VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

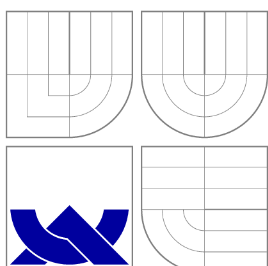
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ČTEČKA BRAILLOVA PÍSMA PRO ANDROID OS

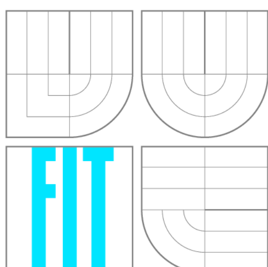
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — DANIEL BOKIŠ
AUTHOR

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ČTEČKA BRAILLOVA PÍSMA PRO ANDROID OS

BRAILLE READER FOR ANDROID OS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — DANIEL BOKIŠ
AUTHOR

VEDOUCÍ PRÁCE — Ing. BORIS PROCHÁZKA
SUPERVISOR

BRNO 2012

Abstrakt

Tato práce se zabývá analýzou a rozpoznáním znaků Braillova písma ve fotografii pořízené mobilním telefonem. Popisuje řešení návrhu a implementace aplikace pro rozpoznávání na mobilní platformě Android OS. Popsané obecné principy analýzy a zpracování obrazu jsou však uplatnitelné rovněž u jiných systémů.

Abstract

This thesis deals with the analysis and recognition of the Braille characters from a photo taken by a mobile phone. It describes design and implementation of application for recognition on Android OS mobile platform. Described general principles of analysis and processing of image are also applicable in other systems.

Klíčová slova

Braill, Braillovo písmo, Android OS, zpracování obrazu, rozpoznání znaků, výpočet mobilním telefonem

Keywords

Braille, Android OS, image processing, character recognition, mobile computing

Citace

Daniel Bokiš: Čtečka Braillova písma pro Android OS, bakalářská práce, Brno, FIT VUT v Brně, 2012

Čtečka Braillova písma pro Android OS

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Borise Procházky. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Daniel Bokiš
14. května 2012

Poděkování

Tímto bych rád poděkoval panu Ing. Borisi Procházkovi za konzultace a také své přítelkyni a kamarádům za pomoc při shánění testovacích vzorků.

© Daniel Bokiš, 2012.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Úvod **2**

1 Braillovo písmo **3**
- 1.1 Základní informace . . . 3
- 1.2 Různé jazyky a normy Braillova písma . . . 4

2 Android OS **5**
- 2.1 Základní principy . . . 5
- 2.2 Tvorba aplikací pro Android OS . . . 9
- 2.3 Práce s fotoaparátem . . . 11
- 2.4 Převod textu na zvuk . . . 11

3 Návrh rozpoznání Braillova písma v obraze **12**
- 3.1 Předzpracování obrazu . . . 13
- 3.2 Detekce objektů a odstranění ruchu . . . 16
- 3.3 Seskupení . . . 20
- 3.4 Analýza a překlad . . . 24

4 Implementace aplikace pro Android OS **25**
- 4.1 Součásti aplikace . . . 25
- 4.2 Uživatelské rozhraní . . . 28
- 4.3 Optimalizace . . . 29

5 Testování **31**
- 5.1 Podmínky zachycení obrazu . . . 31
- 5.2 Výsledky testování . . . 33

Závěr **34**

Seznam použitých zdrojů **37**

Seznam použitých zkratek a symbolů **38**

Seznam příloh **39**

A Základní česká znaková sada **40**

B Ukázky testovacích obrázků **41**

Úvod

Na světě je přibližně 285 milionů zrakově postižených lidí. Z toho je asi 39 milionu slepých [1]. Místo klasického textu mohou nevidomí použít Braillovo písmo, které je možné číst hmatem. V České republice můžeme toto písmo nalézt na mnoha veřejných místech. Nejtypičtějším místem jsou výtahy, dále pak vlaky, autobusová nádraží a jiná místa obsahující informační sdělení. Mnoho nevidomých a slabozrakých však toto písmo neovládá. Například v USA dokáže číst Braillovo písmo pouze 10% slepých [2].

Mohlo by tak být žádoucí, vytvořit program, který by dokázal přeložit Braillovo písmo z fotografie pořízené přenosným zařízením. V dnešní době nástupu chytrých telefonů již existují funkce, dovolující nevidomým tato poměrně komplikovaná zařízení ovládat. Jedná se například o systém převodu textu na hlas, který je aktivován postupným posouváním prstu nad jednotlivými prvky na displeji. Mohlo by nás napadnout, že by bylo jednodušší, vytvořit aplikaci schopnou rozpoznat z fotografie běžný psaný text. Problém je však ten, že nevidomý psaný text nevidí a tak neví, kde přesně se vyskytuje a tedy neví co má fotit. Proto se zaměříme na rozpoznání Braillova písma, jež může nevidomý snadno lokalizovat hmatem.

Cílem této práce je tedy návrh metod, které umožní rozpoznat jednotlivé znaky Braillova písma z fotografie pořízené mobilním telefonem a dokáží znaky převést na psaný text. Tyto metody jsou následně uplatněny v praxi a to v aplikaci vytvořené pro mobilní telefon. Ta může sloužit jako pomoc pro nevidomé, nebo jako překladač pro běžné nadšence, které zajímá, co jednotlivé nápisy v Braillově písmu znamenají.

Aplikace by teoreticky mohla běžet na jakémkoliv výkonějším mobilním telefonu s fotoaparátem, nezávisle na jeho operačním systému. V poslední době však nejrychleji roste trh s mobilními telefony vybavenými operačním systémem Android. Podle informací z roku 2011 jím bylo osazeno 48,8% nových zařízení zakoupených v tomto roce [3]. Pro srovnání druhý byl systém iOS s pouhými 19,1%. Z tohoto důvodu je naše aplikace implementována právě pro operační systém Android.

Samotná práce začíná dvěma teoretickými kapitolami. První z nich se zabývá obecně Braillovým písmem a v druhé jsou přiblíženy základní principy systému Android. Třetí kapitola popisuje návrh rozpoznávání znaků Braillova písma ve fotografii. Tato kapitola je z celé práce stěžejní a zde popsané metody jsou následně použity ve výsledné aplikaci. Samotná implementace včetně popisu uživatelského rozhraní a použitých optimalizací je přiblížena ve čtvrté kapitole. V poslední kapitole jsou diskutovány výsledky testování a prezentována celková úspěšnost vytvořené aplikace. V závěru jsou pak zhodnoceny dosažené výsledky a navržena možná vylepšení do budoucna.

Kapitola 1

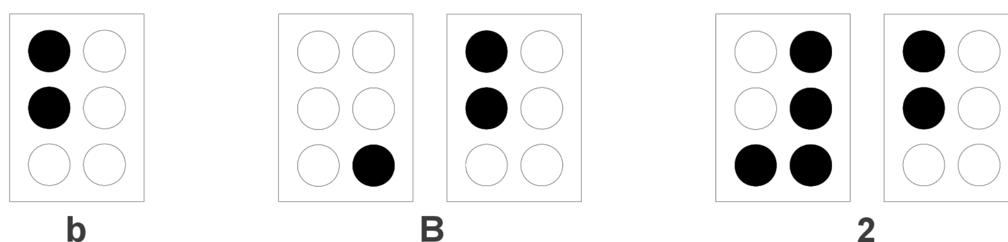
Braillovo písmo

Tato kapitola se zabývá Braillovým písmem a jeho principy. V první části textu je popsána historie, grafická podoba písma a základy sémantického významu. Druhá část se pak zabývá normami a jazykovými modifikacemi Braillova písma.

1.1 Základní informace

Braillovo slepecké písmo je speciální druh písma určený pro nevidomé a slabozraké [4]. Funguje na principu plastických bodů, které čtenář vnímá hmatem. Písmo je pojmenováno podle Louise Brailla, tehdy patnáctiletého francouzského nevidomého, který písmo vytvořil úpravou vojenského systému, umožňujícího čtení za tmy. Každé písmeno tvoří mřížka šesti bodů uspořádaných do obdélníku 2x3. Na jednotlivé body se odkazuje čísly 1-6. Některé z těchto bodů jsou pak vyvýšeny. Tímto způsobem je možné zakódovat 2^6, tedy 64 různých znaků. Prázdný znak se používá pro mezeru, tedy zbývá 63 použitelných znaků. Vzhledem k tomu, že každá pozice může nabývat pouze dvou stavů, je Braillovo písmo netradičním příkladem použití binárního kódu mimo počítačovou techniku.

Základní sada obsahuje především malá písmena a interpunkci. Čísla a velká písmena se tvoří pomocí prefixů. Například písmeno „b“, „B“ a číslice 2 se zapíší stejným znakem, jen u velkého písmene a číslice je použit odpovídající prefixový znak (obr. 1.1).



Obrázek 1.1: Příklad užití prefixů.

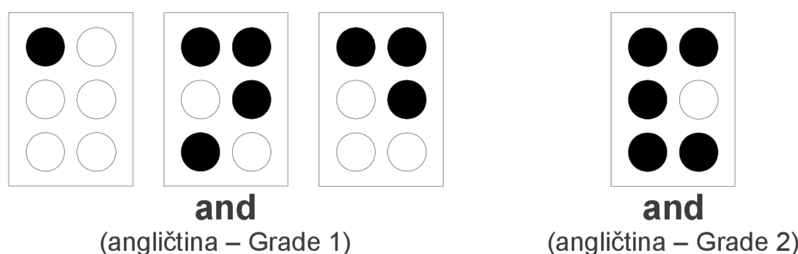
Existuje také modifikace, kde se k šestibodové mřížce přidala další řada dvou bodů, čímž počet použitelných symbolů vzrostl na 255. Takto je možné zapsat všechny znaky ASCII tabulky. Osmibodové písmo se však v praxi příliš nepoužívá, proto se mu v této práci nebudeme věnovat.

1.2 Různé jazyky a normy Braillova písma

Jedním z problémů Braillova písma je jeho nejednotnost v různých jazycích [5]. Jelikož má pouze 63 použitelných znaků, nelze zakódovat znaky všech jazyků v jedné společné normě. Hlavní rozdíly jsou samozřejmě ve znacích s diakritikou, základní sada znaků je u jazyků používající latinku víceméně obdobná.

Existují i varianty Braillova písma pro jazyky nepoužívající latinku, jako je řečtina, hebrejština či čínština, japonština a další. V této práci se ale zaměříme hlavně na písmo české a anglické. Braillovo písmo se dá také použít pro matematické [6] anebo hudební zápisy [7]. Tyto normy jsou však od klasického textu naprosto odlišné a prakticky se nenachází na veřejných místech, kde nás Braillovo písmo zajímá. Proto se jimi také nebudeme blíže zabývat.

I v jazycích používajících latinku se vyskytuje více forem. Existuje základní forma (plnopis, Grade 1), ve které se slova přepisují znak po znaku a dále komplexnější forma (zkratkopis, Grade 2), ve které každý znak reprezentuje také skupinu hlásek. Tedy například anglická spojka "and"se ve formě Grade 1 zapíše třemi znaky a v Grade 2 pouze jedním (obr. 1.2). V češtině se s touto komplexnější formou nesetkáme, ale například v angličtině je běžná a základní formu používají prakticky pouze začátečníci.



Obrázek 1.2: Porovnání spojky "and"v Grade 1 a Grade 2.

U formy Grade 2 je však ten problém, že jsou některé zkratky reprezentovány stejným znakem jako samostatná písmena (např. not a n). Stejně tak není výjimkou, že se pod jedním znakem skrývá více zkratek. O správné reprezentaci tak rozhoduje umístění znaku v rámci slova a v daném kontextu [8]. Rozpoznání již tedy vyžaduje určitou inteligenci a případně kontrolu ve slovníku všech slov daného jazyka. Z důvodu zjednodušení a s přihlédnutím k tomu, že se v českém jazyce zkratkopis nepoužívá, nebudeme se v této práci stupněm Grade 2 zabývat.

Kapitola 2

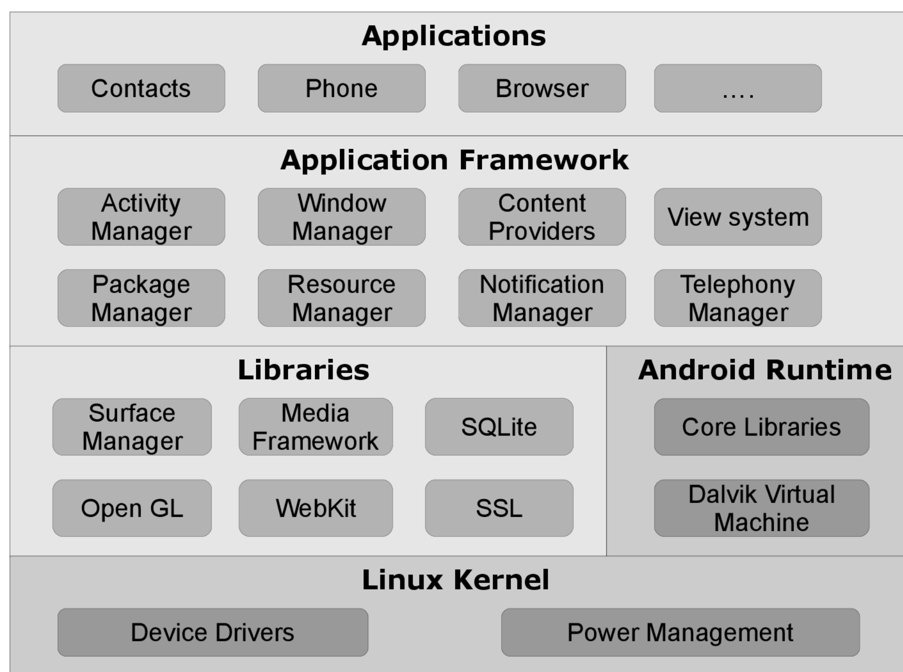
Android OS

Android je poměrně nový open source operační systém pro mobilní zařízení vyvinutý společností Google a uskupením OHA. V této kapitole si nejprve představíme klíčové vlastnosti a principy tohoto systému. Dále jsou zde popsány způsoby tvorby aplikací pro tento systém a nakonec se budeme zabývat technikami pro využití fotoaparátu a převodu textu na zvuk, což jsou nezbytné součásti plánované aplikace.

2.1 Základní principy

Architektura systému

Architektura systému je rozdělena do několika vrstev (obr. 2.1) přičemž každá vrstva využívá služeb poskytovaných vrstvou pod ní.



Obrázek 2.1: Achitektura systému Android.

Vrstvy počínaje spodní jsou [9, 10]:

- **Linux Kernel** – Linuxové jádro poskytuje hardwarovou abstraktní vrstvu, což Androidu dovoluje pracovat na různých platformách nyní i v budoucnu. Android interně používá Linux pro správu paměti, správu procesů, sítí a další služby operačního systému. Samotný uživatel však prakticky s Linuxem nikdy nepřijde do styku a také programátor jej nikdy nebude volat přímo, k tomu slouží vyšší vrstvy.

- **Libraries** – Další vrstvou jsou knihovny, které jsou psány v jazyce C nebo C++ a jsou kompilovány přímo pro určitou hardwarovou architekturu telefonu. Dají se zde nalézt například knihovny zodpovědné za vykreslování oken (Surface Manager), databázovou podporu (SQLite), grafické knihovny (OpenGL, SGL, Freetype), mediální knihovny pro přehrávání audia a videa (Media Framework) anebo knihovny určené pro integrovaný webový prohlížeč (SSL, WebKit). Tyto knihovny existují pouze proto, aby byly volány z vyšších vrstev, samostatně fungovat nemohou.

