České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Martin Milichovský**

Studijní program: Otevřená informatika
Obor: Umělá inteligence

Název tématu: **Mobilní aplikace pro vyhledání a rozpoznání textu v obrazech reálných scén**

Pokyny pro vypracování:

1. Prostudujte předloženou literaturu a nastudujte metodu pro rozpoznání textu.
2. Navrhněte funkčnost aplikace pro mobilní telefony.
3. Navržený způsob implementujte pro platformu Android.
4. Vyhodnoťte schopnosti aplikace a její použitelnost uživateli.

Seznam odborné literatury:

- Neumann L.: Vyhledání a rozpoznání textu v obrazech reálných scén, Master thesis, ČVUT 2010
- Neumann L., Matas J.: Real-Time Scene Text Localization and Recognition, CVPR 2012 (Providence, Rhode Island, USA)

Vedoucí: Ing. Daniel Novák, Ph.D.

Platnost zadání: do konce letního semestru 2015/2016

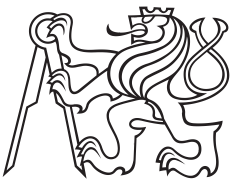
L.S.

doc. Ing. Filip Železný, Ph.D.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 31. 10. 2014

Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Mobilní aplikace pro vyhledání a rozpoznání textu v obrazech reálných scén

Martin Milichovský

Program: Otevřená informatika
Obor: Umělá inteligence

4. 1. 2015
Vedoucí práce: Ing. Daniel Novák, Ph.D.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 4. 1. 2015

Abstrakt / Abstract

Tato práce se zabývá implementací aplikace, která rozpozná text v obrázku reálné scény. Je to komplikovanější obdoba rozpoznání textu z tištěného dokumentu. Aplikace je určena pro mobilní telefony, což je prostředí s omezenými prostředky. Tyto aspekty jsou v práci diskutovány.

Možné využití je pomoc zrakově postiženým osobám nebo automatický překlad nalezeného textu.

Nejprve je popsán postup algoritmu pro získání textu ze vstupního obrázku. Dále je tento postup implementován do knihovny, která jednotlivé úkony provádí paralelně. Z ní vychází výsledná aplikace pro Android.

Na závěr je zhodnocena použitelnost aplikace, jejíž úspěšnost sice není 100%, což je ale dáno náročností této úlohy. Výsledky jsou porovnány s existujícími aplikacemi řešícími podobný problém.

Klíčová slova: lokalizace textu; rozpoznání textu; mobilní aplikace; text v reálné scéně

This thesis describes implementation of an application for scene text recognition. That is a more complex task than text recognition in printed documents. The application is designed for mobile phones, thus allocation of resources is taken into special consideration.

Possible use of such application is assitence for visualy impaired, or machine translation of the recognized text.

At first, the algorithm for extracting text from image is described. This approach is implemented as a library that processes its tasks in parallel. On top of it an Android application is built.

The conclusion is, the application is not 100 % successful, which is determined by the task's complexity. The result is compared to existing state-of-the-art applications for mobile phone scene text recognition.

Keywords: text localization; text recognition; mobile application; scene text recognition; text in the wild; photo OCR

Obsah /

Kapitola 1
Úvod

Rozpoznání textu v reálných scénách spočívá v přečtení textové informace z fotografií světa kolem nás. Narozdíl od čtení dokumentů (dnes již uspokojivě řešeného) se zaměřuje na obrázky, kde text není „černý na bílém“, ale může mít proměnlivou barvu i pozadí, může být našikmo či jinak deformovaný, a zabírá poměrně malou část scény.

Obrázek 1.1. Fotografie obsahující text

Vzhledem k rozšíření chytrých telefonů, jejichž fotoaparáty dokáží zachytit obraz v uspokojivém detailu, je nasnadě vytvořit aplikaci, která bude extrahovat text z právě zachycené scény. Praktické využití takto získaného textu může být například následný překlad při návštěvě cizích krajů, přečtení nahlas pro zrakově postižené nebo prosté zkopírování textu do schránky.

1.1 Cíl práce

V této práci budu vytvářet offline[1]) aplikaci pro Android, který má v současné době 80% zastoupení mezi operačními systémy chytrých mobilních telefonů[2]). Prostředí telefonu má omezenou velikost operační paměti a výpočetní výkon, v porovnání s osobními počítači; na to bude brán zřetel.

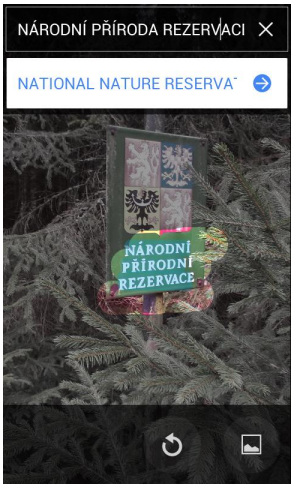
Předně se budu zabývat zpracováním obrázku, které spočívá ve dvou logických stupních: lokalizace, tj. nalezení oblastí s (potenciálním) textem, a rozpoznání písmen nalezeného textu (OCR). Kritické algoritmy na zpracování obrázku jsem se rozhodl implementovat v C++, jež se kompiluje do nativního kódu. Tato část práce tvoří ucelenou knihovnu, kterou nakonec použiji jako základ grafické aplikace, jež je již implementována standardně v Javě.

[1]) Tj. fungující bez potřeby připojení k internetu.
[2]) `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`

1.2 Existující aplikace

1.2.1 Google Translate

V rámci překládací aplikace lze získat text z obrázku, obrázek je zpracován na serverech Google, tedy k používání je potřeba internetové připojení. Spolehlivost a rychlost je obdivuhodná, aplikace je schopná přečíst i velmi malý a hustý text téměř okamžitě. Samozřejmě může záležet na rychlosti připojení, ale zdá se mi, že nějaká analýza probíhá již v telefonu a odesílaná data jsou menší než celý obrázek. Také mi přišlo, že komplikovanější obrázky vyžadují přenést více dat. Použitá metoda PhotoOCR[1] zpracovává obrázek na mnoha serverech paralelně, a vyžaduje velký výpočetní výkon.



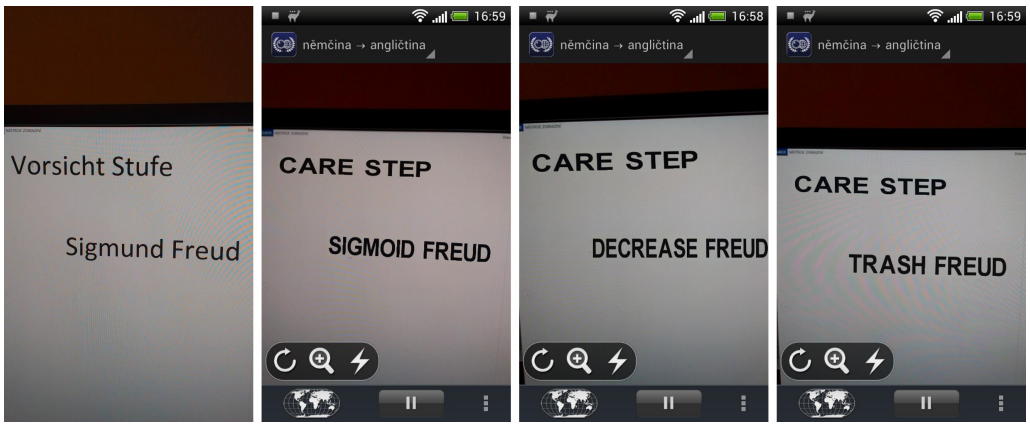
Obrázek 1.2. Google Translate: Národní přírodní rezervace

Jako hlavní nevýhodu vidím potřebu připojení k internetu. Oproti tomu samotné překládání lze provádět offline, pomocí předem stažených jazykových balíčků.

1.2.2 Word Lens

Word Lens zobrazuje text v rámci rozšířené reality. Nabízí překlad mezi hlavními světovými jazyky a přímo v náhledu fotoaparátu nahrazuje nalezený text jeho překladem.

Aplikace rozpoznává pouze fixní množinu naučených slov, která je pro většinu běžných textů dostačující (dává smysl číst jen slova, jež lze dále přeložit). Překlad probíhá slovo po slovu, bez kontextu.



Obrázek 1.3. WordLens: Překlad z němčiny do angličtiny. Neznámé slovo: Sigmund

Rychlost aplikace WordLens je obdivuhodná, nahrazování textu probíhá opravdu v reálném čase. Je to za cenu toho, že si neporadí například s textem na nejednobarevném pozadí. Dále, pokud se do hledáčku dostane větší množství textu, odezva se zpomalí a některá slova „poblikávají“, tedy v žádném okamžiku není přečteno vše pohromadě. Pokud se do hledáčku dostane neznámé slovo, bývá rozpoznáno jako jiné, nesouvisející slovo.

V květnu 2014 koupil firmu Quest Visual (autora aplikace) Google.

1.2.3 Bing Translator

Bing Translator (na Windows Phone) od Microsoftu je také primárně aplikace pro překlad, s možností získání textu z obrázku. Text získává z náhledu kamery a téměř v reálném čase zvládá přečíst i drobný text. Zároveň sleduje jeho polohu. Zpracování probíhá přímo v telefonu. Aplikace často *najde* text i tam, kde žádný není.

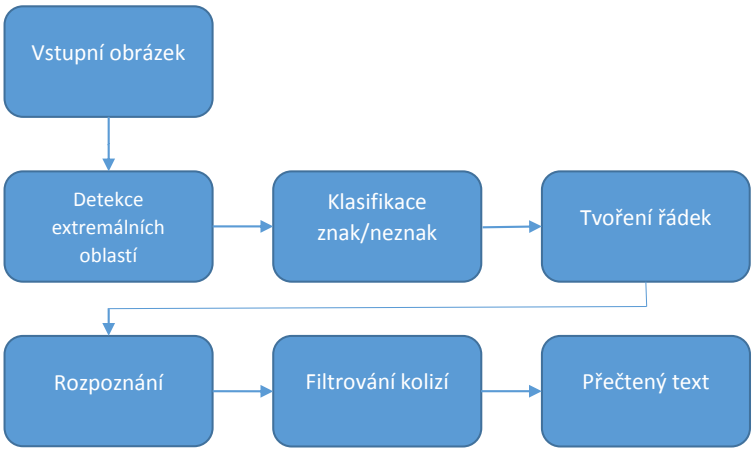
Obrázek 1.4. Bing Translator

Kapitola 2
Návrh řešení

Rozpoznání textu v obrázku spočívá ze dvou fází. Nejprve je potřeba identifikovat oblasti kde se text (pravděpodobně) vyskytuje, tzv. **lokalizace**. Jejím výsledkem jsou *malé obrázky*, které již obsahují jen text. Dalším úkolem je ho přečíst, tzv. **rozpoznání**.

Pro lokalizaci jsem se rozhodl použít metodu podle Neumann a Matas[2–3], jež je založena na předpokladu, že jednotlivá písmena jsou extremálními oblastmi a dále že řádky textu mají dané geometrické vlastnosti.

Takto nalezené řádky dále zpracuji pomocí Tesseract OCR[4] vyvíjeného Google pod licencí Apache 2.0. Výhodou je, že jsou snadno dostupné balíčky s naučenými jazyky.



