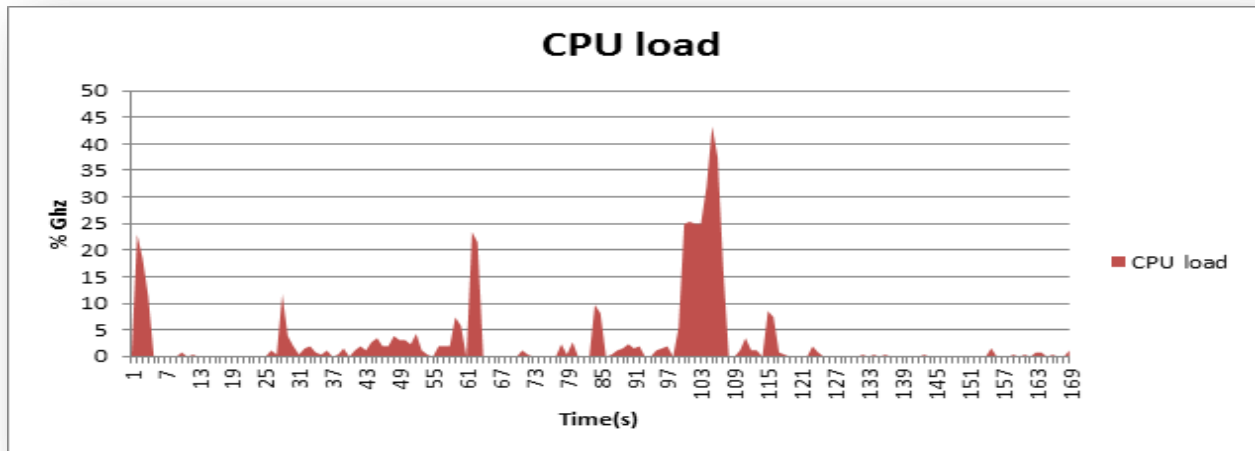


Figure 6.4 : OpenOMR CPU Utilization

The peak is detected when recognition procedure begins and is estimated to reach 44%, which means that the system needs 1.012 GHz.

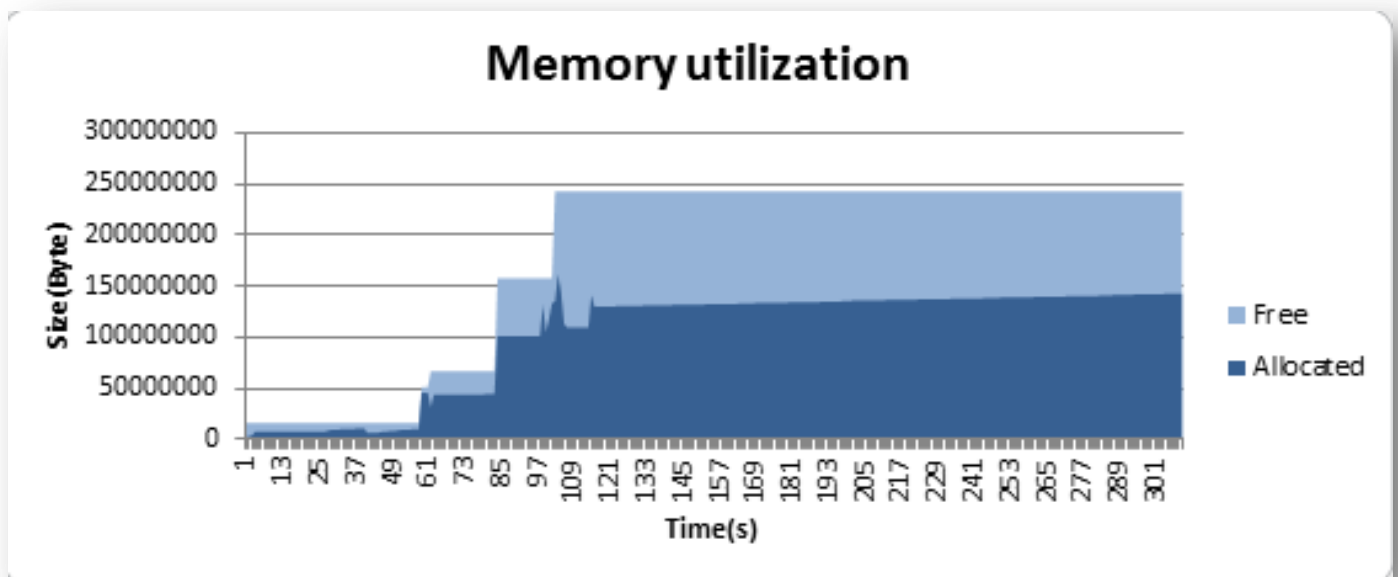


Figure 6.5: OpenOMR Memory Utilization

In detail:

- The first peak is detected as OpenOMR is launched in order to settle the GUI which needs about 24% of CPU, 3MB of memory.
- Then, follows the image directory choosing task. Its peak is detected at 12% of CPU when choosing window is launched.
- In order to load the chosen image into the application the system needs about 23% of CPU, 50 MB of memory.
- In case of the FFT algorithm is applied to the image, the system needs about 9% of CPU, 100 MB of memory and 5s of time.
- When recognition process begins we detect the overall peaks. For image recognition system needs about 44% of CPU, 150 MB of memory and 8s of time.
- Finally, in order to generate a midi file, the system needs about 9% of CPU and the memory used is not affected.

*The elapsed time is measured by `System.currentTimeMillis()`, a Java method which indicates how many seconds or millisecond a certain procedure is taking in order to execute.

6.2.2. Android framework

Benchmarking Smartphone

The handset used during the latter part of development and testing is a Samsung (SM-T520) tablet that features a Quad-core (1.5 GHz) processor , 2GB of RAM and Android™ 4.4, KitKat) Os running.

To eliminate the need for testing session configuration we implemented Monitor tool [33] as a plugin in Android Studio software that reports in real-time how your app allocates memory. The Prerequisites to complete the Application's profile are:

- A mobile device with Developer Options [34] enabled.
- An Application with USB Debugging enabled.