- **Android Runtime** – Další vrstva navazující na linuxové jádro obsahující aplikační virtuální stroj zvaný Dalvik a také základní knihovny, které poskytují většinu funkcí dostupných v základních knihovnách jazyka Java.

  Virtuální stroj Dalvik – Jedná se o jednu z klíčových součástí celého Android OS (v dalším textu jen "Android"). Místo toho, aby Android využíval klasický virtuální stroj od Javy, jako je například Java ME (Java Mobile Edition), využívá svůj vlastní virtuální stroj, který je více optimalizován pro běh na mobilních zařízeních. Hlavním rozdílem oproti tradiční Javě je ten, že virtuální stroj Dalvik spouští `.dex` soubory, které jsou při kompilaci vytvořeny ze standardních `.class` a `.jar` souborů. Tyto `.dex` soubory jsou kompaktnější a efektivnější na mobilních zařízeních, které mají slabší procesor, limitovanou paměť a napájení na baterii. Výsledkem této optimalizace může být současný běh více instancí virtuálního stroje v jednu chvíli.

  Tato vrstva je tedy pohonem všech aplikací a spolu s knihovnami utváří pracovní prostředí pro další vrstvu.

- **Application Framework** – Tato vrstva poskytuje třídy použitelné při programování aplikací. Poskytuje také abstrakci pro přístup k hardwaru, spravuje uživatelské rozhraní a aplikační zdroje. Mezi významné části patří:

  - Activity Manager – část řídící životní cyklus aplikací, více v kapitole 2.1.
  - Views – komponenty používané při konstrukci uživatelského rozhraní aplikace. Mohou to být různé mřížky, seznamy, textová pole, tlačítka nebo také například vložený prohlížeč.
  - Notification Manager – umožňuje aplikacím zobrazovat různá upozornění v stavovém řádku.
  - Content Providers – umožňují aplikacím sdílet data, více v následující kapitole.
  - Resource Manager – podporuje přístup k externím zdrojům jako je grafika, či textové řetězce.

- **Application** – V této vrstvě nalezneme jak vestavěné aplikace, tak aplikace třetích stran. Obě tyto skupiny jsou si rovnocené, jelikož využívají stejné API knihovny. Samotnou aplikací tedy může být například správce kontaktů, sms, webový prohlížeč, nebo i hra.

Základní komponenty

Během programování aplikace pro systém Android se setkáváme se čtyřmi hlavními objekty [11]:

- **Activities** – Aktivita je blok vlastního uživatelského rozhraní, ta část, kterou uživatel vidí na obrazovce. Můžeme si ji představit jako analogii okna v klasických počítačových aplikacích.
- **Content Providers** – Poskytují abstraktní vrstvu nad jakýmikoliv daty uloženými na zařízení tak, aby byla jednoduše dostupná pro další aplikace. Programátor se nemusí starat, jak jsou data fyzicky uložena, pracuje pouze s handlerem a *content provider* se postará o správné provedení požadované operace, ať už se jedná o data uložená v SQLite databázi, klasické soubory na zařízení či soubory na internetovém serveru.
- **Intents** – Jsou systémové zprávy oznamující aplikacím výskyty určitých událostí. Může se jednat například o hardwarové změny (vložení SD karta), příchozí data (příchod SMS zprávy), nebo události aplikací (spuštění aplikace z hlavního menu). Programátor však není omezen pouze na naslouchání a reagovaní na zprávy, může také vytvářet vlastní. Je možné například spouštět jiné aplikace, nebo dát vědět o určité nastalé situaci.
- **Services** – Služby jsou na rozdíl od předešlých objektů navrženy tak, aby běžely dlouhodobě a nezávisle na jiných aktivitách. Může se např. jednat o přehrávání hudby, přestože byl samotný přehrávač zavřen a jeho aktivita tedy již neexistuje.

Vestavěné funkce

Při programování v Android SDK můžeme využít několik vestavěných funkcí, které nám pomohou snadněji vyvíjet aplikace [11]:

- **Uložiště** – Umožňuje aplikaci využít jak uložiště telefonu, tak například SD kartu pro ukládání nebo čtení souborů.
- **Síť** – Zařízení Android jsou typicky schopné připojení k Internetu. Toho můžeme využívat bohatě, od surových Java socketů až po vložený webový prohlížeč do aplikace.
- **Multimedia** – Ačkoliv má většina zařízení schopnost přehrát či zachytit zvuk a video, možnosti se mohou u jednotlivých zařízení různit. Například jen některé zařízení disponují přední kamerou. Nejprve se vytvoří dotaz na možnosti zařízení a podle odpovědi je lze dále v aplikaci využít.
- **GPS** – Jelikož má většina zařízení s Androidem GPS modul, je možné jej využít např. pro zobrazení pozice na mapě, nebo pro sledování pohybu telefonu.
- **Služby telefonu** – Zařízení s Androidem jsou z velké většiny mobilní telefony. Tyto funkce umožňují aplikacím spravovat hovory, zasílat a přijímat SMS zprávy a všechny ostatní telefonické funkce.

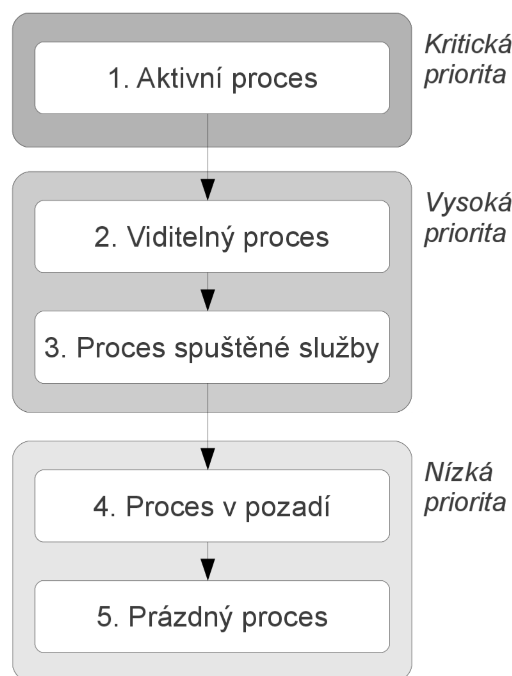
Správa aplikací a jejich životní cyklus

Na rozdíl od klasických operačních systémů, aplikace v systému Android má omezenou kontrolu nad vlastním životním cyklem. Aplikace musí neustále naslouchat změně svého stavu a být vždy připravena na náhlé ukončení.

Standardně obsahuje každá aplikace svůj vlastní proces, který vytváří vlastní instanci virtuálního stroje Dalvik. O správu systémových zdrojů se ale stále stará samotný Android za účelem zaručení neustálé odezvy zařízení. Ve výsledku to může znamenat, že proces může být bez varování násilně ukončen, aby systém uvolnil zdroje pro aplikace s vyšší prioritou.

Existuje 5 stavů procesu (obr. 2.2) [10]:

- **Aktivní** (v popředí) – proces, který je zrovna zobrazen na popředí a uživatel s ním pracuje, má samozřejmě nejvyšší prioritu. Jedná se o proces, který se systém bude snažit za každou cenu udržet odezvyschopný. Bude ukončen pouze pokud bude požadovat více paměti, než kolik je pro zařízení dostupné.
- **Viditelné** – proces, který je viditelný, ale ne na popředí. Může se jednat např. o proces za dialogem, který je v popředí. Standardně nebude ukončen, pokud to není zapotřebí pro bezproblémový běh procesu v popředí.
- **Spuštěné služby** – proces, který je běžící službou. Služby pracují v pozadí bez přímého viditelného rozhraní. Protože služby nejsou v přímé interakci s uživatelem, mají mírně menší prioritu nežli viditelné procesy. Jelikož jsou služby stále důležité, nebudou ukončeny, pokud to nebude nezbytně nutné pro aktivní nebo viditelné procesy.
- **V pozadí** – je proces, který není viditelný uživateli a byl pozastaven. Tento proces není kritický a může být ukončen, pokud jsou potřeba zdroje pro viditelné a aktivní procesy. Proces si musí před ukončením uchovat svůj stav pro případ, kdy uživatel zmáčkne tlačítko zpět a bude očekávat, že aplikaci nalezne v původním stavu.
- **Prázdné** – proces, který neobsahuje ani službu, ani žádnou aktivitu. Pro vyšší výkon Android často nechává proces v paměti i po tom, co ukončil svůj životní cyklus. Tato optimalizace zrychluje znovu spuštění aplikace. Tyto procesy jsou v případě nedostatku paměti ihned ukončeny.



Obrázek 2.2: Stavy procesů.

2.2 Tvorba aplikací pro Android OS

Tato část se zabývá principy a základy tvorby aplikací pro Android.

Android SDK

Nástroje pro vývoj aplikací pro Android jsou obsaženy v SDK, který je dostupný pro řadu systémů. Nejjednodušší je jeho použití pomocí modulu do Eclipse, který nalezneme pod zkratkou ADT (Android Development Tools) [12]. Je ale možné používat také příkazovou řádku SDK.

Výhodou SDK je implementace emulátoru zařízení s Androidem, což nám umožňuje otestovat většinu funkčnosti aplikace i bez reálného telefonu [13]. Logicky má emulátor i svá omezení, jako je reálné použití kamery, telefonování, simulace baterie apod. Velkou výhodou ale může být otestování námi vytvořené aplikace pro různé verze systému, či různé rozlišení a velikosti obrazovky a to bez nutnosti vlastnit několik fyzických zařízení. Samotné programování probíhá v Javě s využitím Android API a několika specifických souborů XML.

Manifest

Základem každé aplikace pro Android je soubor `AndroidManifest.xml`, uložený v hlavní složce projektu. Nejzákladnější informací v manifestu je atribut `package` kořenového prvku `manifest`, který určuje název balíku. V manifestu je možné specifikovat spousta dalších věcí, kde mezi nejdůležitejší patří [11]:

- **Oprávnění** – Aby aplikace mohla používat některé funkce systému, jakožto např. telefonování, fotoaparát, zjištění polohy pomocí GPS, nebo aby získala přístup ke kontaktům, musí být v manifestu nastavena příslušná povolení. Tyto specifikovaná povolení pak vidí uživatel aplikace při instalaci softwaru a souhlasem stvrdí, že tyto funkce či zdroje může aplikace používat.

  Příkladem může být povolení používání fotoaparátu:

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

- **Minimální verze systému** – Android, stejně jako každý jiný systém, prochází určitým vývojem a jednotlivé jeho verze přinášejí vylepšení i změny. Některé z těchto změn mají vliv na samotné SDK a přinášejí tak například nové třídy, metody a parametry nedostupné v předchozích verzích. Někdy tak může být účelné zajistit, aby aplikace byla spustitelná pouze na systému s určitou minimální verzí SDK.

  Následujícím řádkem nastavíme minimální verzi 8 (odpovídá Androidu verze 2.2):

  ```
  <uses-sdk android:minSdkVersion="8" />
  ```

- **Specifikace aplikace** – Samotným jádrem manifestu je element `<application>`, kde specifikujeme samotné součásti aplikace, jako jsou aktivity `<activity>`, služby `<service>`, content providery `<provider>` a další. Dále je zde možné nastavit např. ikonu a název aplikace zobrazované v menu zařízení.

  V nejdůležitějším elementu samotné aktivity nastavujeme jméno třídy (parametr `android:name`), zobrazované jméno aktivity (parametr `android:label`) a často také podřízený prvek `<intent-filter>`, který specifikuje, za jakých podmínek bude aktivita zobrazena.

Příkladem může být specifikace aplikace, která má jednu aktivitu, zobrazenou při spuštění:

```
<application>
   <activity android:name=".Now" android:label="Now">
      <intent-filter>
         <action android:name="android.intent.action.MAIN" />
         <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
   </activity>
</application>
```

Tvorba grafického rozhraní

Ačkoli je možné vytvářet a spojovat prvky grafického rozhraní čistě pomocí Java kódu, existuje ve vývoji Android aplikací i jiný, výhodnější způsob. Tím je specifikace rozvržení pomocí XML souboru [11]. V tomto souboru se definují jednotlivé prvky a hierarchické vazby mezi nimi, pomocí formátu XML. Nespornou výhodou je fakt, že prostředí Eclipse obsahuje grafický editor, pomocí kterého můžeme jednoduše vytvářet a umisťovat jednotlivé elementy a generuje se nám přehledné XML, které je lehce čitelné a upravitelné.

Takovýto XML soubor je v projektu považován za zdroj a je v psaném kódu přístupný pomocí souboru `R.java`, který je v projektu automaticky generován. Pomocí něj můžeme přiřadit dané rozhraní požadované aktivitě, a také přistoupit k jednotlivým prvkům rozhraní a pracovat s nimi (reagovat na stisk apod.).

Lokalizace aplikace

Tvorba lokalizovaných aplikací je v systému Android velice jednoduchá. Systém využívá tzv. *Resource-Switching* [12], kdy jsou aplikaci automaticky vybrány zdroje, které nejlépe vyhovují danému zařízení a jeho nastavení. V praxi to znamená, že můžeme mít připraveny např. rozdílné obrázky pro různá rozlišení zařízení, jiná rozvržení pro svislou a horizontální polohu, lokalizované řetězce pro různé jazyky a mnoho dalších variací.

Samotná lokalizace se týká řetězců uložených v souboru `/res/values/strings.xml`. Například české jazykové verze dosáhneme vytvořením složky `/res/values-cs` a umístěním přeloženého souboru `strings.xml` do této složky [13]. Jakmile je v mobilním telefonu nastaven český jazyk, aplikace bude po spustění automaticky s uživatelem komunikovat česky. V případě nastavení jakéhokoliv jiného jazyka budou použity výchozí řetězce, typicky lokalizované do angličtiny. Přizpůsobit musíme také samotnou aplikaci a v jejím zdrojovém kódu nepoužívat konstantní řetězce, ale využívat výhradně řetězce ze zdrojů. Podobným způsobem je možné pracovat s jinými typy zdrojů. Např. odlišné rozvržení pro horizontální natočení umístíme do složky `/res/layout-land`.

2.3 Práce s fotoaparátem

Fotoaparát lze v uživatelské aplikaci využívat dvěmi způsoby [10]:

- **Pomocí Intent** – nejjednodušší způsob jak získat obrázek ze zabudovaného fotoaparátu. Z uživatelské aplikace se pomocí zprávy (Intent) spustí aktivita kamery, kde si uživatel může měnit jakékoliv nastavení. Po potvrzení vyfocené fotografie je obraz předán zpět do programu. Získaný obrázek pak můžeme v aplikaci přímo využít jako bitmapu, nebo pomocí metody `MediaStore.EXTRA_OUTPUT` uložit originální fotografii na specifikované umístění.

- **Ovládáním kamery** – abychom mohli přistupovat ke kameře přímo, potřebujeme nejprve přidat oprávnění do souboru manifestu, viz kap. 2.2. V aplikaci poté používáme objekt `Camera`, u kterého programově specifikujeme nastavení kamery, a pořizujeme obrázky. Nevyvoláváme tedy nativní aktivitu kamery jako v prvním případě, ale všechny akce ovládáme sami. Hlavní výhodou tohoto přístupu je integrace funkce fotoaparátu přímo do aplikace, tedy zobrazení náhledu do námi určeného prostoru a snímání obrázků na základě stisku námi specifikovaných tlačítek. Vyfocená fotografie je pak v aplikaci reprezentována jako pole bytů se kterým můžeme dále pracovat.