Obrázek 2.1. Diagram fází od zdrojového obrázku k přečtenému textu

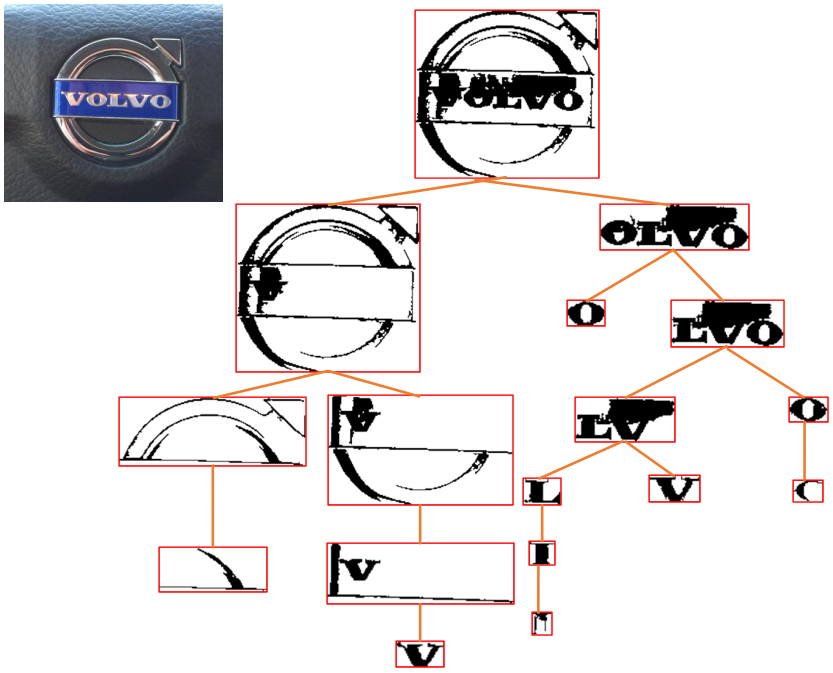
2.1 Extremální oblasti

Pro detekci extremálních[5] oblastí je potřeba, aby byl obrázek reprezentován jako 2D matice uspořádaných hodnot, typicky $\{0, .., 255\}$. Přitom pixel barevného RGB obrázku je v prostoru $\{0, .., 255\}^3$. Pro zpracování převedu obrázek do stupňů šedi.

Extremální oblast je potom, zjednodušeně řečeno, souvislá množina pixelů R taková, že jejich hodnota je $\leq \theta$ a zároveň všechny sousedící pixely mají hodnotu $> \theta$, kde θ je práh pro danou extremální oblast. Jinak řečeno, oblast nelze již rozšířit o žádný další pixel, který by byl alespoň tak tmavý jako θ.

Z toho vyplývá, že pro $\theta = 255$ je celý obrázek jedna velká extremální oblast, a každá oblast s $\theta = n - 1$ je podmnožinou nějaké oblasti s $\theta = n$. Oblasti tedy tvoří stromovou strukturu.

Podle výše popsaného jsou extremální oblasti tmavší než jejich okolí, a tedy světlá písmena na tmavém pozadí nejsou extremálními oblastmi. Proto je potřeba celý proces kromě původního obrázku aplikovat ještě na jeho negativu, tedy $c' = 255 - c$, kde c je původní intenzita.



Obrázek 2.2. Některé extremální oblasti v daném obrázku (reprezentovaným v negativu stupňů šedi); strom odpovídá skládání oblastí jako množin – nadřízený je nadmnožinou podřízeného.

Algoritmus sestaví oblasti v čase $O(n \log(\log(n)))$, kde n je počet pixelů v obrázku[5].

2.2 Klasifikace

2.2.1 Deskriptory

Extremální oblasti tvoří stromovou strukturu, čehož využívá použitá metoda [3] tak, že začíná od malých oblastí ($\theta = 0$) a pro každou oblast eviduje tzv. inkrementálně počítané deskriptory, mající tu vlastnost, že při sloučení dvou oblastí do větší lze spočítat její deskriptor jen na základě předchozích a polohy pixelu, který je sjednocuje; tedy s konstantní asymptotickou složitostí.

- **Plocha** a. Plocha vzniklé oblasti po spojení je prostý součet dílčích ploch.
- **Ohraničující rámeček** – souřadnice levého horního a pravého dolního pixelu $(x_{min}, y_{min}, x_{max}, y_{max})$ definují minimální obdélník do kterého se oblast vejde. Při sjednocení se ze složek určí (min, min, max, max). Šířka oblasti $w = x_{max} - x_{min}$ a výška $h = y_{max} - y_{min}$.
- **Obvod** p. Obvody oblastí se sečtou, avšak je potřeba zvážit možné situace prodloužení obvodu podle toho, z kolika stran se propojující pixel dotýká původních oblastí. (více v [3]).
- **Eulerova charakteristika** η. Vyjadřuje rozdíl mezi počtem souvislých komponent a počtem děr v obrazci. Vzhledem k tomu, že extremální oblast je vždy souvislá, η sleduje počet děr. Při spojování lze η inkrementálně určit podle příslušnosti k oblastem osmi okolních pixelů k přidávanému pixelu. (více opět v [3])
- **Horizontální přerušení** je vektor o délce podle výšky extremální oblasti $(h+1)$, který vyjadřuje pro každou y-hladinu počet přechodů mimo oblast a do oblasti. Pokud pixel doléhá zleva nebo zprava k oblasti, tak nemění počet přerušení. Jestliže doléhá z obou stran, tak snižuje počet přerušení o 2 (zaplňuje mezeru), jinak zvyšuje o dvě.

Pro aplikaci jsem se rozhodl vynechat deskriptor *horizontální přerušení*. Pro spojování oblastí je předpokládán vektor s konstantním přístupem a vkládáním na oba konce. Asymptotická složitost je sice konstantní, ale z hlediska paměťové náročnosti se může jednat až o vektor dlouhý podle výšky obrázku pro každý pixel.

2.2.2 Klasifikátor

Po zkonstruování stromu extremálních oblastí (těch jsou v běžné fotce řádově desetitisíce) je potřeba vybrat ty, které by mohly být znaky. Podle [3] jsem použil AdaBoost.

Jednotlivé slabé klasifikátory jsou lineární prahy s příznaky (vypočtenými na základě deskriptorů oblasti): poměr (w/h), kompaktnost $(\sqrt{a}/p)$ a počet děr $(1-\eta)$. Všechny tyto příznaky jsou nezávislé na měřítku.

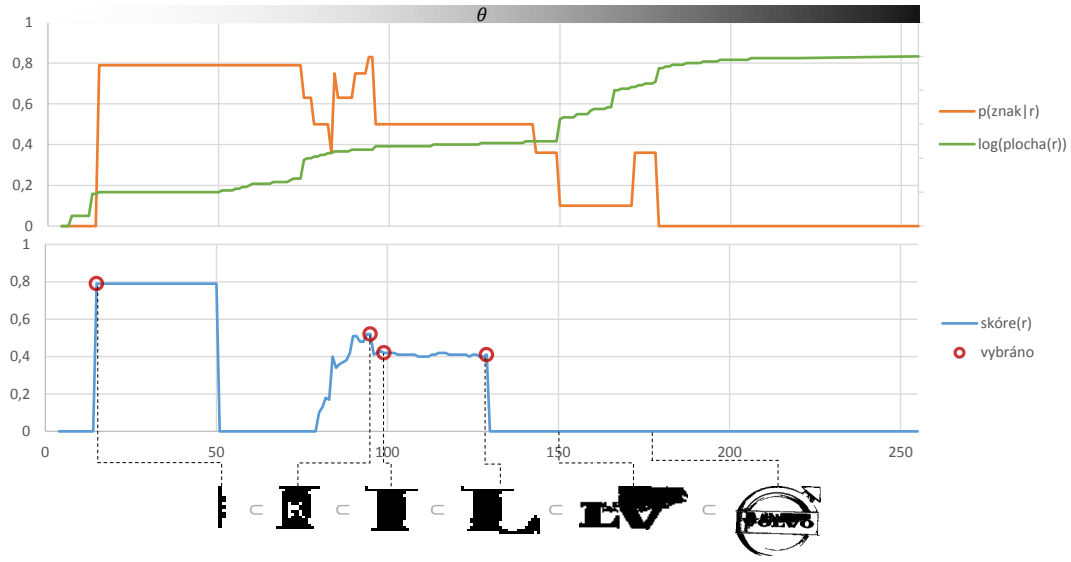
Obrázek 2.3. Ukázka ručně vybraných extremálních oblastí použitých pro trénování klasifikátoru.

Výstup klasifikátoru F je převeden na pravděpodobnost $p(\text{znak}|r) = \frac{1}{1+e^{-2\cdot F(r)}}$ [6]. Dále je pro oblast odhadnuta stabilita tak, že je zjištěna změna velikosti plochy $a' = \frac{a(r_{+20})-a(r)}{a(r)}$ kde r_{+20} je extremální oblast, která je nadmnožinou r a má práh $\theta_{+20} \geq \theta + 20$.

Skóre dané oblasti je stanoveno tak, aby kombinovalo výstup klasifikátoru se stabilitou oblasti. Na obrázku 2.4 je vidět, že $p(\text{znak}|r)$ bývá po určitou dobu konstantní, a tedy faktor změny plochy podpoří výběr oblasti, která má větší stabilitu co se týče změny tvaru.

$$skore(r) = p(\text{znak}|r) \cdot (1 - a')$$

Jako výsledná písmena jsou vybrány oblasti, kde nabývá skóre lokálního maxima s tím, že odstup lokálních maxim musí být alespoň $\Delta_{min} = 0,1$ a skóre musí nabývat hodnoty alespoň $skore_{min} = 0,2$ (experimentálně nastaveno).



Obrázek 2.4. Sekvence $p(\text{znak}|r)$ a $plocha(r)$, v logaritmickém měřítku podle prahu θ. $skore = p \cdot (1 - a')$ v sekvenci inkluze extremálních oblastí. Oblasti vybrané pro další fázi jsou označené kolečky.

Obrázek 2.5. Sjednocení všech oblastí vybraných pro formování řádků.

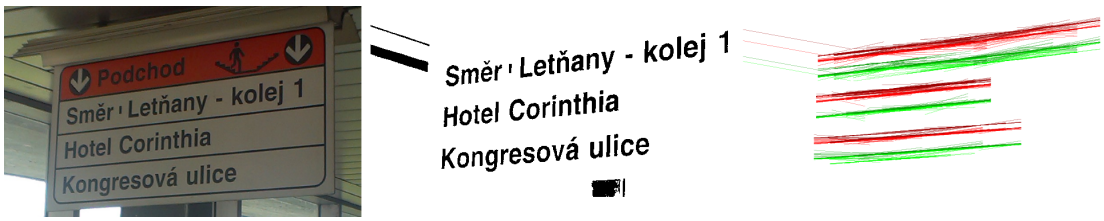
2.3 Tvoření hypotéz o řádcích

Sestavení písmen do řádků textu je založeno na modelu, který se opírá o princip písmové osnovy; tedy že horní a dolní okraje písma sedí na dvou dotažnicích (nahoře i dole).

Obrázek 2.6. Model řádky textu: linky horní dotažnice t_1, střední dotažnice t_2, základní dotažnice b_1, dolní dotažnice b_2.

Metoda[2] nejprve hledá mezi písmeny jednotlivé trojice, které modelu odpovídají. Protože takových trojic je mnoho, je procházení trojic prořezáno podmínkami na následující vlastnosti $\hat{v}$:

- vzdálenost písmen normalizovaná jejich výškou
- vzájemný poměr výšek
- vzájemné překrývání
- úhel svíraný těžišti tří písmen



Obrázek 2.7. Vstup (35670 extremálních oblastí). Vybrané extremální oblasti (71 kusů). Sestavené trojice (514 kusů, vykresleny pouze dotažnice)

Po nalezení všech trojic je třeba z nich vytvořit platné řádky. Parametry dolní dotažnice (b_1 nebo b_2) jsou odhadnuty proložením přímky[1]) dolními body oblastí algoritmem RANSAC[7]. Minimalizovaným kritériem je medián čtverců[8] vzdáleností bodů od dotažnice. Tím se zafixuje směrový parametr řádky a absolutní poloha b_1, b_2 a t_1, t_2 je určena posouváním směrnice do všech kombinací dvojic dolních, respektive horních okrajů oblastí tak, aby v celém řádku byl součet čtverců odchylek okrajů všech oblastí od dvou dotažnic nejmenší.

Platnost řádky se ověřuje podmínkami na výšku mezi linkami $t_1 - t_2$, $b_1 - t_2$ a $b_1 - b_2$. To vše v poměru k celkové výšce.