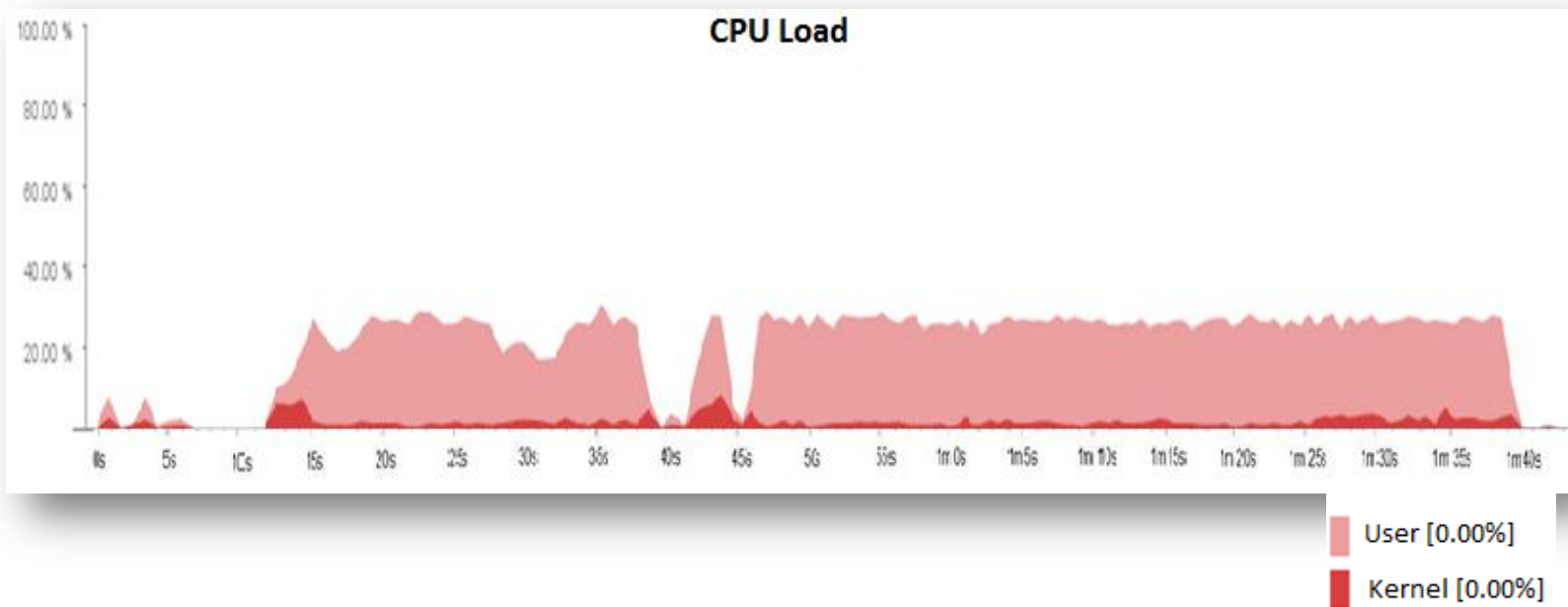


Figure 6.6: AOMR CPU Utilization

We must note that the CPU graph presented above indicates the total use of the system's CPU on the non-idle tasks on a tablet that features a Quad-core (1.5 GHz) processor. When observing each core separately we noticed that during the whole procedure one of the cores was performing on its maximum level, 100%.

The peak is detected when recognition procedure begins and is estimated to reach 38%, which means that the system needs 0,59GHz.

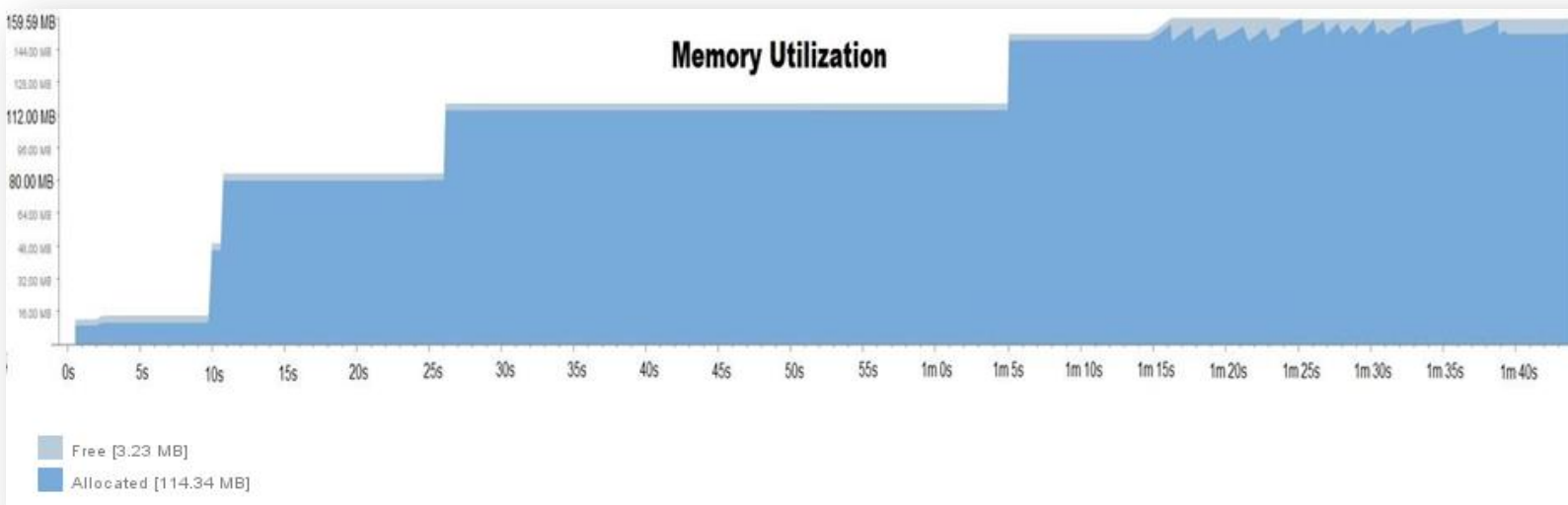


Figure 6.7 : AOMR Memory Utilization

The memory space occupied by the application as we can see in figure 6.7 reaches 159MB which is a large amount of space to be handled by an application in mobile device environment.

In detail:

- The first peak is detected as AOMR is launched in order to settle the UI which needs about 15% of CPU, 12MB of memory.
- Then, follows the image directory choosing task. Its peak is detected at 12% of CPU when choosing window is launched.
- In order to load the chosen image into the application the system needs about 30% of CPU, 80 MB of memory.
- In case of the FFT algorithm is applied to the image, the system needs about 26% of CPU, 112 MB of memory and 5s of time.
- When recognition process begins we detect the overall peaks. For image recognition system needs about 38% of CPU , 159 MB of memory and 52 second of time.
- Finally, in order to generate a midi file, the system is not further affected.

Future Improvements

There are a few aspects in which this project can be improved to obtain more accurate regeneration of the original sheet music. Due to time constraint, there were not implemented on this project but they would certainly be interesting to work on for the future.