2.4 Převod textu na zvuk

Již od verze 1.6 Android obsahuje modul pro převod textu na zvuk s názvem Pico [13]. Ten dovoluje jakékoliv aplikaci přečíst textový řetězec na základě zvoleného nastavení. V nastavení modulu je možné vybrat jazyk, pomocí kterého bude čtení prováděno a také rychlost tohoto čtení. Pro samotné použití je také nutné nainstalovat zvolené slovníky z internetu.

V samotné aplikaci využijeme funkci převodu textu na zvuk pomocí dodaného API a objektu třídy `android.speech.tts.TextToSpeech`. Před samotným použitím převodu v aplikaci, musíme zkontrolovat, zda dané zařízení obsahuje zdroje potřebné pro samotný převod (zejména jazykové soubory). Tuto kontrolu provedeme spuštěním předdefinované aktivity s akcí `TextToSpeech.Engine.ACTION_CHECK_TTS_DATA`. Výsledkem této aktivity je informace, zda jsou zdroje v pořádku, či nikoliv. Pokud zdroje chybí, spustíme předdefinovanou aktivitu instalace těchto zdrojů. V případě, že jsou zdroje přítomny, můžeme provést samotný převod následujícími příkazy:

```
TextToSpeech mTts = new TextToSpeech(context, listener);
mTts.speak("Sample text", TextToSpeech.QUEUE_ADD, null);
```

Zadaný text je poté přečten jazykem, který byl vybrán v nastavení. Nástroj Pico však bohužel obsahuje jen několik západních jazyků: angličtinu, němčinu, francouzštinu, italštinu a španělštinu. Český text převedený například pomocí anglických slovníků bude tedy velice zkreslený. K zakoupení jsou ale dostupné i jiné komerční moduly pro převod textu na zvuk a některé z nich obsahují právě i češtinu. Výběr takového modulu a případného českého jazyka pak probíhá v globálním nastavení zařízení a samotná tvorba aplikací využívajících převod se vůbec nezmění.

Kapitola 3

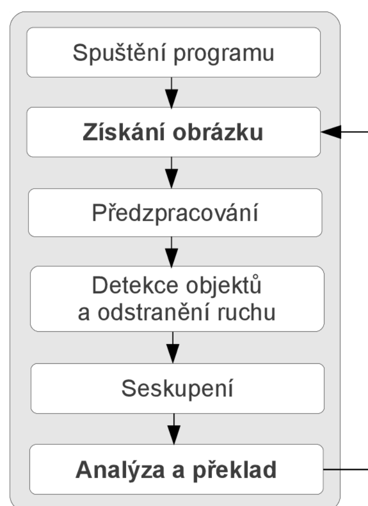
Návrh rozpoznání Braillova písma v obraze

Většina publikovaných vědeckých článků, týkajících se rozpoznání Braillova písma, se zabývá rozpoznáním celých stran písma vytlačeného v papíru, tedy obvykle knih pro zrakově postižené. Obrázek celé strany písma je obvykle získán pomocí scanneru [14, 15]. Takto získaný obraz je dobře a rovnoměrně osvětlen, není obvykle nesprávně natočen a neobsahuje tolik ruchů. Tyto vlastnosti umožňují uplatnit jednodušší formy předzpracování, nežli je tomu u fotografie veřejného prostranství pořízené mobilním telefonem. Existují také fixní normy vytlačování Braillova písma do papíru a tak je na základě znalosti DPI možné určit absolutní velikost znaků a mezer mezi nimi [16]. Programy popsané v těchto pracech jsou také navrhovány pro použití na počítači, a tak si mohou dovolit použít časově a prostorově náročnější metody. Oproti fotografii Braillova písma na veřejném prostranství, jsou zde jiné problémy a to především s oboustrannými dokumenty. Tyto algoritmy musí rozhodnout, zdali se jedná o tečku vytlačenou, či vystouplou [17].

Většina principů popsaných v těchto publikacích však není uplatnitelná v našem případě. Naopak zde musíme řešit spoustu jiných problémů. Při návrhu použitých metod musíme brát ohled na to, že získáváme obrázek z mobilního telefonu a veřejného prostranství. Není tak výjimkou, že získaná fotografie není řádně zaostřená, typicky není orientována naprosto rovně a může být vyfotografována ve špatných světelných podmínkách. Získané tečky a mezery mezi nimi jsou také pokaždé jinak velké.

Jediným nalezeným zdrojem, který popisuje přesně tuto problematiku, je článek *A Braille Recognition System by the Mobile Phone with Embedded Camera* [18]. Následující návrh jednotlivých kroků rozpoznávání byl tedy inspirován především tímto zdrojem. Článek ovšem obsahuje pouze základní myšlenky, a tak jsou jednotlivé popsané postupy a algoritmy z valné většiny vlastní.

Samotné rozpoznávání se skládá z několika kroků. Představme si jejich základní princip, přičemž podrobnosti budou popsány v této kapitole níže.



Obrázek 3.1: Diagram fází aplikace.

Prvotním úkonem aplikace je samotné získání obrázku. Před detekcí jednotlivých teček ze získaného obrazu, musíme fotografii nejprve předzpracovat. Pro rychlejší práci s obrazem jej nejprve převedeme do stupňů šedi. Tečky, které reprezentují Braillové znaky typicky barevně vyčnívají z pozadí, jsou buďto vyvedeny jinou barvou, nebo zachytíme jejich odlesk či stín. Pro jejich odlišení aplikujeme na obrázek prahování. V samotném obrázku však může být kromě teček Braillova písma i celá řada jiných ruchů, například text, nečistoty nebo odlesky. Některé z nich můžeme rozpoznat a před dalším zpracováním odstranit.

V další fázi již tedy máme sadu teček a ty seskupíme do jednotlivých řádků a sloupců. Na základě umístění v řádcích a sloupcích následně sestavíme jednotlivé znaky a ty poté přeložíme na souvislý text. Překlad číselné hodnoty na znak závisí na použitém jazyku. Po dokončení je možné zpracovat další fotografii. Celý průběh je demonstrován v diagramu 3.1.

Dále v této kapitole budou podrobněji popsány konkrétní postupy těchto fází, směřující k rozpoznání Braillova písma v obraze. Jednotlivé metody jsou prokládány demonstračními obrázky, které znázorňují jejich funkci. Kapitola je členěna do jednotlivých bloků rozpoznávání a tyto bloky obsahují popis jednotlivých rozpoznávacích technik.

3.1 Předzpracování obrazu

Předzpracování obrazu patří mezi nejdůležitější části spolehlivého rozpoznávání. Pokud je během tohoto procesu ztracena nějaká informace, následující kroky ji již neobnoví. Proto je důležité zvolit takové metody, které rozpoznají maximum teček Braillova písma v obraze a to nejlépe tak, aby při nich vzniklo co nejméně rušení a tečky by tak mohly být dále snadno zpracovány. V této části se seznámíme jak s obecnými principy předzpracování, tak s konkrétními metodami použitými v této práci.

Převod do odstínů šedi

Jelikož klasické metody předzpracování počítají s šedotónovým obrazem a barva pro nás v tomto případě nemá velkou informační hodnotu, obraz nejprve převedeme do odstínů šedi podle vztahu [19]:

$$I = 0.299 * R + 0.587 * G + 0.144 * B \tag{3.1}$$

Kde R, G a B reprezentují základní barevné složky a I dává výslednou hodnotu šedi.

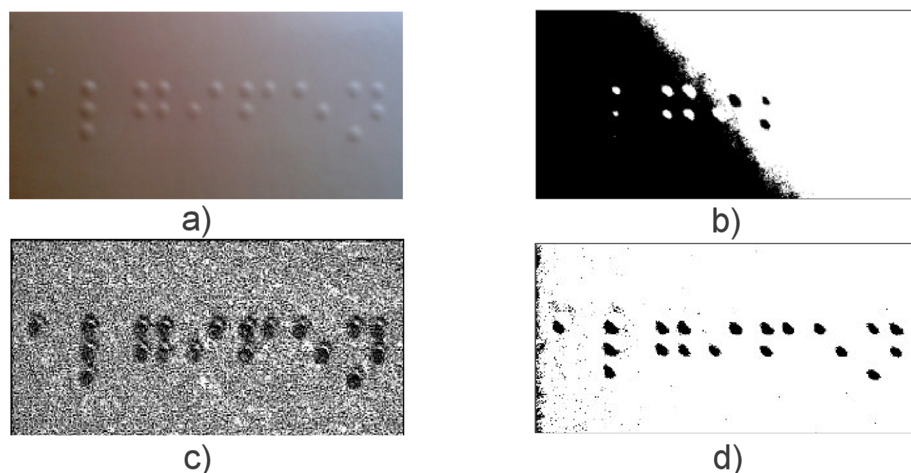
Prahování

Tečky Braillova písma jsou vždy viditelně odlišitelné od svého pozadí, ať už jsou vyvedeny jinou barvou, nebo zachytíme jejich stín či odlesk. Fakt, že jsou viditelné, znamená, že mají tečky od svého pozadí odlišnou hodnotu jasu.

Prahování se používá k převedení obrazu s více úrovněmi jasu na obraz se dvěmi úrovněmi jasu (černá a bílá). Pro každý pixel se vyhodnotí, zda je jeho hodnota jasu nižší či vyšší než zadaný práh a podle toho je mu přiřazena hodnota 0, či 1 (případně 255). Prahování může být globální, nebo lokální [20]:

- **Globální prahování** – u tohoto způsobu prahování je zvolena nebo nalezena určitá hodnota prahu T a poté se sekvenčně prochází celý obraz. Pokud platí $f(x, y) \geq T$, je pixelu přiřazena 1, jinak 0. Práh je možno stanovit různými způsoby. Je možné zkoušením vybrat hodnotu, která dává nejlepší vizuální výsledek, ale k tomu je potřeba uživatelova aktivita, což je v tomto případě nepřípustné. Budou nás zajímat metody, které jsou schopny určit práh automaticky. K tomuto slouží několik metod, například *Otsuova metoda* [21]. Jak ale můžeme vidět na obr. 3.1 b), globální prahování není vhodné v případě nerovnoměrného osvětlení či rozdílných hodnot jasu v různých částech obrazu.

- **Lokální prahování** – neboli také adaptivní či dynamické prahování. Hlavní výhodou je větší úspěšnost při prahování snímku s nerovnoměrným osvětlením, viz obr. 3.1. Princip lokálního prahování spočívá v tom, že je postupně zkoumáno okolí každého bodu a hodnota prahu T je vždy vypočtena pro toto okolí. Okolí je určeno maskou o určité velikosti, která se pohybuje po obraze. Pro určení hodnotu prahy v tomto okolí je možné využít statistické funkce nebo například již zmíněnou *Otsuovu metodu*. Výsledek prahování je pak závislý na zvolené velikosti masky. Do této skupiny patří například metoda *p-procentního prahování* [22], *Wellnerova metoda* [23], nebo její modifikace s využitím integrálního obrazu [24].



Obrázek 3.2: Různé způsoby prahování. a) originální obrázek. b) globální prahování s výpočtem prahu pomocí Otsuovy metody. c) lokální prahování metodou p-procentního prahování. d) lokální prahování Wellnerovou metodou s použitím integrálního obrazu.

V našem případě, kdy je fotografie z mobilního telefonu často nerovnoměrně osvětlená, použijeme zajisté *Lokální prahování*. Konkrétně použijeme modifikaci Wellnerovy metody s využitím integrálního obrazu [24]. Jak můžeme vidět na obr. 3.1 d), tato metoda dosahuje nejlepších výsledků.

Lokální prahování s využitím integrálního obrazu

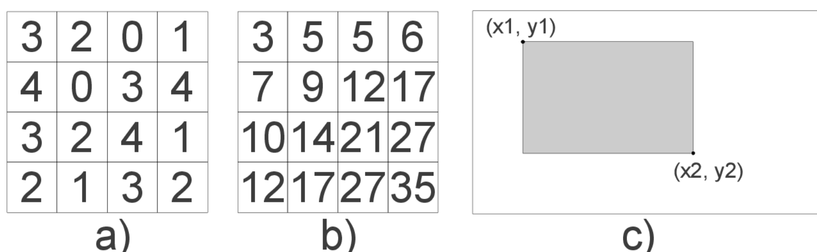
Tato metoda byla publikována ve článku *Adaptive Thresholding Using the Integral Image* [24]. V této práci si popíšeme základní princip této metody a její výhody.

Integrální obraz je technika, pomocí které jsme schopni efektivně spočítat sumu hodnot pixelů v určité oblasti obrazu. Tuto sumu můžeme rovněž spočítat bez integrálního obrazu, procházením a sčítáním všech bodů v dané oblasti. V případě adaptivního prahování však potřebujeme počítat okolí každého pixelu a to by bylo klasickým způsobem velmi pomalé. Pokud bychom aplikaci provozovali na klasickém počítači, časový dopad by při dnešních výkonech procesorů nebyl tak velký. Naše aplikace je ale navrhována pro mobilní telefon s často slabým procesorem, a tak musíme dbát také na efektivnost metod. Z toho důvodu použijeme namísto klasického přístupu integrální obraz, pomocí kterého dokážeme spočítat sumu oblasti s konstantním počtem operací nezávisle na její velikosti a to pouze s lineární složitostí předzpracování.

Předzpracováním se myslí výpočet integrálního obrazu z dané šedotónové fotografie. Výpočet pro každý bod, ilustrován na obr. 3.3, probíhá podle následujícího vzorce:

$$I(x, y) = f(x, y) + I(x - 1, y) + I(x, y - 1) - I(x - 1, y - 1). \tag{3.2}$$

Kde $I(x, y)$ je hodnota integrálního obrazu a $f(x, y)$ úroveň jasu šedi v daném bodě.



Obrázek 3.3: Integrální obraz. a) vstupní matice. b) vypočítaný integrální obraz. c) použití integrálního obrazu k výpočtu sumy šedé oblasti.

Když máme vypočítaný integrální obraz, suma jakéhokoliv obdélníku s krajními body naznačenými v obr. 3.3 c), může být v konstantním čase spočítána pomocí následující rovnice:

$$\sum_{x=x_1}^{x_2} \sum_{y=y_1}^{y_2} f(x, y) = I(x_2, y_2) - I(x_2, y_1 - 1) - I(x_1 - 1, y_2) + I(x_1 - 1, y_1 - 1) \tag{3.3}$$

Zde vidíme, že suma jakkoliv velké oblasti, se na rozdíl od klasického procházení všech pixelů, spočítá pouze pomocí tří operací sčítání a odčítání, což je velká časová úspora. Je sice nutné nejprve z fotografie vypočítat integrální obraz, to však znamená jen jeden průchod navíc přes všechny pixely. Z důvodů optimalizace je dokonce možné počítat integrální obraz již v prvním průchodu, kdy fotografii pixel po pixelu převádíme do odstínů šedi. Získanou sumu následně použijeme pro výpočet průměrné hodnoty dané oblasti.

Prahování samotné je založeno na Wellnerově metodě [23], která prahuje obrázek pomocí jediného průchodu. Ta v koncepci porovnává každý pixel s průměrnou hodnotou jeho okolí. Pokud je hodnota pixelu t procent menší než průměr, je nastavena hodnota černé (0), jinak bílé. Okolí u Wellnerovy metody je bráno jako s naposledy zpracovaných pixelů, což může být nepřesné, jelikož není uvažováno okolí do všech směrů od pixelu. Právě zde využijeme výhodu integrálního obrazu a budeme počítat průměr oblasti velké $s*s$ pixelů, se středem ve zkoumaném pixelu. Tímto do průměru zahrneme okolí pixelu ze všech stran. Velikosti hodnot t a s záleží na konkrétní aplikaci a pro tuto práci byly zvoleny experimentálně na základě několika testovacích obrázků.