Dále se porovnává vzdálenost písmena (resp. jeho horního a dolního okraje) od linek pro danou řádku textu normalizovaná výškou řádky ($d(\tau, x)$). Obdobně je definovaná i vzdálenost dvou řádek textu $d(\tau_1, \tau_2)$[2], která, zjednodušeně řečeno, vypočte, jakou

[1]) Pro řádky z více písmen je prokládána parabola, která umožní zakřivení řádky.

maximální odchylku mají v nejzazších bodech spodní nebo dolní linky, a to při nejpříznivější možné záměně t_1 za t_2 a b_1 za b_2[1]).

2.3.1 Seskupování do řádků

Pro seskupování trojic do celých řádků jsem se rozhodl odchýlit se od původní metody a navrhl jsem algoritmus, který vytvoří prototypy podle vzájemných vzdáleností trojic. Teprve v mezích hotových prototypů se řeší rozdělení na možné hypotézy tak, aby se písmena nepřekrývala, nebo si nebyla moc vzdálená.

$$similar(\tau, X, \delta) = x \in X : d(\tau, x) \leq \delta$$

```
procedure prototypes(triplets, lineDist, letterDist)
  openList := triplets
  prototypes := empty set
  while |openList|>0
    candidate := argmax(x in openList ...
                        -> |similar(x, openList, lineDist)|)
    remove candidate from openList
    material := union(x in similar(candidate, openList, lineDist) ...
                      -> x.letters)
    if |material|=0
      break
    prototype := new Prototype(material)
    prototype.letters := similar(prototype, material, letterDist)
    for p in openList
      if |similar(prototype, p.letters, letterDist)|=|p.letters|
        remove p from openList
        continue
    if(not prototypes.contains(prototype))
      openList += prototype
      prototypes += prototype
  endwhile
  return prototypes
```

Obrázek 2.8. Oblasti použité k sestavení prototypu (černé a šedivé). Červenozelené linky jsou nalezený prototyp. Šedivá oblast (svislá pomlčka) po sestavení prototypu vyloučena, protože *nesedí* na linkách. Body použité k prokládání směrnice jsou červené čtverečky.

Takto nalezené prototypy mohou obsahovat sekvence oblastí, které neodpovídají podmínkám $\hat{v}$, proto jsou prototypy rozděleny na výsledné řádky.

```
procedure split(prototype)
  openList := ['']
  output := empty set
  for l in prototype.letters
```

[1]) Některé trojice patřící do stejné řádky nemusí mít shodné dotažnice. Např. slovo *hlt* nemá identifikovatelnou střední dotažnici ani dolní dotažnici, jiná slova s ním na řádce budou mít běžně střední a často i dolní dotažnice. Obdobně také při části řádky napsané verzálkami.

```
    for s in openList
      if v(s.back, l)              % ověření podmínek pro koncový znak
        s += l                     % OK
      else if |l.x-s.back.x|>3*prototype.h   % moc velký rozestup
        output += s                % dosavadní ukončíme
        openList -= s
        openList += l              % vytvořím novou hypotézu
      else                         % else např. překryv
        alternative := s           % vytvořím alternativu
        alternative.back = l
        openList += alternative
    endfor
  endfor
  output += openList
  return output
```

2.4 OCR

Převod nalezených řádek na text provádí Tesseract OCR[4]. Zvolil jsem tuto cestu, protože doposud popisovaná metoda dále provádí rozřezávání extremálních oblastí obsahující (hypoteticky) více znaků a především samotnou klasifikaci znaků metodou SVM. Tato metoda se mi neosvědčila (možná jsem používal špatně spočtené příznaky) a výsledkem bylo vždy příliš mnoho support vektorů, SVM se „přeučilo“ a reprezentace zabírala příliš mnoho místa.

Dále jsem experimentoval s Ferns klasifikátorem, který je účinný především na rozpoznávání textur. Ten ovšem občas nebyl schopen určit správné písmenko, což ve výsledku vedlo k jistotě, že v řádku bude něco špatně.

Volba Tesseractu vyplynula z chvály na jeho kvality coby otevřeného OCR a poměrně snadné implementace a snadné možnosti rozšíření o další jazyky prostým stažením balíčku z webu.

2.4.1 Rektifikace

Tesseract umožňuje zpracovat obrázek s příznakem režimu segmentace *Treat the image as a single text line*, tedy přesně případ pro předávání jednotlivých řádek. Jediné, co je potřeba, je řádek, který je v obrázku našikmo a možná prohnutý, narovnat vodorovně. K tomu se použije algebraická reprezentace dotažnic.

Práce s písmenem jako s extremální oblastí (podotýkám, že je z definice souvislá) implikuje, že znaky, které se skládají z více komponent jako například diakritika, znak % (popřípadě interpunkce, která není na dotažnicích) nejsou zpracovány. Proto jsem se rozhodl provádět projekci z původního obrázku do přímé řádky podle dotažnic, která pochopitelně takové znaky obsáhne. Pokud je text dostatečně odlišitelný od pozadí, tak nemá Tesseract problém jej přečíst.

V některých případech byl lepší výsledek pro čtení extremálních oblastí a jindy zase čtení z původního obrázku. Pro rozpoznávání textu na extremálních oblastech jsem naučil OCR Tesseract na písmenech, které tvoří jen jednu komponentu (např. `i` bez tečky). Na OCR projekcí používám předpřipravené balíčky pro daný jazyk.

Obrázek 2.9. Narovnané řádky s extremálními oblastmi (dole), a projekcemi z původního obrázku (nahoře). Vlevo byla větší úspešnost OCR na projekci. Vpravo na extremálních oblastech (původní obrázek ze sady ICDAR 2003[9]).

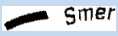
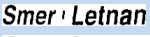
2.5 Kolize

Vzhledem k různým možným segmentacím písmen vznikne procedurou `split` z prototypu více řádek, které se obvykle překrývají. Také prototypy se mohou vzájemně překrývat.

Je tedy potřeba vybrat nejpravděpodobnější rozložení textů tak, aby se nepřekrývaly. Tesseract OCR kromě samotného textu poskytuje i míru jistoty, *confidence*, což je číslo v rozsahu od 0 do 100. Dále počítám s jazykovým modelem, kterým stanovím p_{lang}. Pro každou řádku vypočítám skóre:

$$skóre(s) = p_{lang}(s) \cdot confidence(s)$$

V ideálním případě by se scéna převedla na graf, kde uzly jsou řádky textu a hrany jsou mezi uzly, které se překrývají. Potom by stačilo použít algoritmus na hledání maximální vážené nezávislé množiny. Což je, bohužel, NP-úplný problém.

vstup OCR	přečtený text	*conf.*	p_{lang}	*skóre*
	`l emer`	53	0,014	0,75
	`Smer i Leman`	61	0,036	2,20
	`Smer Letnany kolej 1`	68	0,070	4,76

Tabulka 2.1. Výsledky OCR a skóre

Proto jsem použil jednoduchý (hladový) aproximační algoritmus:

```
procedure remove_conflicts(lines)
  while some conflict
    s := argmax(x in lines: x has some conflict ...
                  -> skore(x)/ ...
                       median(y in lines: x overlaps y -> skore(y)))
    lines -= z in lines: z overlaps s
  endwhile
  return lines
```

Pokud existuje konflikt, tak se najde řádka x s maximálním poměrem

$$skóre(x)/median(skore(y))$$

a ta se zachová, zatímco konfliktní se zahodí (y jsou řádky překrývající x).

2.5.1 Jazykový model

Jazykový model je funkce $p_{lang} : \Sigma \to \langle 0, 1\rangle$ a text o n znacích je $S = s^n \in \sigma^n \subset \Sigma$.

$$P(S) = P(s_1 s_2 \dots s_n) = P(s_1)P(s_2|s_1)P(s_3|s_1 s_2) \dots P(s_n|s_1 \dots s_{n-1})$$

Chtěl bych, aby $p_{lang}(S) = \sqrt[|S|]{P(S)}$, tedy průměrná[1]) pravděpodobnost na jeden znak řádky.

Jednotlivé podmíněné pravděpodobnosti pro texty všech délek ovšem nelze odhadnout ani uložit do paměti počítače, takže jsem se omezil na markovův řetězec 4. řádu.

$$p_{lang}(S) = \sqrt[|S|]{P(s_1)P(s_2|s_1)P(s_3|s_1s_2)\prod_{n=4}^{|S|}P(s_n|s_{n-3}s_{n-2}s_{n-1})}$$

Na to je potřeba tabulka všech kombinací čtveřic $P(\sigma_4|\sigma_1\sigma_2\sigma_3)$, což jsem udělal tak, že např. pro češtinu jsem si zafixoval abecedu $\sigma = \{a \dots z, 0, \perp\}$, tj. písmena, čísla a *ostatní*. Všechny znaky `a-Z` vč. diakritiky převedu na malé písmeno bez diakritiky. Čísla `0-9` na `0` a vše ostatní (vč. mezery) je $\perp$. To je celkem 28 různých symbolů. Dále text upravím tak, že ze začátku odstraním všechny $\perp$, a více $\perp$ za sebou nahradím jen jedním $\perp$, což mi umožňuje v tabulce čtveřic reprezentovat i trojice, dvojice a samostatné pravděpodobnosti nepodmíněných znaků ($P(\sigma|\perp\perp\perp) = P(\sigma)$).

Tabulka pro češtinu a jiné jazyky s abecedou `a-z` má tedy $28^4 = 614656$ hodnot, což při ukládání desetinného čísla jako `float`, 4 bajty na číslo, je necelých 2,5MB na jazykový model.

2.6 Výstup

Výstupem procesu je seznam řádek s textem včetně jejich geometrických vlastností a skóre.

Obrázek 2.10. Ukázka výstupu vykresleného přes původní obrázek

text	*skóre*
`Směr|Letňany - kolej 1`	7,4
`Hotel Corinthia`	3,5
`Kongresová ulice`	8,6

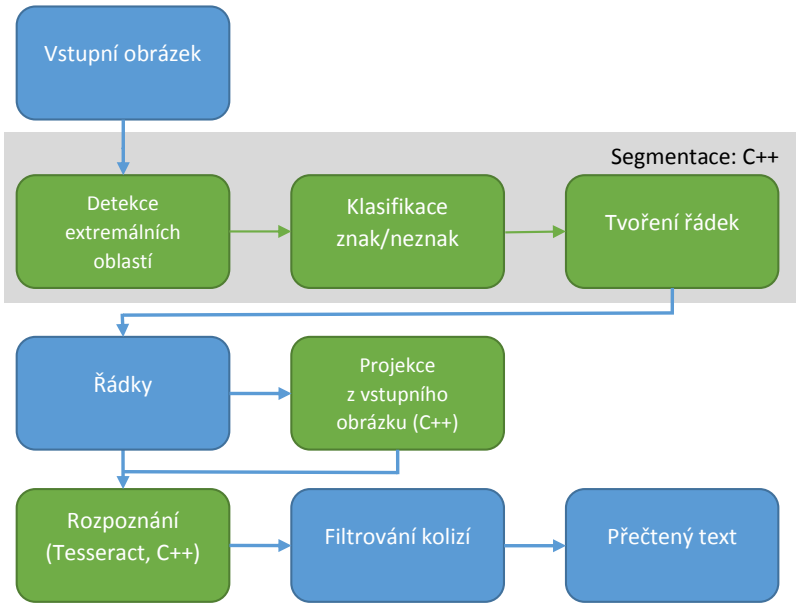
[1]) Jinak totiž platí, že $P(\texttt{Dob}) \geq P(\texttt{Dobrý}) \geq P(\texttt{Dobrý den})$, protože dílčí pravděpodobnosti jsou z defnice ≤ 1. Delší texty by tedy měly vždy nižší skóre než kratší.

Kapitola 3
Realizace

V Javě je možné využívat JNI (Java Native Interface), což znamená, že metody označené klíčovým slovem `native` jsou implementovány v nativním kódu. Ve statickém konstruktoru třídy s JNI načteme knihovnu pomocí `System.loadLibrary`, což načte příslušnou knihovnu (`.dll`/`.so` soubor).

Pokud se například má zpracovat 1Mpx obrázek, tak jej reprezentuji polem s milionem prvků. Dalvik VM[1]) je optimalizován především pro nízkou spotřebu paměti než pro výkon[2]). Takové množství operací[3]) pro zpracování obrázku je v nativním kódu řádově desetkrát rychlejší[10]. Android umožňuje integraci nativního kódu do aplikací pomocí balíčku Android NDK[4]).