➤ Level-2 Segmentation

Ideally, the glyphs produced from the level 2 segmentation module should be used to determine the overall accuracy of the neural network. However, the level 2 segmentation module is not producing accurate results as is and if the level 2 segmentation module is improved in a future release, this method for testing the neural network should be used.

➤ Fetch Music from a Database

It is possible to build a recognition system based on short piece of correctly-recognized music. By using the first few notes of a piece of music as reference, the system would be able to determine the title of the music and retrieve a well-performed version of that music.

First, we would build a database that contains a large number of good-quality music. Then after the sheet music is analyzed, we should be able to match the first twenty notes with the corresponding music in the database. Once we find a high-probability match, that music file would similarly be streamed to the Android phone for playback. If no match is found, a MIDI file could be generated as what we have done.

➤ Interactive Features

A nice feature would be to allow users to click on the recognized score in order to modify incorrectly identified notes. For example, if a note was omitted, the user could simply click on that note and the application would add it. Another interactive feature would be to illuminate the notes in a different colour as they are played.

➤ Polyphony

Due to the specialization on simple tunes, it was obvious to have the software recognize each sequence of staves as one line of a song. Given a notation sheet that not only provides for one but for two or more voices/instruments, the algorithm will recognize each subsystem as subsequent lines. Hence, the melody, which is intended to be polyphonic, will be played as a sequence instead of simultaneously. In order to deal with this issue, we could introduce an algorithm that measures the gap between each set of staves in order to derive related sets of different voices. Another feature of related staves is the property of having barlines reaching from the top of the first stave to the lowest staff line of the stave that depicts the last voice. Another way of representing a polyphonic musical piece where the respective voices are identical in phrasing is to attach more than one note head to a stem. That particular case sees the detection algorithm recognizing the leftmost note that is connected to the stem while skipping recognition on the other side of the stem. Consequently, the algorithm design would need to be adjusted in order to satisfy the needs of such types of notation.

Conclusion

My task was to design and implement an Android application that would enable users to generate a song from a sheet music image. The significant part was dedicated for designing and implementing an API which constitutes an offline tool for “reading” sheet music. To build this kind of system on a mobile phone is quite a challenge. Apart from the problems of limited computational resources and electrical power, there are other limitations to be concerned in terms of technical software development as well. The challenge posed by non-perfect sources has been highlighted in this thesis, making a wise choice of methods necessary for obtaining good results. Image processing enhancement, versatility independency of neural network implementation and audible representation via MIDI synthesis were the most challenging tasks and constitute the innovative parts of our proposed system. AOMR application takes advantage of many solutions and techniques already pre-installed on the Android platform. It has been optimized according to the best practices recommended by Google in order to increase user experience, maintain system stability and reduce power consumption while providing an acceptable performance.

By the time of finishing this thesis, great progress has been made in offering OMR software to a broader audience, as we achieved to overcome issues arisen adequately. This application is available to end users for an offline optical music recognition which leads to a shift in workplace. Now, musicians and composers alike are allowed to write or capture music notation independent of their current location.

At the same time, system’s accuracy problem is still one of the open questions left, as the algorithms chosen have found to be limited in their capabilities. Moreover, results could be further improved by adopting an interactive approach, where both user and recognition framework benefit from each other.