Optimalizace - jelikož se zabýváme návrhem aplikace určené pro mobilní telefon, je pro nás žádoucí, aby byly metody co nejlépe optimalizovány. Hlavním nedostatkem této metody je nutnost procházení celého obrázku dvakrát. Jednou pro výpočet integrálního obrazu, podruhé pro samotné prahování. Počet kroků předzpracování fotografie je tedy celkově $2 * width * height$.

To je však možné vylepšit. Můžeme provést prahování pro pixel $(x - s/2, y - s/2)$, ve stejném kroku jako výpočet integrálního obrazu v bodě (x, y) [24]. Tímto bude výpočet prováděn pro následující počet pixelů:

$$(w * h) + (\frac{s}{2} * w) + (\frac{s}{2} * h) - (\frac{s^2}{4}) \tag{3.4}$$

Kde w reprezentuje šířku fotografie a h její výšku.

Pro obrázek velikosti $600*450$ px a s rovnu 40 se jedná o cca 1,86x méně operací. Jelikož je předzpracování jedna z nejvíce časově náročných fází navrhovaného programu, jedná se o znatelné zrychlení celého překladu.

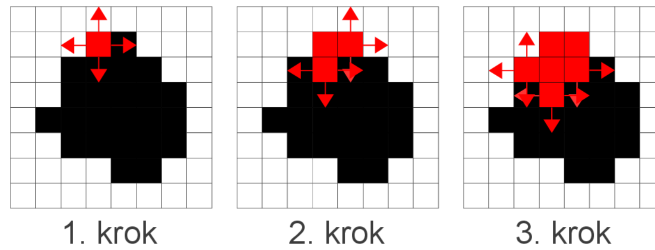
3.2 Detekce objektů a odstranění ruchu

Po uplatnění prahování, je nutné v obraze detekovat objekty, které byly prahováním nalezeny. Při detekci objektů je ale detekováno kromě opravdových teček také mnoho ruchů. Může se jednat například o odlesky, text či jiné nečistoty v obraze. Pro zjednodušení analýzy teček Braillova písma je nutné tyto ruchy odstranit. V této části se budeme zabývat konkrétními technikami pro detekci objektů v obraze a jejich klasifikací jako potenciální tečku či ruch. Jako vstup je uvažován binární obraz, kde barva pozadí je černá (hodnota 0) a barva potenciálních objektů bílá (hodnota 1, resp. 255).

Detekce objektů

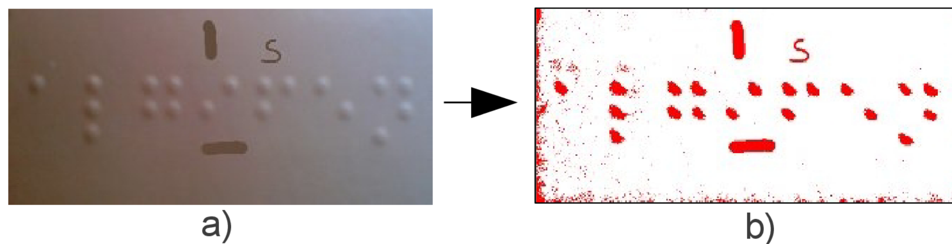
Ze všeho nejdříve musíme v obraze detekovat shluky pixelů bílé barvy, tedy potenciální objekty. Pro identifikaci spojených oblastí v obraze je možné použít algoritmus *barvení* [25]. Cílem tohoto algoritmu je pomocí dvou průchodů celého obrazu opatřit každou spojitou oblast neopakujícím se přirozeným číslem. Dalšími technikami pak musíme získat informace o tvaru těchto identifikovaných oblastí, o jejich velikosti apod.

V této práci ale volíme jiný přístup. Využijeme algoritmu semínkového vyplňování [19], který se primárně využívá na obarvování spojitých rastrových oblastí. V počátečním pixelu (semínku) se zkontroluje, zdali je barva rozdílná od pozadí a pokud ano, pixel se obarví a algoritmus se rekurzivně spustí na sousední pixely ve čtyř-okolí. Princip je názorně zobrazen na obr. 3.4.



Obrázek 3.4: Princip semínkového vyplňování.

Pro detekci objektů postupně po řádcích procházíme obraz, a pokud narazíme na pixel bílé barvy (okraj shluku pixelů), spustíme algoritmus semínkového vyplňování. Algoritmus postupně projde všechny pixely shluku a obarví je na černo. Tím objekt zmizí z obrazu, což je nezbytné k tomu, aby nebyl shluk znovu analyzován při průchodu dalším řádkem. Naším cílem však není pouze obarvování shluků, nýbrž počítání počtu pixelů oblastí a také výpočet minimální a maximální x a y hodnoty shluku. Tyto hodnoty jsou získávány již během samotného obarvování a následně slouží k výpočtu šířky a výšky shluku a také k určení pozice jeho středu. Na obr. 3.5 můžeme vidět, že bylo detekováno i spousta šumu a ruchů. V dalších krocích je popsáno, jak se jich zbavit.



Obrázek 3.5: Výsledek detekce objektů. a) originální obrázek. b) objekty (červeně) po detekci.

Jak můžeme vidět, oproti dvouprůchodovému algoritmu *barvení*, popsaného v prvním odstavci, semínkové vyplňování prochází celý obraz jen jednou. Počet operací samozřejmě naroste každým "zasazením semínka"a spuštěním samotného algoritmu obarvování oblasti. Pokud však uvážíme, že je po prahování většina obrazu vyvedena černou barvou a je zde jen několik objektů, které jsou obarvovány, jedná se o značnou časovou úsporu. Navíc jsme schopni během operace barvení přímo počítat počet pixelů daného shluku a další potřebné informace. To při sekvenčním procházení u algoritmu *barvení* není možné.

Jelikož algoritmus semínkového vyplňování využívá rekurzi, hrozí zde přetečení programového zásobníku při vyplňování velkých oblastí. Řešením může být nastavení určité meze počtu pixelů, které algoritmem zpracováváme. Po dosáhnutí této meze se algoritmus ukončí a zbytek shluku je analyzován dalším posunem v obraze. Zde se však může objevit problém u velkých oblastí, které jsou takto rozděleny do více částí. Vzhledem k tomu, že je aplikace navrhována pro mobilní telefon, nemůžeme si dovolit dát mez příliš velkou a může se tedy stát, že by mohly být rozděleny i velké kandidátní tečky. Lepší variantou je nahrazení rekurze zásobníkem, kdy dokážeme detekovat libovolně velké objekty a tím detekci zpřesnit.

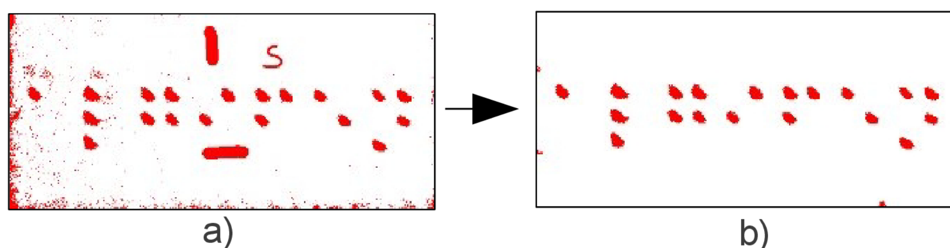
Podmínky objektů

Ve chvíli, kdy skončí algoritmus vyplňování, máme k dispozici počet pixelů každého zkoumaného shluku, jeho šířku a výšku. Již v této fázi můžeme na základě těchto informací některé shluky vyřadit a prohlásit je za ruch. Jsou tři základní typy ruchů:

1. Zaprvé se jedná o příliš malé či velké objekty (např. jednopixelové ruchy či kus špatně vyprahovaného pozadí). Stanovíme zde mez, mezi kterou se musí pohybovat počet pixelů shluku.

2. Dále se jedná o shluky, které jsou moc široké či vysoké (např. šmouhy, některá písmena, číslice). Zde vypočteme poměr mezi šířkou a výškou a opět určíme mez, za kterou objekt prohlásíme za ruch.

3. Poslední podmínkou je poměr skutečného počtu pixelů k obsahu obdélníku, který shluk opisuje. Kupříkladu kruh, reprezentující tečku, opisuje čtverec. Ovšem například malé písmeno „s“ také opisuje zhruba čtverec. Zatímco obsah kruhu odpovídá cca 80% obsahu opsaného čtverce, u malého písmena „s“ je to cca 50%. Můžeme tedy určit mez, kterou musí překračovat poměr obsahu opsaného obdélníku a skutečného obsahu shluku, aby byl shluk považován za kandidátní tečku.



Obrázek 3.6: Uplatnění podmínek. a) všechny objekty po detekci. b) objekty vyhovující podmínkám.

Shluky, které na základě těchto podmínek nejsou vyhodnoceny jako ruch, přidáme do seznamu objektů, se kterým dále pracujeme. Jednotlivé meze jsou určeny experimentálním měřením s tím, že nejsou příliš striktní. Raději prohlásíme více ruchů za kandidáty, které jsme schopni vyřadit dále, než abychom označili reálné tečky za ruchy a tím dále způsobili chybu překladu. Na obr. 3.6 se můžeme přesvědčit, že jen pomocí jednoduchých podmínek, výrazně zredukujeme počet ruchů.

Podobnost

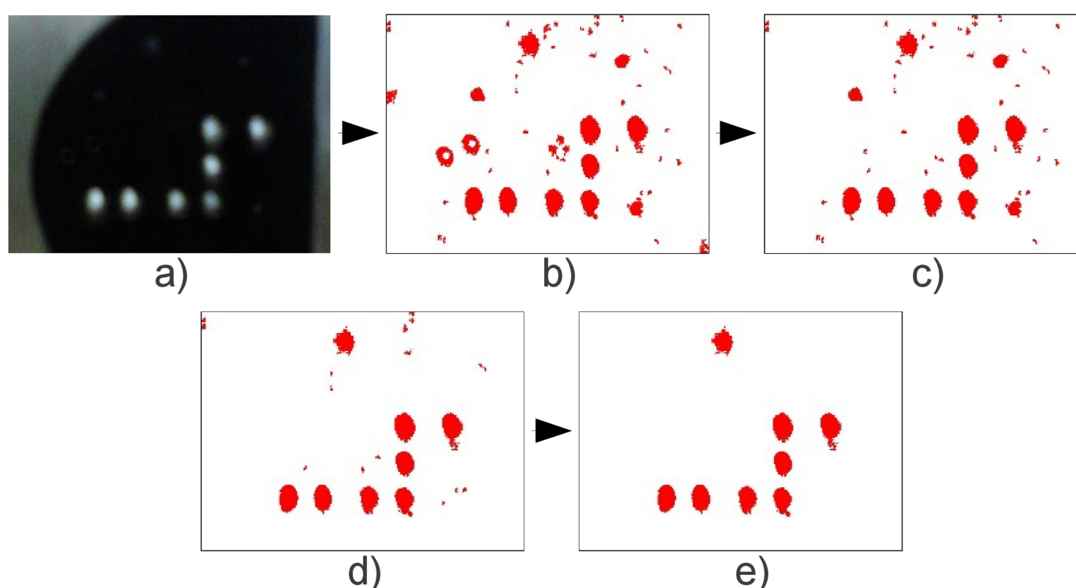
Reálně jsou všechny tečky Braillova písma v obraze stejně velké, a i když se jejich tvar může mírně měnit například na základě osvětlení, jsou si tvarově přibližně podobné. Dalším krokem odstranění ruchů je tedy eliminace objektů, které si nejsou podobné s ostatními objekty. V obraze může být velké množství objektů, které mají přibližně kruhový tvar a vyhovují všem podmínkám popsaným v předešlé sekci, i když se jedná o ruchy. Tyto ruchy jsou často různě velké a tak lze konstatovat, že pokud se v obraze nevyskytují alespoň další dva objekty podobné zkoumanému objektu, jedná se o ruch.

Pro zkoumání podobností postupně procházíme seznam objektů a kontrolujeme každý objekt. Kontrolou rozumíme druhý cyklus procházení seznamu a srovnávání aktuálního objektu s ostatními v seznamu. V druhém cyklu procházíme seznam pouze dále od aktuálního

objektu, jelikož objekty umístěné před aktuálním objektem již byly s tímto objektem srovnávány. Samotná podobnost se určuje opět mezí poměru počtu pixelů objektů a poměru šířek a výšek. Ve chvíli, kdy zjistíme, že je některý objekt s aktuálním podobný, zvýšíme počítadlo podobností u obou těchto objektů. Až se tedy bude zkoumat objekt, v tomto cyklu vyhodnocený jako podobný, bude již mít informaci o tom, že je jeden podobný objekt umístěný před ním. Pokud má aktuální objekt po prozkoumání počítadlo podobností menší než tři, znamená to, že objekt nemá v obraze dostatečný počet podobných objektů a bude smazán. Tímto přístupem je poměrně efektivně srovnán každý s každým objektem a v důsledku výrazně zredukován počet ruchů v obraze. Na obrázku 3.7 c) vidíme ukázku uplatnění této techniky.

Osamocené objekty

Po odstranění ruchů na základě podobností, mohou v obraze stále zůstat skupiny ruchů, které jsou si tvarem či velikostí podobné. Často však nejsou tyto skupiny podobných ruchů umístěny blízko u sebe. Tyto ruchy lze tedy identifikovat na základě faktu, že se tečky Braillova písma vyskytují vždy ve skupinách a tak můžeme prohlásit, že pokud se v určité vzdálenosti od objektu nenachází podobný objekt, jedná se o ruch.



Obrázek 3.7: Odstranění ruchů. a) originální obrázek. b) výsledek po uplatnění podmínek. c) po porovnání podobnosti. d) odstranění osamocených obj. e) odstranění malých obj.

Jelikož v předchozím kroku získáváme seznam podobných objektů ke každému objektu, můžeme postupně zkoumat seznam těchto podobných objektů a vyhodnotit, zda jsou k aktuálnímu objektu blízké. Pokud se zkoumaný objekt nachází v okolí aktuálního, inkrementujeme počítadlo blízkých objektů a to jak u aktuálního objektu, tak u objektu, který byl vyhodnocen jako blízký. Jako okolí objektu uvažujeme prostor pětinásobku šířky a pětinásobku výšky na každou stranu od středu zkoumaného objektu. Pokud aktuální objekt nemá po ukončení zkoumání žádný podobný objekt v jeho okolí, je smazán ze seznamu objektů. Tato metoda vykazuje výborné výsledky a redukuje počet ruchů až o polovinu, což je možné vidět na ukázkovém obrázku 3.7 d).

Malé objekty

Výstupem předchozího kroku je sada objektů, z nichž je většina teček. Mohou se ale objevit také ruchy, které jsou si podobné a jsou navíc umístěny ve skupině. Většinou se jedná o ruchy, které jsou k tečkám poměrně malé. Pokud by se jednalo o ruchy přibližně velké jako tečky, které prošly všemi filtry, je nutné je analyzovat. Pro odstranění malých objektů vypočteme průměrnou velikost oblasti všech objektů a odstraníme ty, které mají oblast menší než 60% průměru. Toto číslo bylo určeno experimentálně na základě několika testovacích obrázků. Jak vidíme na obr. 3.7 e), odstraněním malých objektů se nám obrázek vyčistí od zbývajících malých ruchů, které by nám při dalším zpracování mohly dělat problémy. V obraze se mohou vyskytovat také ruchy, které jsou větší než tečky, těchto je ale obvykle málo a jsme schopni je odstranit při seskupování.