Rozhodl jsem se tedy implementovat algoritmus dle Neumann a Matas[2–3] v C++ kvůli velkému množství zpracovávaných dat. Výsledné hypotézy o řádcích jsou již reprezentovány objekty v Javě. Pomocí Tesseract OCR jsou k nim doplněny texty. Případně je k řádku vytvořena projekce z původního obrázku (opět v C++) a Tesseractu je předána i ta. Řešení kolizí je implementováno v Javě, protože se jedná maximálně o nižší desítky objektů.



Obrázek 3.1. Diagram zpracování obrázku. Zelené jsou části implementované v nativním kódu (specifikace obr. 2.1).

V následujících sekcích popíšu, jak jsou jednotlivé fáze detailně řešeny, se speciálním ohledem na organizaci paměti.

[1]) Dalvik je implementace JVM pro Android
[2]) Oproti JVM pro desktopy, kde se rychlost řádově shoduje s rychlostí provádění nativního kódu
[3]) Byť triviálních, třeba jen obyčejný převod na negativní hodnoty pixelů v jednom průchodu
[4]) `https://developer.android.com/tools/sdk/ndk/index.html`

	čas [ms]
Java	500
C++	15

Tabulka 3.1. Čas převodu barevného obrázku 800 × 600px do stupňů šedi

3.1 Využití paměti

Android má nastavené limity pro haldu aplikace poměrně skromně. Je to obvykle 24 MB, podle zařízení[1]). Tento limit se vztahuje na objekty vytvořené v Javě. Nativní objekty se nevytváří na haldě JVM a programátor je zároveň zodpovědný za jejich uvolnění (narozdíl od automatické správy v Javě). I samotný OS Android má `android.graphics.Bitmap` řešeno tak, že pixely jsou uložené nativně a při volání metody `finalize` nebo `recycle` je uvolní. Je tedy potřeba si dávat větší pozor na uvolnění dynamicky alokované paměti.

Nativní řešení segmentace a OCR je vhodné, protože neukrajuje paměť aplikaci, která má přidělenou paměť takto k disposici pro své GUI.

Z `android.graphics.Bitmap` lze hodnoty pixelů získat do pole voláním jedné procedury, což je asi milionkrát rychlejší než získávat barvu jednotlivých pixelů pomocí metody `getPixel(int x,int y)`, a toto pole je jednodimenzionální a organizované po řádcích (*row-major order*). Proto ve svém kódu 2D objekty uvažuji v této reprezentaci. Pro převod souřadnic 2D obrázku (širokého w pixelů a vysokého h pixelů) do lineární podoby, tj. index do pole délky $w \cdot h$, používám jednoduchý převod $(x, y) \leftrightarrow lin$:

$$\begin{aligned} lin(x, y) &= x + y \cdot w \\ x(lin) &= lin \bmod w \\ y(lin) &= \lfloor lin/w \rfloor \end{aligned}$$

Veškeré indexování je od nuly, a $(0, 0)$ jsou souřadnice levého horního pixelu obrázku.

Pro potřeby ukládání různých bitmap, pracovních bitmap, masek a polí mám naalokovaný dostatečný kus paměti v jednom kuse, místo dílčí alokace a dealokace menších (až titěrných) polí. Lépe se tak zamezí úniku paměti (memory leak), a každé volání `malloc` zabírá trochu času i místa. Další důvod je lokalita přístupu a využití cache – mám zaručeno že má data jsou si blízko. Ono pole ve třídě `Cteni` je výmluvně se jmenující `int32_t* customArray5`, a má tedy velikost $5 \cdot 4 \cdot w \cdot h$ bajtů (tj. 20 MB na 1 Mpx). Dále v této práci budu popisovat, co se do pole ukládá[2]).

Také musím představit jednoduché definice typů, zavedené pro přehlednost kódu:

```
typedef uint8_t  GrayPixel;     // hodnota pixelu (stupeň šedi 0..255)
typedef uint32_t LinearPixAddr; // poloha pixelu v lineární reprezentaci
```

3.2 Extremální oblasti

V tuto chvíli mám obrázek načtený v `GrayPixel* raw`. Pohled na extremální oblasti obrázku při prahu θ jsou *černé* všechny pixely x, které mají hodnotu `raw[x]` $\leq \theta$. Ostatní jsou *bílé*. Po zvýšení prahu se oblasti mohou rozšířit a také mohou vzniknout

[1]) Starší zařízení 16 MB, novější 32 MB; odvíjí se i od rozlišení displeje kvůli nutnosti pracovat s většími *obrazy*. Pomocí `<appication android:largeHeap=true />` lze zažádat o více.

[2]) Laskavý čtenář zdrojového kódu se nemusí obávat; před místem použití paměti v `customArray5` je vytvořen nový ukazatel na správné místo, který se již jmenuje smysluplně.

nové. Pro identifikaci oblasti není potřeba evidovat uloženou množinu jejích pixelů, ale stačí souřadnice pixelu a s nejvyšší hodnotou `raw[a]` $= \theta$. Oblast se rekonstruuje postupným prohledáváním sousedících pixelů, které mají hodnotu $\leq \theta$, podobně jako nástroj *plechovka* v grafických editorech. Zjednodušený pseudokód zachycující postup:

```
procedure extremalRegions(raw)
  extremal_regions := []
  for t in 0..255
    for pix in (x in raw|x=t)
      er := pix union ("existing ERs touching pix")
      "track incrementally computable features"
      extremal_regions += er
    endfor
  endfor
  return extremal_regions
```

Pro procházení pixelů podle zvyšujícího se prahu jsou souřadnice pixelů nejprve setříděny podle rostoucí intenzity (pomocí counting sortu v lineárním čase). Toto pole se nemusí alokovat, ale použije se výše popsané univerzální pole:
`LinearPixAddr* sortedI = (LinearPixAddr*) ((void*) customArray5)`, čímž je první pětina z `customArray5` obsazená.

Pro sledování komponent souvislosti oblastí jsem použil algoritmus „union-find“, který pracuje s polem, ve kterém jsou uloženy indexy zpět do tohoto pole, což vyjadřuje stromovou strukturu. Kořen je označen jako odkaz sám na sebe (popř. nějakou speciální hodnotu):

```
procedure findComponent(LinearPixAddr* components, LinearPixAddr pix)
  while(pix is not root)
    pix := components[pix]
  return pix
```

Přidání pixelu `x` ke komponentě reprezentované kořenem `y` je jednoduché nastavení `components[x]=y`. Pro sledování komponent je tedy potřeba další pole:
`int32_t* components_tree = customArray5 + wh`, které ovšem začíná až za oblastí `sortedI`, a tímto je 40 % obecného pole obsazeno. Výše popsaný „union-find“ ovšem může tvořit poměrně hluboké stromy a jeho složitost je $O(n \log n)$. Zlepšit se lze přepojováním uzlů stromu blíže ke kořenu, což zrychlí následující dohledávání ($O(n \log \log n)$). To ovšem může porušit dílčí strom vnoření oblastí, který dále potřebujeme (2.2). Do `components_tree` tedy ukládám jen plnohodnotný strom postupného skládání oblastí, a pro sjednocování oblastí na aktuální úrovni θ se použije další pole:
`int32_t* top_components_tree = customArray5 + wh * 2`, přepojovaným „union find“.

Dále je potřeba sledovat, které *pixely* reprezentují extremální oblast (k nimž evidujeme inkrementálně počítané deskriptory 2.2.1). Na to stačí pole hodnot `true/false`
`bool* er_seeds_mask = (bool*) ((void*) (customArray5 + wh * 3))`.

V této fázi algoritmu tedy z `customArray5` využívám 65 %.

Deskriptory oblasti se ukládají do pole `Feature* features` na indexy odpovídající `er_seeds_mask[x] = true`.

```
struct Feature {
  LinearPixAddr parent; // 4B
  uint32_t area;        // 4B
  uint16_t perimeter, x1, y1, x2, y2; // 5*2B
  int16_t euler;        // 2B
```

```
    // podle puvodniho paperu by zde bylo jeste
    // std::deque<uint16_t> crossings, ktere alokuje dalsi pamet na halde
  }; // tj. sizeof(Feature) = 20B
```

Je potřeba mít tedy ještě pole těchto struktur, jež se alokuje samostatně a zabírá také 20 MB na 1 Mpx. Tj. společně s `customArray5` je potřeba 40 MB RAM na 1 Mpx obrázku.

Na závěr je vytvořeno pole indexů, které ve stromu extremálních oblastí reprezentují listy (tzn. nejmenší oblasti). Toho je dále využito při průchodu stromu od listů ke kořenu.

3.3 Klasifikace

Při hledání lokálních maxim skóre na každé cestě od listu ke kořenu stromu extremálních oblastí jsem využil vlastnost stromu, proto tedy nezpracovávám pro každý list jeho cestu ke kořenu individuálně, což by znamenalo zpracování oblastí blíže kořenu vícekrát (např. oblast s $\theta = 255$, tj. celý obrázek, úplně pokaždé), a mělo by složitost $O(n^2)$.

Pro extremální oblast vyhodnotím klasifikátor F a odhad růstu a' a vypočtu *skore*. Stav hledání maxima mám již ve *stavových polích* a při zpracování oblasti se z nich vyčte stav „předaný“ rodičem. Všechna tato pole mají opět délku `wh`, tedy počet pixelů v obrázku, a lze je tedy indexovat identifikátorem extremální oblasti.

Při průchodu od listů ke kořenu používám:

- `LinearPixAddr* lookahead_shortcut`: `lookahead_shortcut[x]` obsahuje pro danou oblast s prahem $\theta =$ `raw[x]` identifikátor nadoblasti s $\theta_{+20} = \theta + 20$, pomocí které se stanoví nárůst velikosti $a'(x)$. Při nalezení `lookahead_shortcut` pro `x` se výsledek předá i do `lookahead_shortcut[features[x].parent]`, což se využije při hledání nadoblasti rodiče tak, že nebude potřeba tak dlouho iterovat.
- `float* grow_ratio`. Samotné hodnoty a'.

Pro hledání extrémů je vhodnější přístup od kořene (tj. celý obrázek) k menším oblastem, protože každá oblast má jen jednu nadoblast (zatímco může mít více podoblastí):

- `float* stack_low`, `float* stack_high` obsahují meze, ve kterých se pohybuje *skore*. Slouží k vyhodnocení podmínky významnosti extrému, tedy aby byl odstup alespoň Δ_{min} od lokálních maxim.
- `int32_t* stack_max_score_idx` určuje oblast, kde *skore* dosáhlo maxima. Že se jedná o dostatečný extrém zjistíme, až *skore* nějaké podoblasti dostatečně klesne.
- `bool* stack_state` nabývá hodnoty `AB_WAITING_FOR_DECREASE`, pokud nějaká oblast již měla větší *skore* než `stack_low` $+ \Delta_{min}$.
  Jinak je `AB_WAITING_FOR_HIGH_ENOUGH`.

Při zpracování oblasti `x` mohu vycházet z hodnot `stack_..[x.parent]`. Tedy kromě kořene `top` (tj. celý obrázek). Protože vím, že ten nikdy nechci považovat za vybranou oblast (potenciální písmeno), tedy pro něj inicializuji:

- `stack_high[top]` = `stack_low[top]` = p_{min}
- `stack_state[top] = AB_WAITING_FOR_HIGH_ENOUGH`
- `stack_max_score_idx[top] = -1`

Za předpokladu, že `float` zabírá 4 B[1]), tato pole zabírají 85 % `customArray5`[2]).