Bibliography

- [1] Bainbridge, D., & Bell, T. (2001). The challenge of optical music recognition. *Computers and the Humanities*, 35(2), 95-121.
- [2] McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4), 115-133.
- [3] Morris, R. G. M. (1999). DO Hebb: The Organization of Behavior, Wiley: New York; 1949. *Brain research bulletin*, 50(5), 437.
- [4] Colton, Simon: Introduction to Artificial Intelligence. Department of Computing, Imperial College London, UK . http://www.doc.ic.ac.uk/_sgc/teaching/v231/index.html
- [5] Johnston, A. (2005). *Classifying Persian Characters with Artificial Neural Networks and Inverted Complex Zernike Moments* (Doctoral dissertation, PhD Thesis, Imperial College of London).
- [6] Honkela, A. (2001). *Nonlinear switching state-space models* (Doctoral dissertation, UNIVERSITY OF TECHNOLOGY).
- [7] PARIZEAU, M. (2006). Réseaux de neurones, GIF-21140 et GIF-64326, Notes de cours et chronologie.
- [8] Haykin, S. S., Haykin, S. S., Haykin, S. S., & Haykin, S. S. (2009). *Neural networks and learning machines* (Vol. 3). Upper Saddle River: Pearson Education.
- [9] INTRODUCTION TO FOURIER TRANSFORMS FOR IMAGE PROCESSING | UNM Computer Science http://www.cs.unm.edu/_brayer/vision/fourier.html
- [10] Pinto, T., Rebelo, A., Giraldi, G., & Cardoso, J. S. (2011). Music score binarization based on domain knowledge. In *pattern recognition and image analysis* (pp. 700-708). Springer Berlin Heidelberg.
- [11] Burger, W., & Burge, M. J. (2009). *Digital image processing: an algorithmic introduction using Java*. Springer Science & Business Media.
- [12] Miles-Huber, D. (1991). The MIDI manual. USA. Howard W. Sams.
- [13] Selfridge-Field, E. (1997). *Beyond MIDI: the handbook of musical codes*. MIT press.
- [14] Colton, Simon: Introduction to Artificial Intelligence http://www.doc.ic.ac.uk/_sgc/teaching/v231/index.html Department of Computing, Imperial College London, UK
- [15] MusicXML for Exchanging Digital Sheet Music. <http://www.recordare.com/xml/faq.html>

- [16] Musitek SmartScore Music Scanning Software. www.musitek.com/musitek.html
- [17] SharpEye music scanning software, SharpEye music OCR program, SharpEye music reader. <http://www.music-scanning.com/sharpeye.html>
- [18] Fast conversion of printed scores to MIDI and Capella format, http://www.software-partners.co.uk/index.php?option=com_content&task=view&id=28&Itemid=71
- [19] OpenOMR project | SourceForge. <http://sourceforge.net/projects/openomr/?source=navbar>
- [20] Yong Li: Music Sheet Reader - An Implementation of Optical Music Recognition System University of Birmingham UK
- [21] Jaime S. Cardoso , "Classification of Ordinal Data", MSc in Mathematical Engineering, FCUP, November 2005.
- [22] Zaragoza, H., & Alché-Buc, F. (1998) . In *Proceedings of the Seventh Int. Conf. Information Processing and Management of Uncertainty in Knowledge Based Systems*. (IPMU), vol.1, pp 886-893.
- [23] Joone-an Object Oriented Neural Engine download | SourceForge. <http://sourceforge.net/projects/joone/>
- [24] Android NDK | Android Developers <https://developer.android.com/tools/sdk/ndk/index.html>
- [25] Level Otsu, N. (1979). A threshold selection method from gray-level histogram. *IEEE Transactions on Systems, Man and Cybernetics*, 9(1), 62-66.
- [26] Benchmarking and Comparing Encog, Neuroph and JOONE Neural Networks | CodeProject. June 3, 2010 <http://www.codeproject.com/KB/recipes/benchmark-neuroph-encog.aspx?ref=dzone>
- [27] Keeping Your App Responsive | Android Developers: <http://developer.android.com/training/articles/perf-anr.html>
- [28] Making ListView Scrolling Smooth | Android Developers: <http://developer.android.com/training/improving-layouts/smooth-scrolling.html>
- [29] AsyncTask | Android Developers: http://developer.android.com/reference/android/os/AsyncTask.html#SERIAL_EXECUTOR
- [30] Szwoch, M. (2008). Using MusicXML to evaluate accuracy of OMR systems. In *Diagrammatic Representation and Inference* (pp. 419-422). Springer Berlin Heidelberg.
- [31] Byrd, D., & Simonsen, J. G. (2013). Towards a standard testbed for optical music recognition: Definitions, metrics, and page images. *University of Copenhagen, Copenhagen*.
- [32] Bellini, P., Bruno, I., & Nesi, P. (2007). Assessing optical music recognition tools. *Computer Music Journal*, 31(1), 68-93.

[33] Memory Monitor Walkthrough | Android Developers:

<http://developer.android.com/tools/performance/memory-monitor/index.html>

[34] Using Hardware Devices | Android Developers:

<http://developer.android.com/tools/device.html#developer-device-options>

[35] IBISWorld. 2011. Sheet music publishers in the US: market research report, IBISWorld Industry Report, Vol. Electronic resource Ibisworld: 30.

[36] IBISWorld. 2013. Sheet music publishers in the US: market research report, IBISWorld Industry Report, Vol. Electronic resource Ibisworld: 30.