Shrnutí

Výstupem těchto čtyř kroků je zredukovaný seznam objektů, které by měl obsahovat pouze kandidátní tečky. Mezi reálnými tečkami se může vyskytnou také ruch velice podobný tečce a v blízkosti teček (možno pozorovat na obr. 3.7 v horní části obrázku). Ten je ale možné odhalit až v následujících postupech.

3.3 Seskupení

V této části bude popsán proces seskupení teček do jednotlivých znaků Braillova písma. Zařazením teček do skupin odpovídajícím normám Braillova písma, nejen že získáme reálné znaky složené z teček, ale také se zbavíme případných ruchů, které do tohoto uspořádání nezapadají. Fotografie Braillova písma má tu nevýhodu, že jsou tečky od fotoaparátu vždy jinak vzdáleny a tím pádem jsou vždy jinak velké. Jsou také jinak velké, pokud se jedná o tečky odlišené od pozadí jinou barvou, nebo jen o zachycené odlesky tečky. S tím souvisí nemožnost zavedení určitých normalizovaných rozměrů a nutnost všechny vztahy určovat dynamicky. Především se pak jedná o rozličnost mezer mezi jednotlivými znaky a jejich sloupci.

Detekce řádků

Ze všeho nejdříve seskupíme tečky do jednotlivých řádků. Seskupení je možné provést na základě vertikální pozice objektů (souřadnice y). Jelikož detekce objektů probíhala zpracováním pixelů obrazu postupně po řádcích, v seznamu objektů jsou objekty seřazeny od toho umístěného nejvýše po ten nejníže, tedy přesně podle souřadnic y. Postupně tedy procházíme seznam všech objektů (v této chvíli již kandidátních teček) a počítáme rozdíl y souřadnic mezi aktuálním a předchozím objektem. Pokud je absolutní hodnota rozdílu menší než polovina výšky objektu, jsou tyto dvě tečky ve stejném řádku. Pokud tomu tak není, je aktuální řádek ukončen a tečka je přidána do řádku nového.

Řádek je reprezentován seznamem teček a ukončením se rozumí seřazení teček podle jejich hodnoty x, výpočet výšky a pozice řádku a jeho přidání do seznamu řádků. Výpočet výšky a pozice řádku je proveden průměrem těchto hodnot u první a poslední tečky v seznamu řádku.

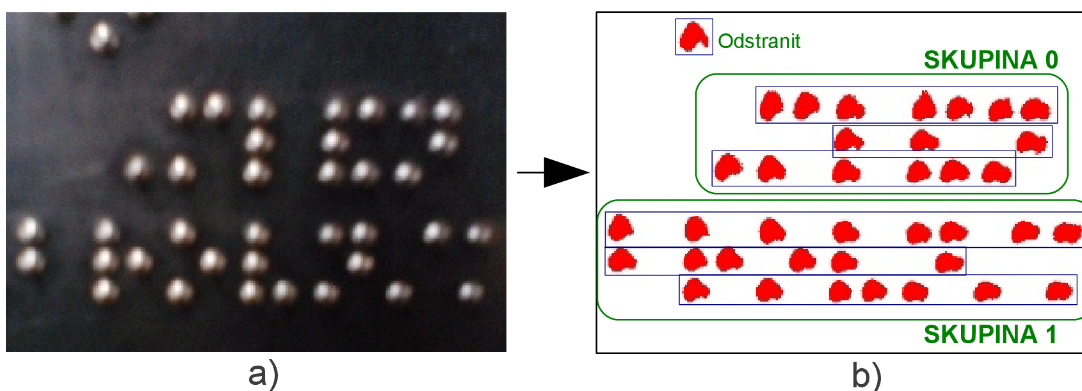
Seskupení řádků

Cílem dalšího kroku je seskupení detekovaných řádků do skupin po třech řádcích, které spolu reprezentují řádek znaků Braillova písma. Mimo jiné tímto postupem chceme odstranit řádky ruchů, které jsou tvarem podobné tečkám, ale vyskytují se mimo skupiny reálných teček písma, které jsou uspořádány s jistou pravidelností.

V samotném algoritmu postupně procházíme všechny řádky rozpoznané v předešlém kroku a seskupování je prováděno na základě porovnávání mezer mezi jednotlivými řádky. Vlastní algoritmus se řídí následujícími kroky:

1. Do seznamu přidáme řádek (*first*).
2. Do seznamu přidáme řádek (*second*) a spočítáme mezeru mezi *first* a *second* (*space*) na základě rozdílu pozice y.
3. Pokud je *space* 3x větší než výška řádku *second*, smaže se ze seznamu *first* a pokračujeme na bod 2.
4. Do seznamu přidáme řádek (*third*) a spočítáme mezeru mezi *second* a *third* (*space2*) na základě rozdílu pozice y.
5. Spočítáme poměr *space2*/*space* (*ratio*).
   Podíváme se na další řádek (*next*) a spočítáme mezeru mezi *third* a *next* (*space3*) na základě rozdílu pozice y.
   Spočítáme poměr *space3*/*space2* (*ratio2*).
6. Pokud se *ratio2* více blíží k jedné (tzn. mezery jsou si více podobné) než *ratio* a zároveň *next* obsahuje více teček nežli *first*, smažeme *first* a pokračujeme na bod 4.
7. Získaný seznam tří řádků prohlásíme za právoplatnou skupinu a pokračujeme opět na bod 1.

Pomocí podmínky v kroku 3. odstraníme řádky, které jsou od dalších vzdáleny výrazně více, než by tomu mělo být u řádků Braillova písma. Podmínkou v kroku 6. se odstraní ruchy, které se vyskytují těsně nad, nebo pod řádky s reálnými tečkami. Vycházíme z předpokladu, že mezi řádky Braillova písma jsou pravidelné mezery a to se nedá říct o řádcích případných ruchů.



Obrázek 3.8: Seskupení řádků. a) originální obrázek. b) řádky (modře) a skupiny (zeleně).

Výsledné skupiny řádků jsou tedy složeny vždy ze tří řádků. Může nás však napadnout, co se stane, když se v obraze vyskytují skupiny znaků, které obsazují pouze dva řádky místo tří. Algoritmus seskupení by v takovémto případě selhal. Tento jev je však v praktickém užití Braillova písma nepravděpodobný a tak můžeme prohlásit, že skupina musí mít tři řádky. Na obrázku 3.8 vidíme příklad detekovaných řádků a jejich skupin.

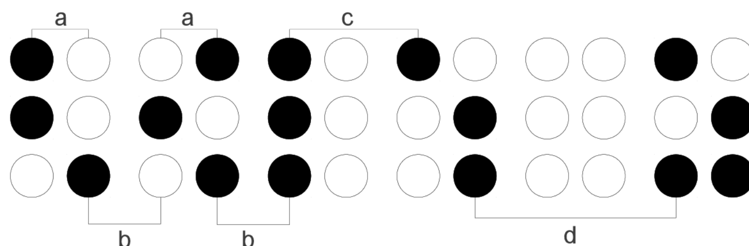
Detekce sloupců ve skupině

Pro další zpracování je nutné v každé skupině detekovat sloupce teček. Pro samotnou detekci vložíme všechny tečky ze tří řádků skupiny do jednoho seznamu a seřadíme podle souřadnice x. Pak je princip podobný jako u detekce řádků, tedy postupně procházíme celý seznam a kontrolujeme rozdíl horizontální pozice aktuálního a předchozího prvku. Na rozdíl od detekce řádků zde použijeme toleranci rovnou polovině šířky objektu. Zároveň se v každém zpracování tečky přiřadí číslo řádku, ve kterém je umístěna. Tato informace bude dále použita při zkoumání, které tečky ve znaku Braillova písma jsou obsazeny.

Jednotlivé sloupce jsou reprezentovány sadou teček a obsahují také svou x pozici a šířku. Tyto dvě hodnoty jsou opět vypočteny průměrem odpovídajících hodnot teček sloupce.

Analýza mezer a prázdných sloupců

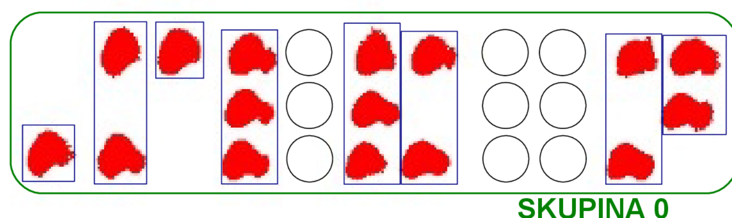
Znaky Braillova písma nemusí vždy tvořit dva sloupce teček, v řade z nich je tečkami obsazen jen jeden sloupec a ten druhý je tvořen mezerou. Dalším případem možné mezery mezi sloupci teček, je mezera mezi slovy znaků Braillova písma. Na základě velikostí mezer mezi jednotlivými sloupci, můžeme takovéto případy odhalit, chybějící sloupce doplnit a ulehčit si práci v následujícím kroku sestavování znaků.



Obrázek 3.9: Označení mezer v Braillově písmu.

Mezery mezi sloupci teček Braillova písma můžeme rozdělit do čtyř skupin, viz obr. 3.9. Nejmenší jsou mezery mezi sloupci jednotlivých znaků (a), pak mezery mezi kompletními znaky (b) a dále již zmiňované mezery při vynechaném sloupci (c) a mezi slovy (d). Poslední dvě skupiny se v některých případech nemusejí vyskytnout, zejména pokud zpracováváme fotografii zachycující málo znaků (typicky číslice ve výtahu). Abychom určili, jak velká musí být mezera, aby byla považována za vynechaný sloupec, zjistíme, jaká je největší mezera mezi kompletními znaky. Větší mezera než tato, automaticky znamená vynechaný sloupec nebo mezeru mezi slovy. O mezeru mezi slovy se jedná, pokud je zkoumaná mezera větší, než prostor daný dvojnásobkem mezery mezi znaky a dvojnásobkem šířky sloupce teček. Mezera mezi slovy by tedy měla být velká jako kompletní prázdný znak včetně mezer kolem něj.

Největší mezeru mezi kompletními znaky určíme analýzou všech mezer mezi sloupci v obraze. Ty získáme prozkoumáním všech sloupců a porovnáním jejich vzájemné horizontální pozice (souřadnice x). Získané mezery seřadíme a poté rozdělíme do skupin podle podobnosti. Hledanou hodnotu by měl správně obsahovat poslední prvek druhé skupiny. Někdy jsou ale rozdíly mezi mezerami první a druhé popsané skupiny (obr. 3.9 a a b) velmi malé a mezery jsou tak zařazeny do stejné skupiny podobnosti. Taková skupina však bude obsahovat většinu mezer mezi sloupci v obraze, a tak pokud má první skupina velikost více než 70% z celkového počtu mezer, můžeme prohlásit, že hledanou hodnotu obsahuje poslední prvek z této první skupiny. Tento výpočet musíme provést pro mezery v každé skupině zvlášť, vlivem osvětlení totiž mohou jednotlivé skupiny znaků obsahovat jinak velké mezery.



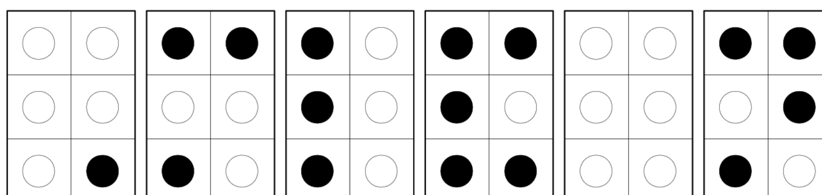
Obrázek 3.10: Detekované sloupce. Prázdné kruhy indikují doplněné prázdné sloupce.

Po zjištění požadované meze, postupně procházíme jednotlivé sloupce v každé skupině a analyzujeme jejich mezery. Pokud vyhodnotíme, že se jedná o mezeru mezi jednotlivými slovy, vložíme do tohoto místa speciálně označený sloupec reprezentující tuto mezeru. Pokud se jedná o vynechaný sloupec, vložíme do tohoto místa jeden prázdný sloupec, který neobsahuje žádné tečky.

V obrázku 3.10 vidíme příklad detekovaných sloupců a doplněných prázdných sloupců podle popsaných podmínek. Dvojice prázdných sloupců, reprezentující mezeru mezi znaky, je v programu reprezentována jen jedním sloupcem s příznakem mezery.

Sestavení znaků Braillova písma

V posledním kroku seskupování nám zbývá sestavit jednotlivé znaky Braillova písma z analyzovaných a doplněných sloupců. Každý znak je složen ze dvou sloupců, a jelikož máme doplněné prázdné sloupce, jako nejjednodušší přístup se nabízí brát postupně vždy dva sloupce a prohlásit je za znak. Toto však nelze použít vždy. Jak vidíme na obr. 3.10, první sloupec ve skupině netvoří levý sloupec znaku Braillova písma, nýbrž pravý a tak nemůžeme za znak prohlásit první dva sloupce.



Obrázek 3.11: Sestavené znaky.

Abychom určili, zda je první sloupec pravý, či levý, porovnáme mezery mezi prvními sloupci. Pokud je mezera mezi prvním a druhým sloupcem větší, než mezera mezi druhým a třetím sloupcem, znak je tvořen druhým a třetím sloupcem a první sloupec je v sestaveném znaku jako pravý. Pokud podmínka neplatí, je znak tvořen prvním a druhým sloupcem. Tento problém se nevyskytuje jen na začátku každé skupiny, ale také po každé mezeře mezi slovy. I tam je nutné provést toto srovnání mezer.

Dalším případem, který komplikuje sestavení, je situace, kdy se před mezerou mezi slovy vyskytuje znak, který obsahuje pouze levý sloupec. Pravý sloupec pochopitelně nebyl doplněn, jelikož místo něj byla doplněna mezera. Jednoduchým řešením je podmínka, kdy kontrolujeme, zdali druhý sloupec, určený pro sestavení znaku, není mezera. Pokud ano, je na její místo (tedy do pravého sloupce) vložen prázdný sloupec a mezera je zpracována v dalším kroku.

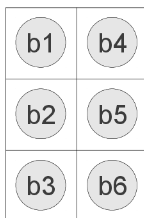
Na obrázku 3.11 vidíme znaky, které byly sestaveny ze sloupců vyobrazených na obr. 3.10. Můžeme si všimnout, že zde došlo k situaci, že byl první sloupec správně zařazen jako pravý.

3.4 Analýza a překlad

Posledním krokem celého zpracování je překlad sestavených znaků na jejich textové ekvivalenty. Vzhledem k pozici teček ve znaku Braillova písma, je možné každý znak zakódovat do binárního tvaru na základě toho, zda jsou pozice ve znaku obsazeny, či nikoliv. Takto získané binární číslo poté převedeme na dekadickou hodnotu a pomocí ní prohledáváme příslušnou jazykovou tabulku znaků a prefixů a přiřazujeme znaku význam.

Podle rozdělení znaku ilustrovaného na obr. 3.12 zapíšeme výslednou hodnotu znaku v decimálním tvaru podle vztahu:

$$hodnota_znaku = b_1 + 2 * b_2 + 4 * b_3 + 8 * b_4 + 16 * b_5 + 32 * b_6 \tag{3.5}$$



Obrázek 3.12: Mapování částí znaku Braillova písma.