[1]) Kontrola pomocí `assert(sizeof(float) == 4)`
[2]) Kvůli využití paměti v této fázi jsem stanovil velikost `customArray5` na 20 B na pixel

3.4 Enumerace písmen

Z celkového počtu oblastí (řádově desetitisíce) je klasifikátorem typicky vybráno necelé procento (řádově nižší stovky). Z těchto oblastí se formují řádky, při tom se pracuje s těžištěm znaku, které není obsaženo v inkrementálně počítaných deskriptorech[1]). Sestavené řádky se převedou na bitmapu sjednocením extremálních oblastí, které daný řádek tvoří. Tedy je potřeba mít vybrané oblasti, doposud reprezentované souřadnicemi pixelu s hodnotou θ, jako bitmapu.

```
struct ExtractedLetter {
  Feature * const feat;       // ukazatel do pole Feature* features
  bool * const bitmap;        // ukazatel do pole customArray5
  pntf2d centr;               // těžiště: struct pntf2d { float x,y; }
  ExtractedLetter * next;     // ukazatel na nejbližší písmeno vpravo
  LinearPixAddr const seed;   // identifikace oblasti
}
```

Pro uložení bitmap se využívá `customArray5`, a to až do $w * h * 4 * 4$ bajtů (tj. zbude ještě minimálně $w * h * 4$ bajtů volných). Bitmapa oblasti má rozměry podle ohraničujícího rámečku. Kdyby se však obrázek skládal pouze z tenkých úhlopříčných čar (a ty by prošly klasifikátorem), tak by mohlo dojít k vyčerpání přiděleného prostoru (tj. bitmapy s *písmeny* by měly 16 krát větší plochu než původní obrázek), nicméně za předpokladu nalezení běžných písmen se toto nestane.

Pro samotné zjištění pixelů, které v mezích ohraničujícího rámečku náleží oblasti, se používá jednoduchý algoritmus: Pokud pixel `x` má `raw[x]` $> \theta$, tak nenáleží do oblasti. Jinak se zjišťuje, jestli patří pixel do oblasti, jež je podmnožinou vykreslované oblasti pomocí iterování `x = features[x].parent`.

Souřadnice pixelů (x, y), náležejících do oblasti, se sčítají a na závěr je zjištěno těžiště $\frac{\sum(x,y)}{a}$.

3.5 Tvoření hypotéz o řádcích

Vektor prvků `ExtractedLetter` se setřídí, aby težiště šla postupně zleva doprava. Poté se podle tohoto pořadí nastaví ukazatele `ExtractedLetter.next` jako spojový seznam. Následně se vektor setřídí shora dolů[2]) a začnou se tvořit řádky z trojic.

3.5.1 Trojice

Sestavování všech trojic vyhovující podmínkám $\hat{v}$ je prořezáno pomocí omezení maximální mezery mezi následujícími znaky - `normalizedHorizontalDistance`. Pokud se vzdálenost následujících znaků zvýší na čtyřnásobek jejich výšky, není už potřeba další kombinace, protože mezera se bude jedině zvětšovat. (Díky tomu, že `ExtractedLetter.next` nabídne znak, který je ještě více vpravo.)

```
procedure buildTriplets(extractedLetters)
  triplets := []
  for l1 in extractedLetters
    for(l2 = l1.next;l2!=NULL;l2=l2.next)
```

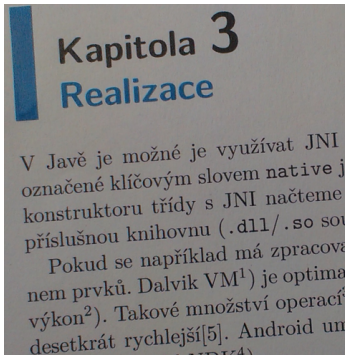
[1]) I když by mohlo být, protože by stačilo jen sčítat (x, y) přidávaných pixelů, a těžiště by potom bylo $(x/a, y/a)$.

[2]) Pokud sestavování trvá příliš dlouho, tak se přeruší, a tímto postupem jsou alespoň některé řádky nahoře kompletní.

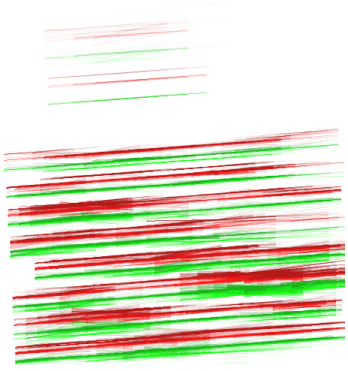
```
        if normalizedHorizontalDistance(l1,l2)>4
          break
        for(l3 = l2.next;l3!=NULL;l3=l3.next)
          if normalizedHorizontalDistance(l2,l3)>4
            break
          TLine line = new TLine(l1,l2,l3)
          if(v(line))
            triplets += line
        endfor
      endfor
    endfor
    return triplets
```

Dále jsem vypozoroval, že při zpracování obrázku s množstvím řádek souvislého textu (přestože metoda je určená spíše pro kratší texty v netextové scéně, stát se to může) je díky blízkosti řádek nalezeno enormní množství trojic, což následně způsobí dlouhé tvoření prototypů a řádků.

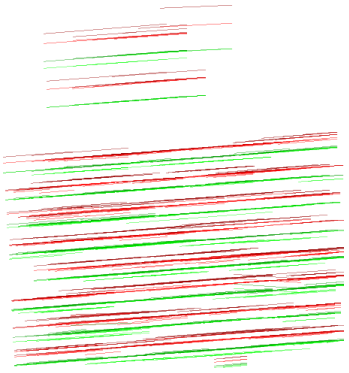
Experimentálně jsem se rozhodl tento případ identifikovat tím, že počet nalezených trojic je 10násobný oproti počtu použitých písmen. V takovém případě se projdou všechny trojice a z jejich směrnic k určím medián $\tilde{k}$. Z trojic, které mají stejné počáteční písmeno, je vybrána jen jedna s $\min \left|k - \tilde{k}\right|$.



Obrázek 3.2. Obrázek s textem (772 vybraných písmen)

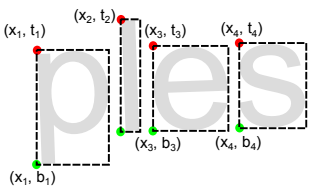


Obrázek 3.3. Sestaveno 41260 trojic



Obrázek 3.4. Omezeno na 397 trojic, $\tilde{k} = -0{,}075$

3.5.2 Geometrické parametry



Obrázek 3.5. Klíčové body oblastí pro odhadnutí parametrů řádky. ($n = 4$)

Hledaným parametrem řádky je směrnice k jejích dotažnic: $y = k \cdot x + c$. Dotažnice jsou rovnoběžné, jejich rovnice se liší tedy pouze konstantou c. Řádka má n písmen,

jejichž levé dolní rohy jsou (x_i, b_i) a levé horní (x_i, t_i). Ideálně bych vyřešil:

$$A \cdot K = B$$
$$\begin{pmatrix} x_1 & 1 \\ \vdots & 1 \\ x_n & 1 \end{pmatrix} \cdot \begin{pmatrix} k \\ c \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

Jenomže se jedná o přeurčenou soustavu ($n \geq 3$), která typicky nemá řešení. Parametry jsou odhadnuty minimalizací mediánu čtverců, tj.

$$\text{argmin}_{K \in \mathbb{R}^m} \left(\text{median} \, (A \cdot K - B)^2 \right)$$

Pro hledání rovnice dotažnic jako polynom 2. stupně (lze snadno zobecnit i na vyšší):

$$y = c + \sum_{i=1}^{2} k_i x^i$$
$$A = \begin{pmatrix} x_1 & x_1^2 & 1 \\ x_2 & x_2^2 & 1 \\ \vdots & \vdots & 1 \\ x_n & x_n^2 & 1 \end{pmatrix}$$
$$K = \begin{pmatrix} k_1 & k_2 & c \end{pmatrix}'$$

V případě, že je použito málo písmenek, používám k odhadu i horní body (x_i, t_i). Potom pro dotažnici jako přímku[1]) jsou matice:

$$A = \begin{pmatrix} x_1 & 1 & 0 \\ x_1 & 0 & 1 \\ \vdots & 1 & 0 \\ \vdots & 0 & 1 \\ x_n & 1 & 0 \\ x_n & 0 & 1 \end{pmatrix}$$
$$K = \begin{pmatrix} k & c_b & c_t \end{pmatrix}'$$
$$B = \begin{pmatrix} b_1 & t_1 & \dots & \dots & b_n & t_n \end{pmatrix}'$$

Směrnice k (resp. k_i) jsou nalezeny algoritmem RANSAC: ze soustavy je opakovaně náhodně vybráno m (počet neznámých, tj. počet prvků K) rovnic. Pro ně se vypočtou neznámé K z rovnice $A' \cdot K = B'$ (A' je již regulární) a vyhodnotí se kritérium pro celou soustavu.

K řešení soustavy rovnic jsem se rozhodl použít Cramerovo pravidlo. A to tak, že jsem si vygeneroval kód pro $m = \{2, 3, 4\}$, tedy sice determinanty počítám se složitostí $m!$, avšak nepotřebuji alokovat žádnou paměť, ani podmíněné skoky CPU. Už pro $m = 4$ je kód trochu kuriosní (inu, faktoriál), ale nakonec se řeší pouze soustavy s $m = 3$ a s rychlostí této operace žádný problém není[2]).

Konstanty c pro dolní a horní dotažnice (tedy t_1, t_2 a b_1, b_2) jsou nalezeny prostým vyzkoušením vedení křivek všemi páry klíčových bodů $(x_i, t_i), (x_j, t_j)$, respektive $(x_i, b_i), (x_j, b_j)$ po zafixování směrnice k (nebo k_1, k_2). Výsledkem jsou ty s nejmenším součtem čtverců odchylek od klíčových bodů.

[1]) Pro parabolu jsou obdobné, ovšem horní body používám jen pro nižší počty znaků, a pro vyšší počty znaků, kdy už dovoluji parabolu, zase nepoužítám horní body.

[2]) Mohlo by být zajímavé změřit, jak rychle se reálně na CPU vypočte determinant podle definice, ovšem s plným využitím pipeline, nebo podle nějaké n^3 metody...

3.5.3 Rektifikace

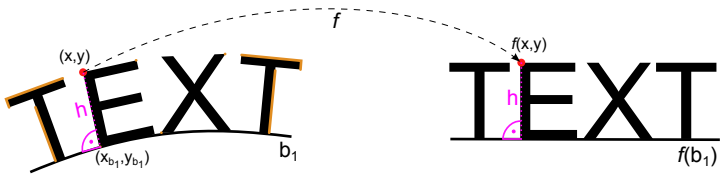
Převod z prostoru obrázku na rovnou linku definuje mapování f, které pracuje se základní dotažnicí b_1:

$$f\begin{pmatrix}x\\y\end{pmatrix} = \begin{pmatrix}x_{b_1}\\y_{b_1}+h\end{pmatrix}, \text{kde}$$

$$x_{b_1} = \text{argmin}_{x'_{b_1}\in\mathbb{R}}\left|\begin{pmatrix}x\\y\end{pmatrix} - \begin{pmatrix}x'_{b_1}\\b_1(x'_{b_1})\end{pmatrix}\right|,$$

$$y_{b_1} = b_1(x_{b_1}),$$

$$h = \begin{cases} +\left|\begin{pmatrix}x\\y\end{pmatrix} - \begin{pmatrix}x_{b_1}\\y_{b_1}\end{pmatrix}\right| & \text{pro } y_{b_1} \le y, \\ -\left|\begin{pmatrix}x\\y\end{pmatrix} - \begin{pmatrix}x_{b_1}\\y_{b_1}\end{pmatrix}\right| & \text{pro } y_{b_1} > y. \end{cases}$$



Obrázek 3.6. Rektifikace: převod z prostoru obrázku na rovnou linku mapováním f. Oranžově jsou vyznačené body, které jsou transformovány pro zjištění rozměrů v narovnaném prostoru.

Hledání nejbližšího bodu x_{b_1} na parabole jsem vyřešil půlením intervalu, kdy počáteční interval nastavím $(x - h_{max}, x + h_{max})$. To funguje za předpokladu, že strmost dotažnice je menší než 100 %, což je realistický předpoklad. Iterace se zastaví, pokud je odhad v mezích 1 pixelu.

Pomocí f transformuji jen horní a dolní okraje písmen, a v případě prvního a posledního písmena řádky i jeho stranu směřující pryč od textu (oranžově vyznačené v obrázku 3.6). Z těchto bodů mohu stanovit extrémní souřadnice a tedy výšku, šířku a relativní nulu rektifikované bitmapy.

Pro vytvoření bitmapy pixel po pixelu používám inverzní funkci f^{-1}, která dělá zpětnou projekci do souřadnic původního obrázku. Výslednou hodnotu pixelu získám bilineární interpolací (protože f^{-1} z celočíselné polohy v cíli vypočítá desetinnou polohu v původním obrázku). Pro f^{-1} je potřeba derivace dotažnice b'_1, což je konstanta (pro přímku) nebo derivace polynomu (resp. paraboly). Jedná se tedy o jednušší převod než f, což se hodí vzhledem k mnohem většímu počtu volání.

$$f^{-1}\begin{pmatrix}x\\h\end{pmatrix} = \begin{pmatrix}x + h\cdot\sin(\alpha)\\b_1(x) + h\cdot\cos(\alpha)\end{pmatrix},$$

$$\alpha = -\text{atan}(b'_1(x))$$

Jedná-li se o projekci z původního obrázku, bere se přímo intenzita pixelu na daných souřadnicích. V případě rektifikace extremálních oblastí využívám `(bool *) ExtractedLetter.bitmap` s tím, že `true` je 0 (tj. černá) a `false` 255 (tj. bílá). Na základě souřadnic se stačí dívat do maximálně dvou extremálních oblastí řádky, jejichž ohraničující rámeček je obsahuje.

Převod také škáluje obrázek tak, aby výsledná bitmapa byla 30px vysoká – což je dostačující pro Tesseract, aby byl schopen pracovat. Pole bitmapy je dynamicky alokováno a je potřeba je po využití pro OCR uvolnit. Rozhodl jsem se je neukládat do `customArray5`, aby mohlo být využito i poté, co je segmentován již jiný obrázek.

3.6 Předání skrze JNI do Javy

Nyní již jsou řádky kompletní a je potřeba je předat do Javy. Jedná se o pole řádek, kde každá řádka má

- rektifikovanou bitmapu (`GrayPixel *`),
- rozměry bitmapy: w, h,
- parametry křivek dotažnic k_i (podle tvaru $i = 1$ přímka, $i = \{1, 2\}$ pro parabolu),
- konstanty absolutních poloh dotažnic c_{t_1}, c_{t_2}, c_{b_1} a c_{b_2}

Rozhodl jsem se z těchto informací sestavit řetězec a ten převést na objekt v Javě.

```
public class CteniCppApi {
  static {  System.loadLibrary("ctenicpp"); }
  private static native String nativeSegmentationPixArr(long thiz,
                                  int[] pixels, int pixW, int pixH);
  ...
}

JNIEXPORT jstring JNICALL Java_CteniCppApi_nativeSegmentationPixArr
                  (JNIEnv * env, jclass cl, jlong thiz,
                  jintArray pixels, jint pixW, jint pixH) {
    // v javě si držím ukazatel na instanci Cteni
    // a předám ho pomocí jlong thiz
    Cteni * ct = (Cteni *) thiz;
    assert(sizeof(GrayPixel)<sizeof(jint));
    jint *pixelsArr = env->GetIntArrayElements(pixels, NULL);
    GrayPixel* img = (GrayPixel*) (void*) pixelsArr;
    for (int i = 0; i < pixW * pixH; ++i)
    {  // převod z barvy na stupně šedi
        jint argb = pixelsArr[i];
        GrayPixel col = ((((argb >> 16) & 0xFF)
                + ((argb >> 8) & 0xFF) + ((argb) & 0xFF)) / 3);
        img[i] = col;
    }
    ct->resize(pixW, pixH);
    ct->zpracuj(img, false);
    env->ReleaseIntArrayElements(pixels, pixelsArr, 0);
    std::string sout = "";
    for (auto & x : ct->outStrs)
        sout += x + "\n";
    return env->NewStringUTF(sout.c_str());
}
```

Toto je ukázka nativní metody v Javě a její protikus v C++, kde se pole barev kódovaných jako `int` převede do stupňů šedi (v tom samém poli) a zavolá `Cteni::zpracuj`. Tam se provedou všechny úlohy popsané výše a do `vector<std::string> outStrs` střádá textové reprezentace řádek, popřípadě i jiné údaje o běhu. Ty se nakonec spojí do jednoho dlouhého řetězce a převedou na `java.lang.String`.

O obsluhu JNI se tedy v Javě stará třída `CteniCppApi`, která nabízí různé metody pro segmentaci např. jen části obrázku, možnost obrázek transponovat, obrátit horizontálně i vertikálně. Řádek textu reprezentuje `CteniCppApi.TLine` a to vč. adresy paměti s rektifikovanou bitmapou extremálních oblastí `long patchAddr`.

rozlišení	čas [ms]
450 × 270	500
900 × 540	2300

Tabulka 3.2. Typický čas segmentace jednoho kanálu na telefonu HTC Desire X. Zpracovává se i negativ, tedy je potřeba počítat s dvojnásobkem času.

Statická metoda `projectPatch(Bitmap scene, TLine tl, float scale)` provádí projekci řádku z původního obrázku.

3.7 Tesseract OCR

Pro využití knihovny Tesseract v Androidu jsem použil připravený projekt tess-two[1]), který poskytuje třídu `TessBaseAPI`.

```
TessBaseAPI tesseract = new TessBaseAPI();
//načte soubor "/storage/sdcard0/cteni/tessdata/ces.traineddata"
tesseract.init("/storage/sdcard0/cteni","ces",TessBaseAPI.OEM_DEFAULT);
tesseract.setPageSegMode(TessBaseAPI.PageSegMode.PSM_SINGLE_LINE);
tesseract.setImage(rectifiedLineBitmap);
String text = tesseract.getUTF8Text();
tesseract.end();
```

Ukázka kódu s inicializací OCR, nastavení vstupu do režimu *jeden řádek*, předání obrázku a získání výsledného textu.

fáze	čas [ms]
inicializace	2000
rozpoznávání	80–1000

Tabulka 3.3. Typické časy fází Tesseract OCR na telefonu HTC Desire X.

Trochu zákeřné je, že segmentované řádky se zřetelně čitelným textem jsou přečteny do řádově sta milisekund, zatímco hůře čitelné řádky, popřípadě *řádky*, kde není vůbec žádný text (ale např. křoví), trvají zpracovat někdy i vteřiny. Bohužel, není možnost Tesseract nastavit tak, aby řádky které, se zdají problematické nezpracovával.

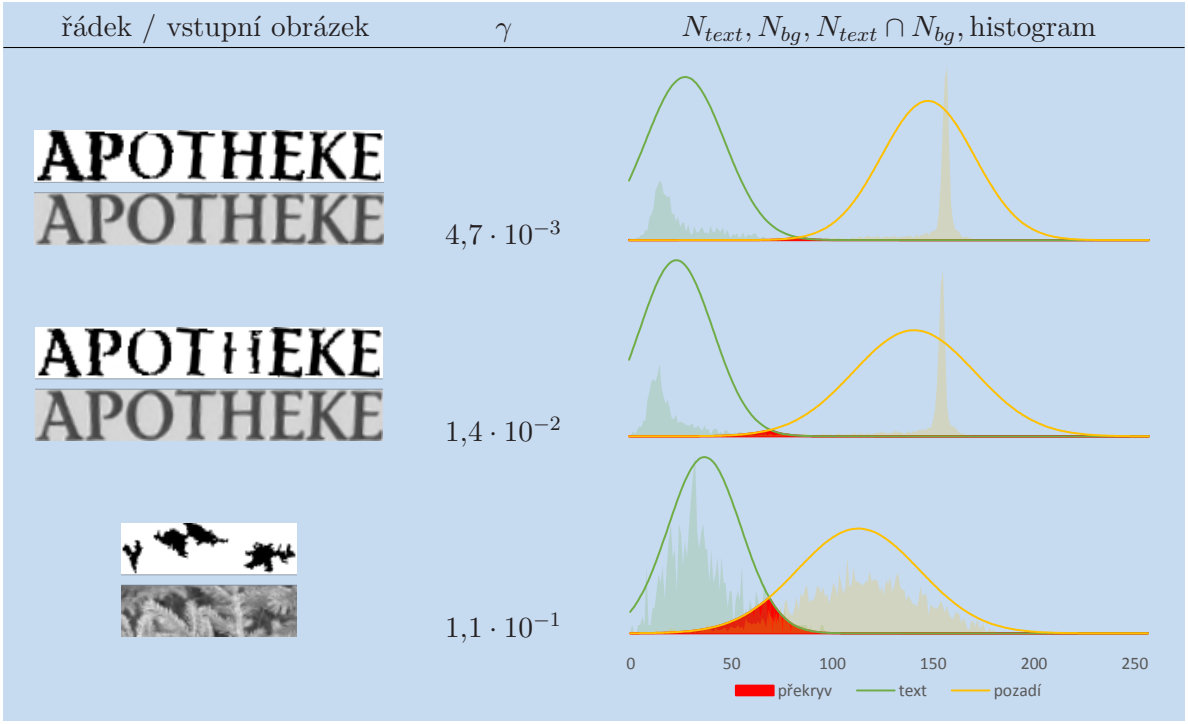
3.7.1 Třídění řádků

Vzhledem k dlouhému přemýšlení OCR nad špatně identifikovanými řádky je žádoucí nejprve zpracovat čitelnější řádky, aby bylo případně možné uživateli prezentovat částečný výsledek, a čtení méně slibných řádek nechat nakonec. Snažil jsem se vymyslet nějaké vhodné kritérium založené na počtu znaků v řádku, jejich rozestupech a celkové šířce.

Nakonec se nejlépe osvědčilo následující: Během rektifikace řádku se z bitmap extremálních oblastí zjišťuje, zda-li má být pixel černý (text) nebo bílý (pozadí). Z pixelů pozadí a textu jsou odhadnuty parametry normálního rozdělení $N_{text}(\mu_{text}, \sigma^2_{text})$ a $N_{bg}(\mu_{bg}, \sigma^2_{bg})$. Je spočten překryv těchto rozdělení $\gamma = N_{text} \cap N_{bg}$ na rozsahu $\langle 0, 255 \rangle$.

Při výběru vstupu pro OCR jsou nejprve vybrány řádky s nižším γ, protože pravděpodobněji obsahují korektně vytažený text. Toto kritérium dobře funguje i co se týče výběru nejlepší segmentace v rámci jednoho prototypu, viz tabulka 3.4.

[1]) `https://github.com/rmtheis/tess-two`, vč. instrukcí k načtení jako *library* projekt v Eclipse. Tesseract OCR i tess-two jsou k disposici pod licencí **Apache License 2.0**



Tabulka 3.4. Kritérium odstupu textu od pozadí $\gamma = N_{text} \cap N_{bg}$ určující pořadí předání řádek OCR.

3.8 Jazykový model

Podle 2.5.1 je počítána pravděpodobnost přečteného textu ve třídě `Language4`, která je inicializovaná pro konkrétní jazyk.

Text znak po znaku převádí pomocí tabulky `charClass`[1]) na kód třídy $\langle 0, \texttt{numclass})$. `numclass` je například pro češtinu 28, tj. 26 písmen, číslice a $\perp$. $\perp$ je symbol pro *ostatní znaky* (reprezentovaný číselnou hodnotou proměnné `other`). Více $\perp$ za sebou je převedeno jen na jeden $\perp$. Pravděpodovnost je tedy počítána jen se slovy a čísly, bez ohledu na interpunkci a např. párovost závorek.

0123456789	aAáÁ	bB	cCčČ	dDďĎ	eEéÉěĚ	...	zZžŽ	$\perp$
0	1	2	3	4	5	...	26	27

Tabulka 3.5. Převodní tabulka znaků `charClass` pro češtinu. Vše co není uvedeno (mezery, interpunkce) se považuje za $\perp$ (kód třídy 27).