Kromě znaků samotných musíme sledovat také mezery mezi slovy. Ty jsou důležité nejen pro oddělení slov, ale také jako ukončení platnosti některých prefixů. Po uplatnění prefixů, můžeme znaky sestavit do slov a výsledný řetězec zobrazit na obrazovce mobilního zařízení. Tím je proces překladu dokončen.

Kapitola 4

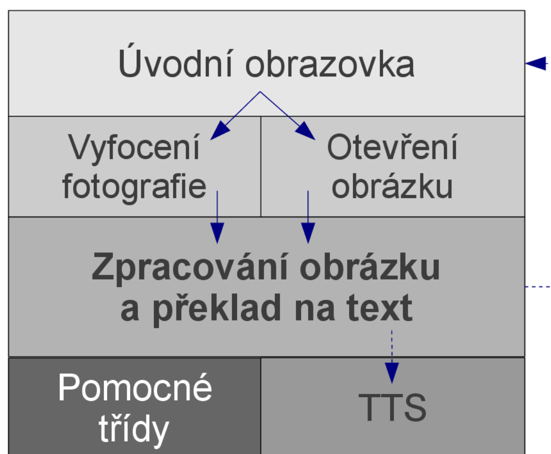
Implementace aplikace pro Android OS

Tato část bakalářské práce využívá poznatky získané v předešlých kapitolách a popisuje implementaci rozpoznávací aplikace pro operační systém Android. Kapitola obsahuje přiblížení jednotlivých součástí aplikace, popis uživatelského rozhraní a jeho ovládání a nakonec popisuje použité optimalizace.

Aplikace byla vyvíjena v prostředí Eclipse za pomocí modulu ADT, obsahujícího Android SDK. Funkčnost aplikace byla průběžně testována pomocí emulátorů obsažených v tomto modulu. Pro použití na reálném telefonu bylo nutné pomocí ADT vyexportovat aplikaci do souboru formátu `.apk`, který je instalátorem aplikací na platformu Android.

4.1 Součásti aplikace

Aplikace *BrailleReader* je rozdělena do několika logických celků, které mezi sebou komunikují. Architektura aplikace včetně naznačených vazeb je znázorněna na obr. 4.1. Plné modré šipky naznačují pevnou návaznost těchto kroků, naopak přerušované šipky ukazují volitelný přechod do této součásti.



Obrázek 4.1: Architektura aplikace.

Úplný název balíku této aplikace je `com.fitvutbr.BrailleReader`. Odkazované soubory obsahující implementaci jsou uloženy ve složce `/src/com/fitvutbr/BrailleReader/`.

Význam jednotlivých součástí je následující:

- **Úvodní obrazovka**. Aktivita, která je nastavena jako spouštěč celé aplikace. Obsahuje úvodní logo a uživatel se zde rozhoduje, zda obrázek, určený ke zpracování, vyfotí vestavěným fotoaparátem, nebo vybere z galerie. V každém z těchto případů je vytvořena zpráva (*Intent*) určující, která aktivita se má spustit. Poté je sledována návratová hodnota této aktivity a z ní je extrahován požadovaný obrázek. Získaný obrázek je následně předán ke zpracování. Implementace této aktivity se nachází v souboru `HomeScreen.java`.

- **Vyfocení fotografie**. Obrázek je získán pomocí vestavěné aktivity fotoaparátu. Ta je z úvodní obrazovky spuštěna pomocí zprávy s akcí `ACTION_IMAGE_CAPTURE`. Samotná data obrázku jsou poté obsažena v extra informacích návratové zprávy a bitmapu z nich získáme funkcí `getParcelableExtra()`.

- **Otevření obrázku**. V této aktivitě je obsluhován výběr obrázku z vestavěné paměti telefonu. Nejprve je zavolána aktivita pro získání obsahu s nastaveným filtrem výběru obrázků. Tento požadavek může být dokončen několika programy a každý z nich může vracet jinou informaci. Konkrétně v této aplikaci, jsou správně interpretovány návratové hodnoty z aplikace vestavěné galerie a programů, které vracejí přímo URI obrázku. URI je poté převedeno na cestu k obrázku a tato cesta je pomocí extra informací zprávou předána zpět úvodní obrazovce. Tato aktivita je implementována v souboru `OpenImage.java`.

- **Zpracování obrázku a překlad na text**. Stěžejní aktivita, ve které je zpracováván obrázek Braillova písma a je získán jeho textový překlad. Obrázek je do této aktivity před jejím spuštěním předán z úvodní obrazovky. Tato součást je podrobněji popsána v sekci 4.1 a implementace je uložena v souboru `ProcessImage.java`.

- **TTS**. Aktivita obsluhující převod získaného textu na zvuk (Text-To-Speech). Text je zde předán z předcházející části a aktivita je vytvořena podle principů popsaných v sekci 2.4. Po kontrole dostupnosti zdrojů a zahájení čtení daného textu je aktivita ukončena a běh pokračuje v předcházející části. Zdrojový kód implementace je v souboru `TTS.java`.

- **Pomocné třídy**. Aktivita zpracování obrázku ke svému běhu využívá několik pomocných tříd, které reprezentují jednotlivé součásti Braillova písma. Tyto třídy jsou:

  - **Objekt**. Třída reprezentující jednotlivé rozpoznané objekty v obraze. Obsahuje informace o šířce a výšce objektu, jeho pozici v obraze, seznam všech bodů využitý pro vykreslení a také pomocné ukazatele jako je počet podobných a blízkých objektů. Tyto informace jsou využity v postupech popsaných v kapitole 3. Zdrojový kód této třídy nalezneme v souboru `Obj.java`.
  - **Řádek**. Třída popisující reprezentaci řádků teček Braillova písma. Kromě seznamu objektů v tomto řádku obsahuje také informaci o pozici a výšce řádku. Definici třídy nalezneme v souboru `Line.java`.
  - **Sloupec**. Vyjádření jednotlivých sloupců znaků Braillova písma. Třída obsahuje seznam objektů v daném sloupci, jeho pozici, šířku a mezeru k dalšímu sloupci. Je zde také implementována metoda pro výpočet dekadické hodnoty tohoto sloupce na základě obsazenosti jednotlivých pozic ve sloupci. Zdrojový kód implementace je v souboru `Column.java`.

- **Znak**. Touto třídou je reprezentován jeden znak Braillova písma, který je složen ze dvou sloupců teček. Mimo to obsahuje také svou přeloženou dekadickou hodnotu. Ta je získána součtem dekadické hodnoty prvního sloupce a osminásobku dekadické hodnoty sloupce druhého. Tento vztah vychází z rozložení prezentovaného na obr. 3.12 a vzorce 3.5. Definici této třídy nalezneme v souboru `BraillChar.java`.
- **Skupina**. Tato třída vyjadřuje jednu skupinu znaků Braillova písma a v podstatě jen obaluje seznamy předešlých tříd. Nejprve jsou seskupeny řádky, poté jsou v této skupině detekovány sloupce a následně jsou tyto sloupce převedeny na jednotlivé znaky. Implementaci této třídy je v soubotu `Group.java`.

Zpracování obrázku a překlad na text

V této části aplikace jsou implementovány všechny postupy popsané v kapitole 3. Aby byla stále s uživatelem udržována aktivita, je informován o průběhu zpracování pomocí dialogu (`ProgressDialog`). Aby bylo možné tento dialog vytvořit a měnit zobrazovaný text, podle toho která část právě probíhá, je nutné samotné zpracování spustit v samostatném vlákně. S aktivitou v pozadí, která obsahuje tento dialog, komunikuje vlákno pomocí *handleru*, kterému zasílá zprávy o aktuálním průběhu. Tento *handler* poté zprávy přijímá a interpretuje. Stejným způsobem, tedy zasláním zprávy, vlákno informuje o svém ukončení a vyvolá tím dialog zobrazující výsledný text.

Jednotlivé kroky popsané v kapitole 3 jsou zde reprezentovány funkcemi, které obvykle vytvářejí či modifikují seznamy pomocných objektů popsaných výše. Po převedení znaků na jejich číselné hodnoty, zbývá jen jejich překlad na jednotlivá písmena. Abychom to mohli udělat, potřebujeme mít k dispozici tabulku znaků pro daný jazyk a také potřebujeme porozumět aplikování prefixů.

Tabulka znaků je v programu reprezentována jako zdroj řetězcových hodnot. U tabulek znaků pro různé jazyky tedy aplikujeme stejná pravidla jako pro lokalizované řetězce použité v aplikaci. Samotné XML této tabulky pro český jazyk je tedy uloženo v souboru `/res/values-cs/chars.xml` a má následující strukturu:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- Prefixy -->
    <item name="char_48" type="string">^</item>   <!-- Velka pismena
    <item name="char_60" type="string">#</item>   <!-- Cislice
    ...
    <!-- Znaky zakladni sady -->
    <item name="char_1" type="string">a</item>
    <item name="char_3" type="string">b</item>
    ...
    <!-- Cislice -->
    <item name="char_101" type="string">1</item>
    <item name="char_103" type="string">2</item>
    ...
</resources>
```

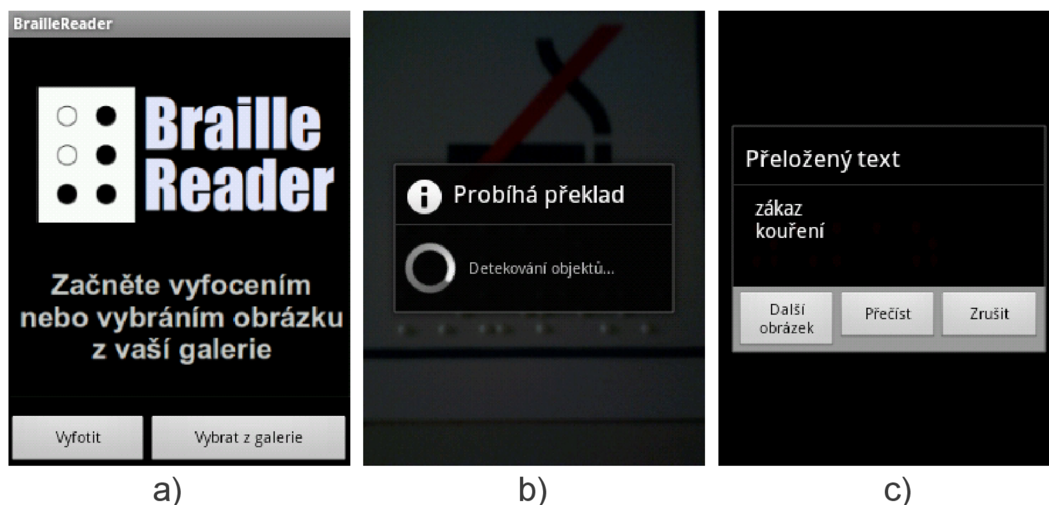
Tím, že se tabulka znaků jeví aplikaci jako zdroj, odpadá jakékoliv parsování XML z absolutně zadaného souboru do virtuální struktury, kterou bychom až poté prohledávali. K řetězcovému zdroji v aplikaci přistupujeme jednoduše pomocí několika vestavěných funkcí. Příkladem může být funkce pro překlad znaku ze zadaného čísla:

```
private String translate(Integer n) {
    int id = getResources().getIdentifier("char_".concat(n.toString()),
             "string", getPackageName());
    if (id == 0) return "";
    return getString(id);
}
```

Prefixy jsou v souboru reprezentovány speciálními znaky, které se jinak v základní sadě Braillova písma nevyskytují. V cyklu překládání jednotlivých znaků pak máme několik `boolean` proměnných, reprezentující prefixy. Proměnné jsou nastaveny při objevení znaku prefixu, při své platnosti modifikují následující znaky a jsou deaktivovány při přítomností mezery, nebo nové skupiny (nového řádku textu). Jak je naznačeno v ukázce souboru tabulky znaků, číslice jsou reprezentovány znakem s hodnotou o 100 vyšší. Číselný prefix tedy před překladem přičte 100 k aktuální hodnotě znaku. Velká písmena jsou nastavena vestavěnou funkcí `String.toUpperCase()`. Prefixy týkající se řecké abecedy nebyly implementovány, jelikož je nepravděpodobné, že by se vyskytovaly v textech na veřejných místech. Jejich princip by ale byl obdobný jako v případě číslic.

4.2 Uživatelské rozhraní

Uživatelské rozhraní aplikace bylo navrhnuto s ohledem na **jednoduchost** a **rychlost** ovládání. Jelikož může být aplikace ovládána nevidomým člověkem, je celý proces rozpoznání omezen na co nejmenší počet stisku tlačítek. Uživatelské rozhraní je ve výchozím stavu v anglickém jazyce, existuje také česká lokalizace, která je vybrána automaticky, pokud má telefon nastaven český jazyk.



Obrázek 4.2: Uživatelské rozhraní. a) úvodní obrazovka. b) průběh zpracování. c) dialog obsahující výsledný text.

Na obrázku 4.2 a) je zobrazena úvodní obrazovka aplikace. Aktivita zpracování je spuštěna ihned po získání obrázku jedním ze dvou popsaných způsobů. Při zpracování je na pozadí zobrazen zdrojový obrázek a přes něj je v dialogu reflektován aktuální stav. Toto stádium zobrazuje obr. 4.2 b). Po skončení zpracování a překladu je zobrazen dialog obsahující výsledný text, viz obr. 4.2 c). Na pozadí jsou v tuto chvíli zobrazeny detekované tečky, které jsou na obr. 4.2 c) zakryty dialogem. Uživatel se v tomto dialogu může rozhodnout, zda chce analyzovat další obrázek, nebo chce přečíst získaný text. Funkce čtení není spuštěna automaticky, jelikož pokud nemá uživatel české slovníky, jsou reprodukovaná slova dosti zkreslená.

4.3 Optimalizace

Aplikace, kterou vytváříme, je určena pro mobilní telefony, které mohou mít slabý procesor a málo paměti. Je tedy žádoucí, aby byla aplikace co nejlépe optimalizována a běžela tak co možná nejrychleji. V této části budou popsány jak obecné způsoby optimalizace aplikací pro systém Android, tak konkrétní postupy vyplývající z funkce programu.

Zmenšení zpracovávaného obrazu

Zaprvé je vhodné se zamyslet nad velikostí zpracovávané fotografie. Dnešní mobilní telefony již disponují poměrně kvalitním fotoaparátem, který v některých případech produkuje fotografie o velikosti 2592 x 1944 bodů i více. Zpracovat takto velký obrázek pixel po pixelu představuje cca 5 milionů operací. Předzpracování obrazu a v návaznosti bezpochyby také detekování objektů, by bylo tedy velice časově náročné. Pokud bychom obrázek před zpracováním zmenšili, například na rozměry 600 x 450 bodů, snížil by se počet operací na 270 tisíc, což je cca 18x méně. Otázkou je, kolik informací se tímto zmenšením ztratí a zdali toto zmenšení nebude ke škodě kvality rozpoznání.

Problém při zmenšení by mohl prakticky nastat, pouze pokud by byly vyfotografovány příliš malé tečky Braillova písma a ty by po zmenšení v obraze zanikly, případně byly považovány za malé ruchy. Aplikace je však primárně určena pro vyfotografování poměrně krátkého textu a tak je předpoklad, že tečky takového textu budou relativně velké a budou rozpoznány. Naopak, pokud fotoaparát mobilního telefonu nedisponuje funkcí zaostření, zmenšením obrazu v některých případech částečně minimalizujeme rozmazání, uzavřeme objekty a tím vylepšíme výsledné rozpoznávání.

Získaný obrázek zmenšíme tak, aby jeho šířka ani výška nepřekračovala velikost 600px. Zmenšíme vždy delší stranu a nový rozměr té kratší dopočítáme tak, aby byl zachován správný poměr. Tuto redukci provedeme ještě před předáním obrázku části zpracování a tím zredukujeme velikost dat, které se musí předat mezi těmito aktivitami. Samotnou operaci zmenšení provedeme integrovanou metodou `Bitmap.createScaledBitmap()`.

Obecné způsoby zvýšení výkonu

Jako u každého programovacího jazyka, také v systému Android existují zásady, díky kterým jsme schopni zrychlit běh celé aplikace. Těchto zásad je více, zde si však zmíníme jen ty, které se týkají prvků hojně použitých v této aplikaci [12]:

- **Vícedimenzionální matice**. Pokud pracujeme s obrazem, automaticky pracujeme také s dvojrozměrným polem, souřadnicemi x a y. Mnohem efektivnější, než použití vícedimenzionálního pole, je použití jednorozměrného pole s mapovací funkcí. API Androidu nám umožňuje jednoduchý převod bitmapy do jednorozměrného pole pomocí funkce `Bitmap.getPixels(int[], ...)`. Mapování souřadnic do tohoto pole poté provádíme mapovací funkcí ve tvaru `y * width + x`.

- **Reálná čísla**. Práce s reálnými čísly je na zařízeních Android přibližně 2x pomalejší, než práce s celočíselnými. Tam, kde není přesnost reálných čísel nezbytně nutná, použijeme výhradně čísla typu `int`.

- **Procházení seznamů**. V programu je využito poměrně velké množství neomezených seznamů typu `ArrayList`. Tyto seznamy lze procházet hned několika způsoby: pomocí iterátoru, klasickým cyklem `for`, nebo `while` a také zkráceným cyklem `for` (*for-each*). Právě poslední zmíněný způsob procházení je nejrychlejší a také zdaleka nejpřehlednější. Tam, kde je to možné, tedy využijeme právě tento způsob. V některých případech, kdy v průběhu procházení potřebujeme např. vkládat další prvky, je výhodnější použití iterátoru, i přesto, že se nejedná o procházení nejrychlejší.

- **Práce s pamětí**. Ve správné aplikaci bychom se měli vyhnout zbytečnému alokování paměti. To znamená nevytvářet nadbytečné pomocné proměnné, nové pole pixelů získat přepsáním starého nepotřebného pole apod. O uvolňování nepotřebných objektů se totiž nestaráme sami, ale provádí je *garbage collector*. Ten je spouštěn při nedostatku paměti, nebo také periodicky. Každopádně jeho spuštění má vždy vliv na výkon a tak pokud nemusíme, nebudeme alokovat nový prostor v paměti.

Kapitola 5

Testování

V této kapitole zhodnotíme úspěšnost celého procesu rozpoznání a provedeme analýzu, za jakých podmínek je rozpoznání bezproblémové a kde může docházet k chybám. Vyhodnocení bylo provedeno na základě výsledků 133 testovacích fotografií získaných ze třinácti různých zdrojů. Mezi těmito zdroji jsou výtahy, interiéry vlaků, nástupiště na autobusových nádražích či popisy učeben ve škole. Zpravidla se jedná o krátké texty informativního charakteru.

5.1 Podmínky zachycení obrazu

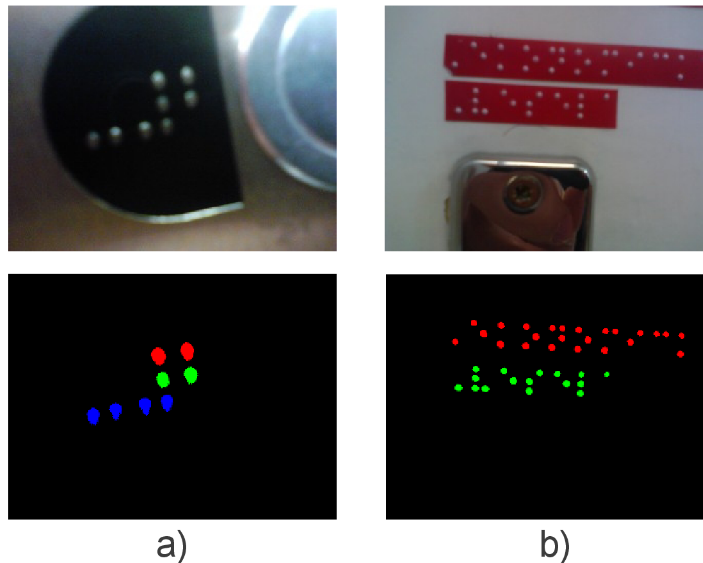
V této části budou popsány situace, za kterých mohou nastat problémy při rozpoznávání. Budou zde zmíněny také přibližné meze, při kterých by funkčnost aplikace omezena být neměla.

Otočení

Nesprávné natočení fotografie bude nejspíše jeden z nejčastějších problémů. Nejprve se musíme zamyslet, co se stane, pokud uživatel vyfotí fotografii mobilním telefonem otočeným někdy na výšku a někdy na šířku. Mohlo by nás napadnout, že v jednom z těchto případů bude získaná fotografie o 90° pootočena. Není to naštěstí tak. Mobilní telefony se systémem Android pomocí zabudovaných čidel rozpoznají, jak jsou otočeny a podle toho orientují fotografii. Pokud je tedy pořízena fotografie mobilním telefonem převráceným vzhůru nohama, aplikace získá fotografii orientovanou správně.

Toto ovšem platí pouze pro otočení o násobky 90°, uživatel stále může pořídit fotografii, která je až o 45° nesprávně natočena. V takovém případě by byly správně detekovány objekty, ale jelikož program neobsahuje metody pro analýzu otočení, výsledné seskupení a překlad by byl nesprávný. Budeme muset předpokládat, že uživatel znaky Braillova písma vyfotografuje relativně rovně. Drobné odchylky jsou samozřejmě akceptovány, platí zásada, že vždy následující tečka v řádku musí být svou horní či spodní stranou maximálně v polovině předchozí tečky. Pokud je odchylka větší, nebudou již tyto dvě tečky rozpoznány ve stejném řádku.

Míra natočení, při kterém jsou řádky spolehlivě rozpoznány, velice záleží na délce textu. Text s krátkými řádky může být daleko více natočen, nežli delší text. Je to dáno tím, že když je natočen dlouhý řádek, rozdíl mezi pozicí první a poslední jeho tečky je tak velký, že může být poslední tečka již na úrovni první tečky dalšího řádku. Řádky jsou poté nesprávně sloučeny a překlad selže. Příklad tohoto jevu můžeme vidět na obr. 5.1, kde je každý detekovaný řádek zobrazen jinou barvou.



Obrázek 5.1: Příklad natočení krátkého a dlouhého textu. a) řádky správně detekovány. b) v důsledku otočení jsou řádky sloučeny.

Světelné podmínky

Fotografie Braillova písma může být vyfotografována za špatných světelných podmínek. Obzvláště pokud mobilní telefon nemá blesk, může se stát, že získaná fotografie bude příliš tmavá na to, aby bylo rozpoznání úspěšné. V obraze jsou rozpoznávány odlesky či změny barev jednotlivých teček. Pokud jsou tedy světelné podmínky natolik špatné, že na fotografii nejsou tečky zřetelně viditelné, nerozezná je ani program. Na obr. B.2 v příloze B můžeme vidět příklad špatně osvětlené fotografie, u které z tohoto důvodu nejsou správně detekovány všechny tečky.

Rozmazání

Při použití fotoaparátu nevybaveného zaostřováním, nebo v důsledku jeho pohybu při uzavření závěrky, může vzniknout fotografie, která je rozmazaná. V aplikaci nejsou použity žádné algoritmy, které by rozmazání napravovaly, či zmenšovaly jeho dopad. Proto zde opět platí, že pokud je rozmazání velké a tečky mají po segmentaci příliš odlišný tvar, nebo jsou například spojeny, bude rozpoznání neúspěšné. Neznamená to však, že musí být fotografie naprosto ostré. V příloze B na obrázku B.3 vidíme, že byly tečky úspěšně rozpoznány i přesto, že se jedná o poměrně výrazné rozmazání.

Vzdálenost

Kvůli zmenšení fotografie diskutované v sekci 4.3, má aplikace svá omezení co se týče vzdálenosti, ze které je schopna spolehlivě rozpoznat znaky Braillova písma. Efektivní vzdálenost se liší od kvality zachycovaných teček. Například u teček, které jsou vyvedeny jinou barvou, než je pozadí, je plocha segmentovaného objektu větší, než u teček barvy stejné jako pozadí. V prvním případě je totiž prahováním zachycena celá tečka, kdežto v druhém případě jen její odlesk, tedy jen část. Z toho důvodu je efektivní vzdálenost větší u teček, které mají barvu odlišnou od pozadí. Aplikace tedy není určena pro rozpoznávání celých listů textu, nebo písma, které je vyfoceno z moc velké dálky.

Povrch a barva teček

Jak bylo naznačeno v předchozím odstavci, lépe jsou rozpoznávány tečky s barvou odlišnou od pozadí. Naopak nejhůře jsou na tom tečky, které jsou vytlačeny do lesklého plechu či plastu. Světlo totiž na lesklém povrchu může způsobit parazitní odlesky a tím zapříčinit zmizení některých teček nebo jejich deformaci. Příklad lesklého povrchu u kterého není možné rozpoznat všechny tečky, lze pozorovat na obr. B.5 v příloze B.

5.2 Výsledky testování

V této sekci budou zobrazeny a diskutovány výsledky získané testováním. Samotné testování bylo prováděno v emulátoru se systémem Android verze 2.3.3. Do sady testovacích obrázků byly záměrně zařazeny i fotografie, které zjevně obsahují chybu, ale zároveň názorně ukazují některé z výše popsaných problémů.

Úspěšnost rozpoznání