Klíč K pro čtveřici po sobě jdoucích znaků *abcd* je definován:

$$k(x) = \begin{cases} \texttt{charClass}[x] & \text{pokud } \texttt{charClass.contains}(x), \\ \texttt{other} & \text{jinak.} \end{cases}$$

$$K(abcd) = ((k(a) \cdot \texttt{numclass} + k(b)) \cdot \texttt{numclass} + k(c)) \cdot \texttt{numclass} + k(d)$$

Lze si jej představit jako převod do číselné soustavy se základem `numclass`. Nabývá tedy hodnot $\langle 0, \texttt{numclass}^4)$. Výhodou této reprezentace je, že máme-li $K(abcd)$, tak z něj lze odvodit $K(bcde)$:

$$K(bcde) = (K(abcd) \cdot \texttt{numclass} + k(e)) \bmod \texttt{numclass}^4$$

[1]) `java.util.Map<Character,Byte> charClass`

K	`probs[`K`]`	popis
$K(abcd)$	$P(d\|abc)$	P(po abc následuje d)
$K(\perp\perp ab)$	$P(b\|\perp a)$	
$K(\perp\perp\perp a)$	$P(a\|\perp)$	P(a na začátku slova)

Tabulka 3.6. Významy pro možné typy klíče v poli pravděpodobností.

Klíč K se použije jako index do pole `float[] probs`. Výpočet pravděpodobnosti lze provést počítáním klíčů jako *rolling hash*:

```
K0 := K(other,other,other,other)
mod := numclass*numclass*numclass*numclass
procedure P(text)
  K := K0
  P := 1
  for char in text
    K := (K*numclass+k(char))%mod
    P *= probs[K]
  return P^(1/length(text)) // průměrná P na znak
```

Jazykový model jsem vytvářel podle stažených úryvků textu z Wikipedie. Model je uložen v souboru s příponou `.lng4` a obsahuje serializované položky načítané takto:

```
FileInputStream fis = new FileInputStream(file);
ObjectInputStream ois = new ObjectInputStream(fis);
numclass = ois.readInt(); // počet tříd (pro češtinu např. 28)
other = ois.readByte();   // klíč třídy "ostatní"
charClass = (java.util.Map<Character, Byte>) ois.readObject();
probs = (float[]) ois.readObject();  // pole s numclass^4 prvky
Date date = (java.util.Date) ois.readObject(); // datum vytvoření
ois.close(); fis.close();
```

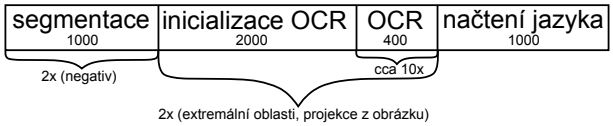
Načtení pole `float[28^4]` proběhne na mém telefonu HTC Desire X za cca jednu vteřinu. Zajímavostí je, že pokud se aplikace spustí v ladícím (*debug*) režimu, tak to trvá minutu.

Problém tohoto přístupu je u jazyků, které mají mnohem větší počet tříd písmen než 28. Už například ruština má tříd 34, což znamená 5,3 MB soubor (oproti 2,5 MB pro 28 tříd).

Další omezení je maximální index pole cca 2^{31}, toho by se dosáhlo při více než 215 třídách písmen, a velikost by byla 8,5 GB. Pro takové jazyky by bylo potřeba vytvořit jiný jazykový model.

Kapitola 4
Asynchronní implementace

V kapitole 3 jsem popsal úkony prováděné pro získání textu z obrázku. Obrázek 4.1 zachycuje orientační časy trvání časově náročných fází. Vyplývá z něj, že pro zpracování jednoho takového obrázku by bylo potřeba 15 vteřin, a to v případě dobře provedené segmentace a bezproblémového OCR. V některých komplikovaných případech se stává, že je mnoho řádek předáno OCR fázi (stránka s textem, nebo je za text považována nějaká pravidelná textura jako třeba okna domu nebo plot), a/nebo OCR čte jednotlivé řádky i několik vteřin.

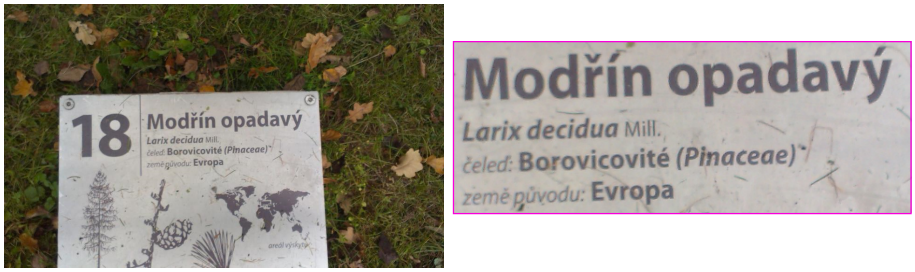


Obrázek 4.1. Orietační časy (v milisekundách) trvání kritických fází zpracování na telefonu HTC Desire X pro obrázek 800 × 600px

Dále je fakt, že pokud je na vstupním obrázku zřetelný velký text, je možné jej úspěšně rozpoznat už při rozlišení 0,1 Mpx.

Mpx	segment. [ms]	ř.	Σ* [ms]	přečteno
0,1	400+220	3	1700	Modřín opadavý
0,25	1000+650	4	2800	Modřín opadavý, Evropa
0,5	1800+1300	10	10200	Modřín opadavý, Lan'x decidua Mill, Borovicovité (Pinaceae), Evropa
1	3200+2600	13	9700	Modřín opadavý, Larix decidua Mill, Borovicovité (Pinaceae), Evropa

Tabulka 4.1. Doba segmentace, počet nalezených řádek (ř.), přečtený text a celková doba zpracování Σ (* ovšem v asynchronním zpracování, které teprve popíši) v závislosti na rozlišení vstupního obrázku 4.2. V počtu řádek jsou zahrnuté i falešné detekce z oblasti trávy.

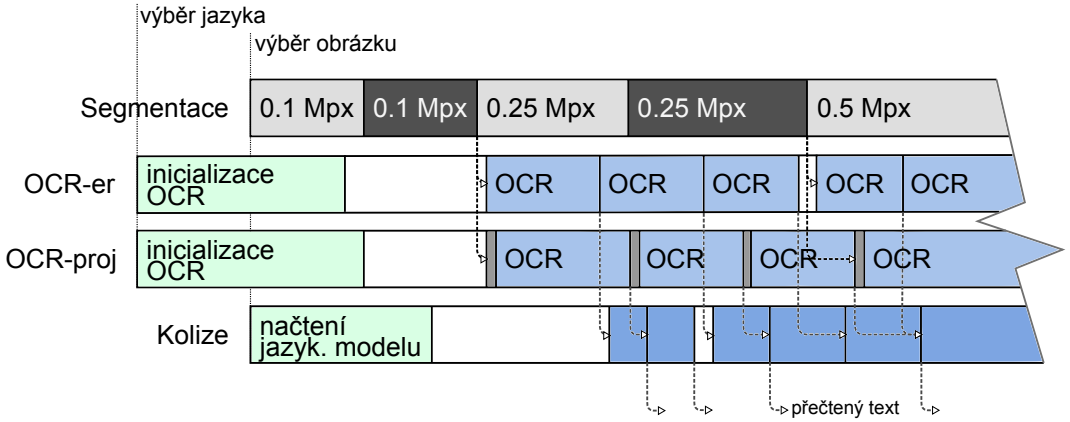


Obrázek 4.2. Ukázkový obrázek s texty v různé velikosti. Vpravo výřez části s textem. Obrázek byl však zpracováván celý, včetně trávy a listí.

Čekání na výsledek (v lepším případě) čtvrt minuty není z hlediska uživatelské přívětivosti ideální, a proto jsem navrhl řešení, které obrázek zpracovává postupně v menším

rozlišení a přechází k vyššímu. Průběžné výsledky předává uživateli. Z tabulky 4.1 vyplývá, že takové řešení umožňuje mít první smysluplný text (i když neúplný) během pár vteřin. Takové zpracování probíhá ve více vláknech, což využije, dnes již běžné, vícejádrové procesory v telefonech. Činnosti jsou rozděleny na:

- Segmentace obrazu
- OCR řádků
- Filtrování kolizí



Obrázek 4.3. Koncepce asynchronního zpracování

Vlivem režie vláken a předávání zpráv lze očekávat, že zpracování bude v součtu trvat déle. Oproti tomu je možné uživatele průběžně informovat o stavu a poskytovat částečné výsledky, takže je možné celé zpracování pojmout jako velmi dlouhou operaci, při které záleží na uživateli, jak dlouho bude čekat na relevantní výsledek. Dále je možné po zvolení jazyka pro OCR a jazykový model inicializovat potřebné struktury (což trvá i několik vteřin), zatímco uživatel ještě vybírá vstupní obrázek ke zpracování.

O celý proces se stará třída `AsyncCteni`, která je abstraktní (je nutné poskytnout metodu `layout`):

```
// "hlavičky" pro základní použití
public abstract class AsyncCteni {

  public AsyncCteni(String dir);

/**
  * Volá se při každém novém přečteném řádku. Pokud
  * {@code confidentCoverageCounter} vzroste, je přečtena celá scéna.
  * @param prunedResult -- nalezené přečtené řádky
  * @param confidentCoverageCounter
  * @param tooMuchText -- příznak nekompletní segmentace
  */
protected abstract void layout(List<TLineOCR> prunedResult,
                    int confidentCoverageCounter, boolean tooMuchText);

/**
  * Začne zpracovávat poskytnutou bitmapu.
  * @param input
  */
public final synchronized void execute(Bitmap input);
```

```
/**
  * Nastaví jazyk pro OCR projekcí z obrázku a pro jazykový model.
  * Musí existovat soubor dir/tessdata/lang.traineddata
  * a dir/lang.lng4
  * @param lang
  */
public synchronized void setLangProjection(String lang);

/**
 * Nastaví "jazyk" pro OCR extremálních oblastí (typicky "latin").
 * Musí existovat soubor dir/tessdata/lang.traineddata
 * @param lang
 */
public synchronized void setLangMser(String lang);
}
```

4.1 Segmentace

Segmentaci, tedy nalezení řádků s textem v obrázku, obstarává `SegmentationWorker`. Vzhledem k tomu, že pro segmentaci není potřeba zdlouhavá inicializace, je vytvořeno vlákno až při požadavku na zpracování (`AsyncCteni.execute`). Parametrem zpracování je `minSize`, tj. počet pixelů delší strany obrázku při nejmenší zpracovávané velikosti (výchozí 300). Další parametr je `scaleStep`, kterým jsou násobeny rozměry obrázku z předchozího kroku tak dlouho, dokud není zpracován obrázek v původním rozlišení. Vzhledem k 2D povaze obrázku hodnota 2 znamená, že obrázek v dalším kroku bude mít 4krát více pixelů. Bitmapy jsou zmenšovány pomocí `createScaledBitmap`[1]) s parametrem `filter = true`. Zmenšení bitmapy (načtené z JPG) zároveň omezuje šum a artefakty komprese.

Zpracované řádky předává do `List<TLineWrapper> segmentedLines`, odkud jsou vyzvednuty OCR vlákny.

4.2 Kompenzace rozdílu v rozlišení

Geometrie řádků po segmentaci je počítaná vzhledem k rozměrům zmenšeného obrázku. Proto je v asynchronní implementaci přistupováno k řádkům přes `TLineWrapper`, který se skládá z původního řádku `TLine tl` v prostoru menšího obrázku a `float scale`. Při získávání poloh dotažnic a jejich koeficientů pro výpočet překryvu jsou souřadnice transformovány: $b_1(x) = tl.b_1(x/scale) \cdot scale$, protože se jedná jen o změnu měřítka, nikoli o posun. Úprava koeficientů je pro polynom

$$b_1(x) = c_{b_1} + \sum_{i=1}^{m} k_i x^i, \text{ pro parabolu } m = 2$$
$$c_{b_1} = tl.c_{b_1} \cdot scale$$
$$k_i = tl.k_i \cdot scale^{-(i-1)}$$

[1]) `http://developer.android.com/reference/android/graphics/Bitmap.html#createScaledBitm...`

4.3 OCR

Ve chvíli, kdy se na `AsyncCteni` zavolá `setLangProjection` nebo `setLangMser`, se vytvoří nové vlákno pro OCR daného jazyka, které inicializuje Tesseract OCR, což trvá i několik vteřin. V tuto chvíli ještě nemusí být předaný obrázek. Naopak je žádoucí, aby inicializace probíhala během toho, co uživatel teprve vybírá obrázek ke zpracování.