Úspěšnost rozpoznání všech testů demonstruje tabulka 5.1. V prvním sloupci jsou testy, ve kterých byly detekovány všechny tečky a byly správně přeloženy na odpovídající text. Druhý sloupec pak zachycuje případy, kdy byly nesprávně detekovány 1-2 tečky a vznikla tím chyba u maximálně dvou znaků. V těchto případech je význam přeloženého textu stále velmi patrný, a tak není tyto testy vhodné považovat za zcela chybné. Jako chybný je považován překlad obsahující více chyb, či překlad ukončený chybovou hláškou o nerozpoznání žádných znaků.

	Kompletně správné testy	Malá chyba v 1-2 znacích	Chybný překlad
Počet	110	7	19
Procento	82,7%	5,3%	12%

Tabulka 5.1: Tabulka úspěšnosti.

Samotné chyby byly způsobeny několika druhy problémů, které jsou popsány výše. Výskyt jednotlivých druhů chyb zachycuje tabulka 5.2. Selhání testu uvedeného v posledním sloupci, bylo způsobeno zachycením slova, které obsahovalo pouze dva řádky teček.

	Rozmazání	Natočení	Osvětlení	Lesklý povrch	Vzdálenost	Jiné
Počet	3	3	3	5	1	1
Procento	18,7%	18,7%	18,7%	31,3%	6,3%	6,3%

Tabulka 5.2: Statistika chybových testů. Jednotlivé důvody chyb a jejich počty.

Rychlost rozpoznání

U vyvíjené aplikace je důležitá také rychlost celého zpracování. Bylo by nežádoucí, kdyby aplikace byla sice spolehlivá, ale rozpoznání každého obrázku by trvalo například více než půl minuty. Experimentálním měřením byl zjištěn průměrný čas zpracování fotografie cca 6s u mobilního telefonu Huawei U8650 Sonic (s procesorem ARM1136 600 MHz a 256MB RAM) a cca 3s v případě mobilního telefonu Sony Ericsson Xperia Mini Pro (s procesorem Scorpion 1000 MHz a 512MB RAM). U výkonnějších mobilních telefonů se dá předpokládat ještě větší zrychlení. Samotný čas byl měřen od výběru obrázku, až po zobrazení výsledného dialogu. Jedná se tedy o dostatečnou rychlost, která zásadně neznepříjemní práci s aplikací.

Závěr

Cílem práce bylo nejprve prostudovat technologii Braillova písma a principy operačního systému Android. Dále jsem prostudoval existující práce, zabývající se rozpoznáním Braillova písma. Většina z nich se však zabývá rozdílnou třídou aplikací, a to rozpoznáním celých stran textu z obrazu pořízeného scannerem. Podobným problémem jaký jsem řešil zde, se zabývá článek *A Braille Recognition System by the Mobile Phone with Embedded Camera* [18], s jehož pomocí jsem navrhl základní koncepci rozpoznávání. V posledním kroku jsem implementoval navržené metody do aplikace běžící na systému Android a provedl důkladné testování této aplikace na reálných fotografiích, pořízených mobilním telefonem.

Hlavním přínosem této práce je kompletní návrh postupu rozpoznávání Braillova písma z fotografie. Vyjma předzpracování jsou všechny popsané metody vlastní - touto prací jsem tedy zpracoval poměrně neprobádanou oblast rozpoznání Braillova písma. Všechny postupy, které jsem navrhl, musí být poměrně dynamické, což je způsobeno zpracováním běžné fotografie ze vždy odlišného prostředí. Rozpoznané tečky a mezery mezi nimi, mohou být pokaždé jinak velké i v jiném poměru. Mezi nejkomplikovanější části tedy patří právě analýza mezer a následné správné sestavení znaků.

V testování, které jsem provedl na základě 133 fotografií z 13 různých zdrojů, bylo kompletně správně přeloženo 83% fotografií. Pokud přičteme testy, kde byla chyba způsobena maximálně dvěma špatně detekovanými tečkami (1-2 nesprávné znaky), stoupne úspěšnost na 88%. V ostatních případech se jedná a rozsáhlejší chyby (rozmazání teček, špatné světelné podmínky, odlesky), případně o nerozpoznání žádných znaků. Velké množství chybných snímků jsou fotografie lesklého povrchu s lesklými tečkami. Tento povrch je nejproblémovější, a pokud bychom nebrali v potaz snímky s chybou způsobenou lesklým pozadím, dostali bychom úspěšnost 91,5%. Chyby byly z největší části způsobeny nerozpoznáním některých teček, tedy problémem způsobeným již ve fázi předzpracování obrazu a detekce objektů. Další častější chybou bylo špatné rozpoznání řádků z důvodu špatného natočení obrazu.

Výslednou aplikaci jsem uzpůsobil pro použití nevidomým. Kladl jsem důraz na jednoduchost ovládání a na získání výsledku v co nejmenším počtu kroků, tedy kliknutím na tlačítko. Dále jsem pro nevidomé implementoval funkci pro převod výsledného přeloženého textu na hlas. Pro reálné použití mezi nevidomými, by ale bylo zřejmě ještě potřeba aplikaci testovat přímo s těmito lidmi a zapracovat jejich poznatky.

Práce by v dalším možném vývoji mohla být vylepšena hned v několika oblastech. Předzpracovaní obrazu by mohlo být doplněno o takové metody, které by zlepšovaly rozpoznání teček za horších světelných podmínek a na obtížnějším povrchu. Dále by mohl být doplněn krok, který by detekoval nežádoucí natočení a aplikoval by na obraz transformace pro jeho narovnání. Otázkou zůstává, jaký by mělo přidání těchto metod dopad na výkon aplikace a čas zpracování. Myslím, že by se jednalo o výrazné zpomalení. Na druhou stranu, výkon mobilních telefonů stále roste a již dnes existují přenosná zařízení se čtyřjádrovým

procesorem. Dalším krokem by tedy mohla být i optimalizace aplikace pro vícejádrové procesory, čímž by některé metody bylo možné zpracovávat paralelně. Velkého vylepšení výsledného překladu by se také mohlo dosáhnout zavedením kontroly neobsazených pozic jednotlivých znaků. Mohly by se u každého znaku Braillova písma dopočítat pozice teček, které nebyly detekovány a kontrolovat, zda na těchto místech nebyly shluky, které byly například z důvodu deformace vyřazeny. Někdy totiž nastává situace, kdy jsou některé z teček nesprávně vyřazeny nebo spojeny kvůli rozmazání. Výsledná slova by také mohla být porovnávána se slovníkem slov daného jazyka. Tím by mohly být opraveny některé chyby, způsobené špatným rozpoznáním několika málo znaků ve slovech.

Literatura

[1] World Health Organization: Visual impairment and blindness [online]. http://www.who.int/mediacentre/factsheets/fs282/en/, October 2011, [cit. 2012-05-01].

[2] National Federation of the Blind: Braille General [online]. http://nfb.org/braille-general, March 2012, [cit. 2012-05-01].

[3] Canalys: Smart phones overtake client PCs in 2011 [online]. http://www.canalys.com/newsroom/smart-phones-overtake-client-pcs-2011, 3.1.2012, [cit. 2012-05-01].

[4] Canadian Braille Authority: About Braille [online]. http://www.canadianbrailleauthority.ca/en/about_braille.php, [cit. 2012-05-01].

[5] Sébastien Sablé: Braille alphabet [online]. http://libbraille.org/alphabet.php, [cit. 2012-05-01].

[6] Nemeth, A.: *The Nemeth Braille Code for Mathematics and Science Notation, 1972 Revision*. American Printing House for the Blind, 1987. URL http://www.brailleauthority.org/mathscience/nemeth1972.pdf

[7] American Foundation for the Blind [online]: Music Braille [online]. http://www.braillebug.org/music_braille.asp, 2011, [cit. 2012-05-01].

[8] Antonacopoulos, A.: Automatic Reading of Braille Documents. In *Handbook of Character Recognition and Document Image Analysis*, editace H. B. P. Wang, World Scientific, 1997, s. 703–728.

[9] Burnette, E.: *Hello, Android: Introducing Google's Mobile Development Platform*. Pragmatic Bookshelf, 2009, ISBN 978-1-934-35617-3, 228 s.

[10] Meier, R.: *Professional Android 2 Application Development*. Wiley Publishing, 2010, ISBN 978-0-470-56552-0, 543 s.

[11] Murphy, M.: *Beginning Android*. Apress, 2009, ISBN 978-1-430-22419-8, 361 s.

[12] Google Inc.: Android Developers [online]. http://developer.android.com/, [cit. 2012-05-01].

[13] Hashimi, S.; Komatineni, S.; MacLean, D.: *Pro Android 2*. Apress, 2010, ISBN 978-1-430-22659-8, 718 s.

[14] Antonacopoulos, A.; Bridson, D.: A Robust Braille Recognition System. In *Document Analysis Systems VI*, editace S. Marinai; A. Dengel, Springer, 2004, s. 533–545.

[15] Mennens, J.; Van Tichelen, L.; Francois, G.; aj.: Optical recognition of Braille writing. In *Document Analysis and Recognition, 1993., Proceedings of the Second International Conference on*, 1993, s. 428 –431.

[16] Al-Ohali, Y.; Al-Salman, A.; Alkanhal, M.; aj.: An Arabic Optical Braille Recognition System. In *ICTA*, Tunisea, April 2007 2007.

[17] Ng, C.; Ng, V.; Lau, Y.: Regular feature extraction for recognition of Braille. In *Computational Intelligence and Multimedia Applications, 1999. ICCIMA '99. Proceedings. Third International Conference on*, 1999, s. 302 –306.

[18] Zhang, S.; Yoshino, K.: A Braille Recognition System by the Mobile Phone with Embedded Camera. In *Innovative Computing, Information and Control, 2007. ICICIC '07. Second International Conference on*, 2007, str. 223.

[19] Kršek, P.; Španěl, M.: Základy počítačové grafiky [online]. `https://www.fit.vutbr.cz/study/courses/IZG/private/`, 2007, [cit. 2012-05-01].

[20] Sonka, M.; Hlaváč, V.; Boyle, R.: *Image processing, analysis, and machine vision*. PWS Pub., 1999, ISBN 978-05-3495-393-5, 770 s.

[21] Otsu, N.: A Threshold Selection Method from Gray-Level Histograms. *Systems, Man and Cybernetics, IEEE Transactions on*, 1979: s. 62–66, ISSN 0018-9472.

[22] Dobeš, M.: *Zpracování obrazu a algoritmy v C#*. BEN - technická literatura, 2008, ISBN 978-80-7300-233-6, 143 s.

[23] Wellner, P. D.: Adaptive Thresholding for the DigitalDesk. Technická zpráva, 1993.

[24] Bradley, D.; Roth, G.: Adaptive Thresholding using the Integral Image. *Journal of Graphics, GPU, and Game Tools*, ročník 12, č. 2, 2007: s. 13–21.

[25] Šonka, M.; Hlaváč, V.; Kozlík, V.: *Počítačové vidění*. Grada, 1992, ISBN 80-85424-67-3, 252 s.

Seznam použitých zkratek a symbolů

zkratka	**celý název**	**vysvětlení**
iOS	iPhone OS	Mobilní operační systém vyvíjený společností Apple.
OHA	Open Handset Alliance	Uskupení firem, které stojí za vývojem operačního systému Android.
OS	Operating System	Operační systém - základní programové vybavení počítače.
SGL	Skia Graphics Library	Open source grafická knihovna.
SSL	Secure Sockets Layer	Kryptografický protokol nabízející bezpečnou komunikaci na internetu.
API	Application Programming Interface	Rozhraní pro programování aplikací.
SDK	Software Development Kit	Sada nástrojů pro vývoj softwaru pro určité zařízení.
SD	Secure Digital	Paměťová karta používaná v přenosných zařízeních.
SMS	Short Message Service	Služba krátkých textových zpráv využívaná v mobilních telefonech.
GPS	Global Positioning System	Globální družicový polohový systém.
XML	Extensible Markup Language	Rozšiřitelný značkovací jazyk.
DPI	Dots per inch	údaj určující, kolik pixelů se vejde do délky jednoho palce
URI	Uniform Resource Identifier	Řetězec sloužící k přesné specifikaci zdroje informací.

Seznam příloh

Příloha A Základní česká znaková sada
Příloha B Ukázky testů

Příloha A

Základní česká znaková sada

a 1 b 2 c 3 d 4 e 5 f 6 g 7 h 8

i 9 j 0 k l m n o p %

q r s t u v w x

y z á č ď é ě í

ň ó ř š ť ú ů ý

ž . , : ; - + /

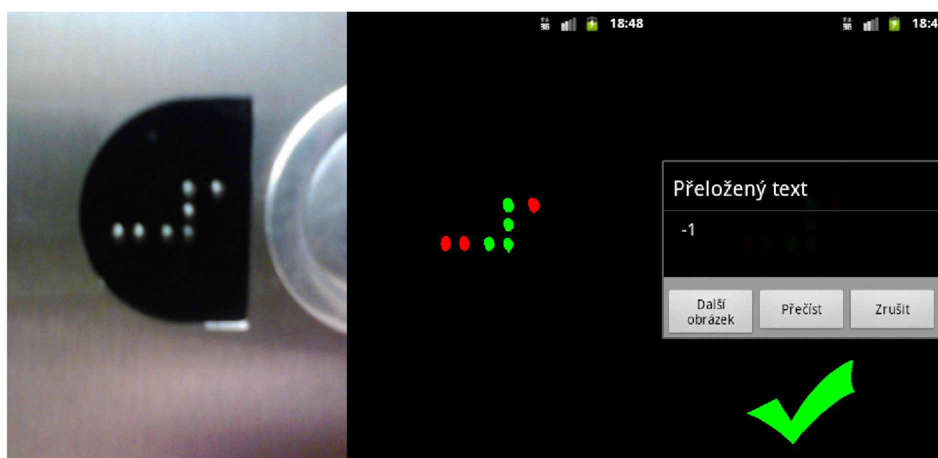
? ! “ ( ) * ' |

prefix velkého písmene řetězec velkých písmen prefix malého písmene číselný prefix mezera

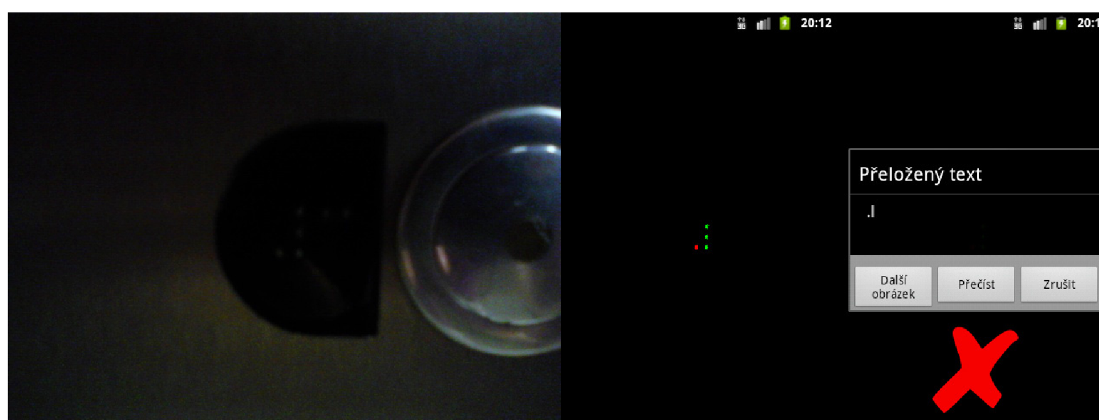
Příloha B

Ukázky testovacích obrázků

Následující obrázky jsou ukázkou zpracování testů. Každý test obsahuje původní obrázek a dva screenshoty z běžící aplikace. První screenshot zobrazuje detekované tečky, přičemž každý detekovaný znak je vyveden jinou barvou. Druhý pak zobrazuje výsledný dialog s přeloženým textem. Kompletní sada testů je umístěna na přiloženém CD.

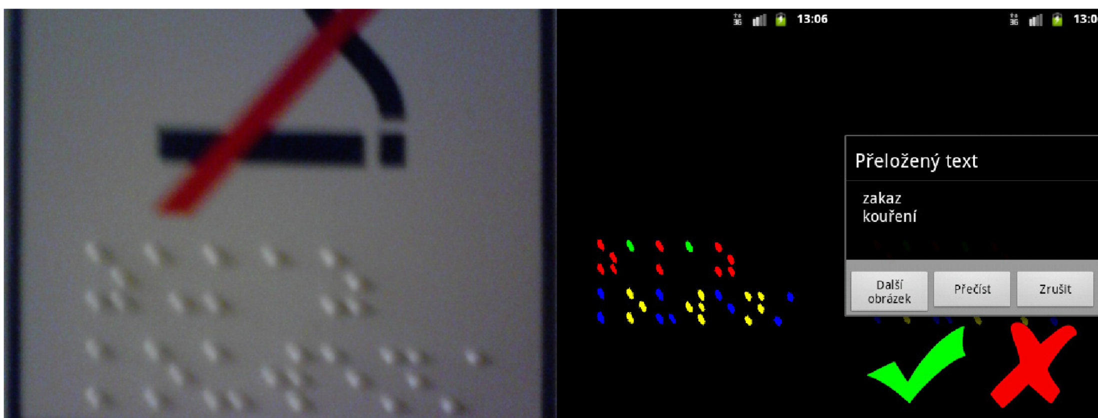


Obrázek B.1: Test 02 - Výtah FIT.



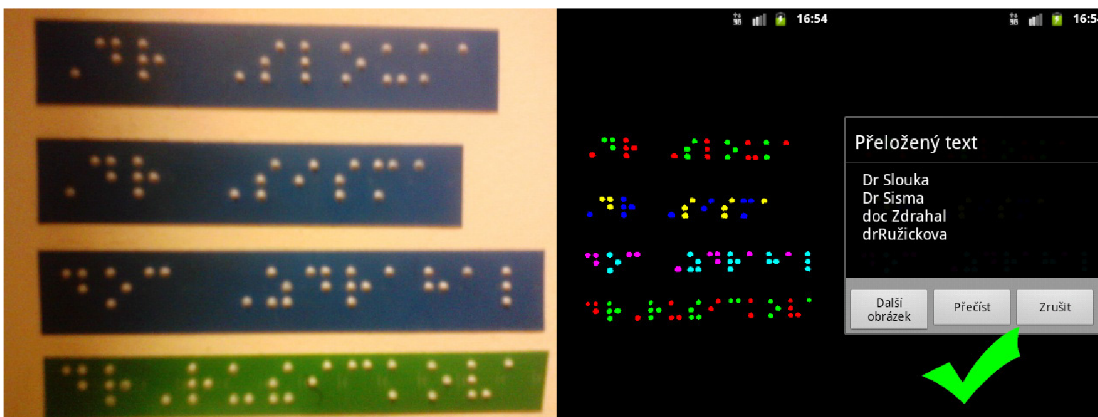
Tmavé osvětlení, nebyly rozpoznány všechny tečky

Obrázek B.2: Test 14 - Výtah FIT.



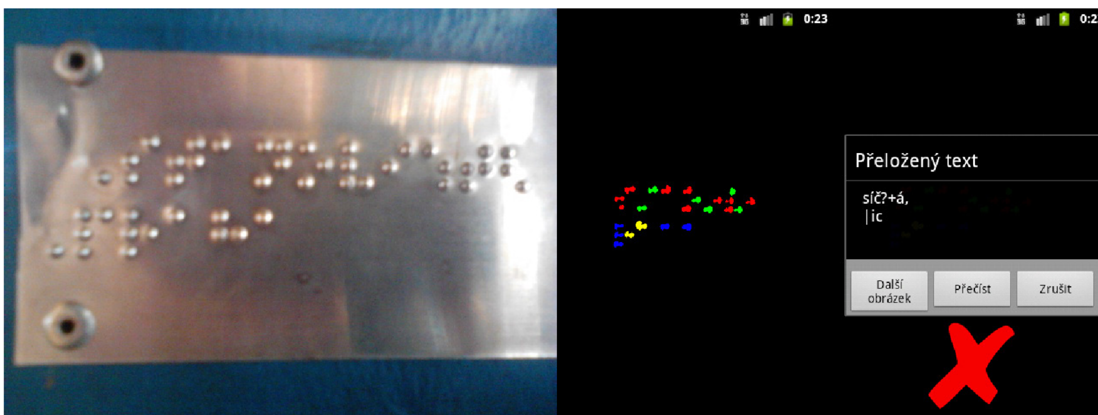
Poměrně rozmazaný obrázek, kde je špatně datekována 1 tečka

Obrázek B.3: Test 36 - Vlak EC Praha-Ostrava.



Na rozdíl od prvních tří řádků, v posledním není mezera mezi titulem a jménem.

Obrázek B.4: Test 67 - Učebny PdF UP Olomouc.



Moc lesklé pozadí. Tečky tak vystupují pokazdé jinak, nebo s ním splývají.

Obrázek B.5: Test 100 - Autobusové nádraží Brno.