Po zpracování celého obrázku vlákno zůstává aktivní a je připraveno pro čtení dalšího obrázku, bez nutnosti inicializace Tesseractu. V případě, že není jazyk nastaven s dostatečným předstihem[1]), inicializace probíhá současně se segmentací, což může zdržovat méně-jádrový procesor, a stává se, že je dříve dokončena segmentace, než je OCR inicializováno – tedy se první výsledky o to zpozdí.

OCR vlákno sleduje příbytky v `segmentedLines` a pokud obsahuje (v daném režimu) ještě nezpracované řádky, tak vybere řádek, který má minimum kolizí s již přečtenými a zároveň nejlepší odstup barvy textu od pozadí (3.7.1). To zajistí že se postupně snaží číst všechny oblasti (aby bylo možné presentovat částečný výsledek) a poté se teprve čtou další segmentace již přečtených oblastí.

- Pokud OCR má za úkol zpracovávat rektifikované extremální oblasti, tak je vstupní bitmapa získána jednoduchým převodem bitmapy vytvořené v C++ (3.5.3) a poté se uvolní její paměť.
- V případě projekce z původního obrázku se použije `CteniCppApi.projectPatch`. Je možné promítnout řádek ve vyšším rozlišení, než bylo segmentováno v případě, že byla segmentace provedena na zmenšeném obrázku.

Je možné sledovat aktuálně čtené řádky přetížením metody `AsyncCteni.onReading`:

```
/**
  * Volá OCRWorker před tím, než začne číst volatilePatch; bitmapa je
  * záhy zničena, pro její využití je potřeba vytvořit kopii.
  * @param tlw -- řádka
  * @param isProjected -- true=projekce z obrázku,
  *                       false=extremální oblasti
  * @param volatilePatch
  */
protected void onReading(TLineWrapper tlw, boolean isProjected,
                         Bitmap volatilePatch)
```

4.4 Filtrování kolizí

`LayoutConflictsWorker` je inicializován také až při zahájení zpracování. Ačkoli načítá jazykový model, a to může chvíli trvat, není to tak kritické jako inicializace OCR a je potřeba ho mít načtený až poté, co proběhne první segmentace plus jsou výsledky z OCR. Ty jsou reprezentované třídou `TLineOCR`, která slučuje `TLineWrapper`, přečtený `String` a `confidence` vrácenou Tesseractem.

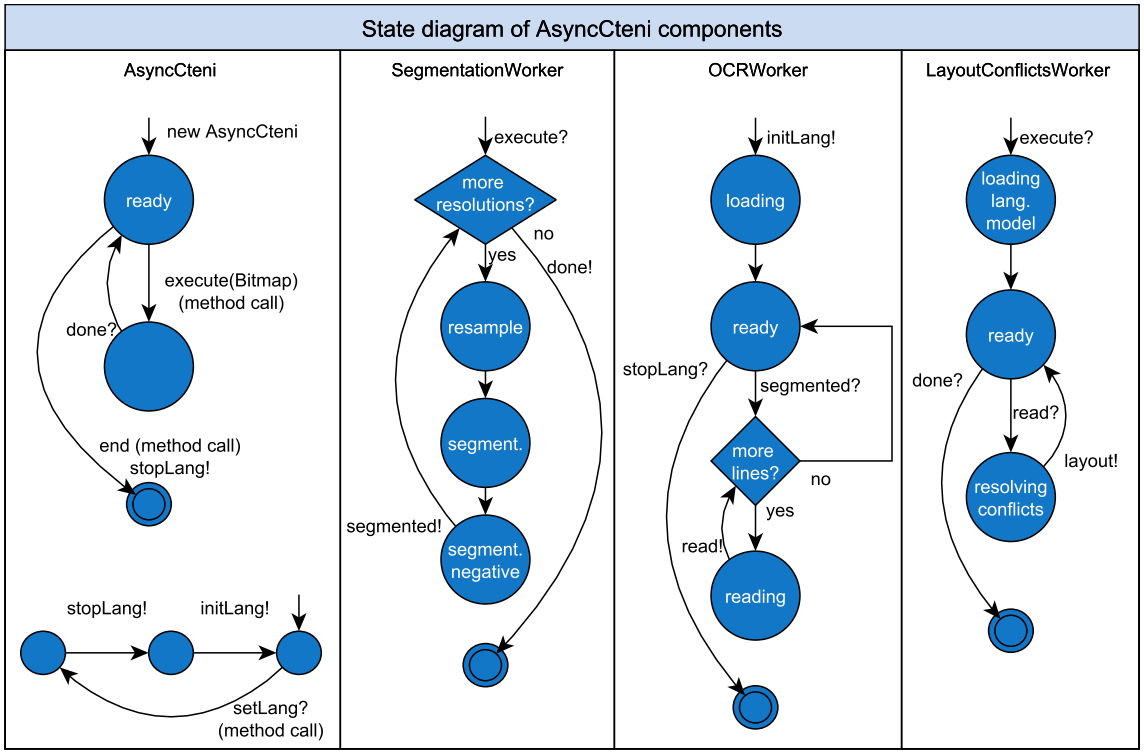
Při nových výsledcích čtení je tedy vyhodnoceno, které jsou relevantní v dané scéně: řádky se nesmějí překrývat a musí mít nějakou realistickou pravděpodobnost. Takto vybrané řádky jsou předány metodě `layout` popsané výše, kde je na uživateli této komponenty, aby vyřešil zobrazení výsledků v GUI – má k disposici řádky včetně jejich poloh v původním obrázku.

[1]) Stává se, že Android při aktivitě výběru obrázku (fotoparát/galerie) ukončí naši aplikaci, takže potom nezbývá než inicializovat OCR během segmentace.

Součástí výstupu je `int confidentCoverageCounter`, což je přirozené číslo vyjadřující kolikrát již byla přečtena celá scéna. Dokud je to nula, pak nebyly ještě čteny všechny oblasti, kde byly nalezeny řádky.

Samotné řešení kolizí je rychlá operace; uživatelem implementovaná metoda `layout` běží v témže vlákně. Pokud je libo pracovat s GUI, je potřeba se přepnout do UI vlákna (např. pomocí `Activity.runOnUiThread`[1]).

4.5 Synchronizace



Obrázek 4.4. Orientační přehled komponent `AsyncCteni`, s vypůjčenými vlastnostmi časových automatů (*timed automaton*, především synchronizace – akce! → akce?)

Vzhledem k tomu, že `java.util.concurrent.Phaser`[2]) je až v API level 21 (tj. Android 5.0, resp. Java 7), tak jsem většinu synchronizace řešil přes `CountDownLatch`[3]) (API level 1). Probouzení OCR potřebuji provádět opakovaně, vyřešil jsem to přes `AtomicReference<CountDownLatch>`, kam v případě potřeby vložím připravenou laťku, kterou segmentace pro všechna OCR sníží po přidání nových řádek.

- Příznak `boolean AsyncCteni.end` si průběžně sledují všechna vlákna a popřípadě okamžitě ukončí činnost. Po zavolání metody `end()` je příznak nastaven a jsou probuzena všechna OCR, která se záhy ukončí. Po jejím zavolání již není možné instanci `AsyncCteni` dále používat. Pokud OCR v daný okamžik zrovna něco zpracovává, ukončování čeká. Z paměti jsou uvolněny všechny nepřečtené rektifikované řádky.
- Při volání `execute(Bitmap)` jsou spuštěna vlákna pro segmentaci a kolize, a čekají na `startSignal = new CountDownLatch(1)`, který je již proveden asynchronně, tedy vlákno volající `execute` je záhy uvolněno.

[1]) `http://developer.android.com/reference/android/app/Activity.html#runOnUiThread(Runn...)`
[2]) `http://developer.android.com/reference/java/util/concurrent/Phaser.html`
[3]) `http://developer.android.com/reference/java/util/concurrent/CountDownLatch.html`

- `SegmentationWorker` poté provádí segmentace pro různá rozlišení. Po dokončení každého rozlišení informuje potenciálně spící OCR pomocí

```
for (AtomicReference<CountDownLatch> latchRef : ocrLatches)
 latchRef.get().countDown();
```

- Vlákna `OCRWorker` jsou puštěna nezávisle na zpracování obrázku, inicializují Tesseract a potom čekají na signál (viz výše). Pokud OCR již nemá co zpracovávat, nastaví referenci na novou laťku a čeká na signál (ten může přijít po dokončení segmentace vyššího rozlišení, nebo až při zpracování dalšího obrázku).
- `LayoutConflictsWorker` je aktivován prostým `Object.wait/notify` nad kolekcí `ocrOutput`[1]).

Celá tato implementace je realizována v přiloženém projektu `AndroidSegmentace`. Třída `AsyncCteni` pak vyžaduje závislost na projektu `tess-two`[2]), který obsahuje Tesseract OCR pro Android.

[1]) `ConcurrentLinkedQueue<TLineOCR> ocrOutput`
[2]) `https://github.com/rmtheis/tess-two`

Kapitola 5
Aplikace pro Android

Popsanou knihovnu na přečtení textu z obrázku jsem použil pro jednoduchou aplikaci. Cílem bylo udělat ukázku využitelnosti zvoleného postupu. Aplikace má jednoduché GUI, ve kterém uživatel zvolí jazyk, obrázek s textem a uvidí výsledek.

5.1 Jazykové balíčky

Aplikace má uložené jazykové balíčky na paměťové kartě ve složce `cteni`:

```
dataDir = new File(Environment.getExternalStorageDirectory(), "cteni");
```

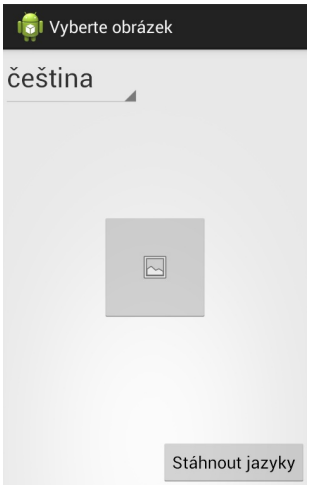
Ta obsahuje jazykové modely (soubory s příponou `.lng4`) a podsložku `tessdata`, kde jsou s odpovídajícími jmény soubory Tesseractu (s příponou `.traineddata`). Jazyky jsou pojmenovány podle ISO 639-3, tj. třípísmenné kódy. O jejich reprezentaci se v aplikaci stará třída `LanguageBundle`, která obsahuje mapování z kódů ISO 639-3 na lokalizované řetězce.

```
# strings.xml
...
   <string name="lang_eng">English</string>
   <string name="lang_ces">Czech</string>
   <string name="lang_rus">Russian</string>
...

# LanguageBundle.java
...
   static{
     idMap.put("eng",R.string.lang_eng);
     idMap.put("ces",R.string.lang_ces);
     idMap.put("rus",R.string.lang_rus);
   }
...
   public String from(Resources r){
     if(idMap.containsKey(ISO_639_3))
            try{
                return r.getString(idMap.get(ISO_639_3));
            }catch(NotFoundException f){
                Log.w(CteniActivity.TAG,
                      "Missing language resource for "+ISO_639_3);
            }
      return ISO_639_3; // return code when localisation not found
   }
...
```

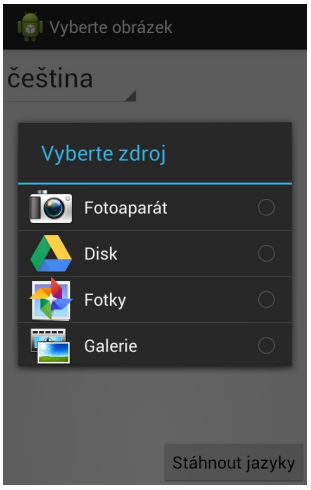
5.2 Vstupní obrazovka

Po spuštění aplikace vidí uživatel obrazovku, na které je Spinner pro výběr jazyka, velké tlačítko pro výběr obrázku, a tlačítko pro stažení dodatečných jazykových balíčků.



Obrázek 5.1. Vstupní obrazovka

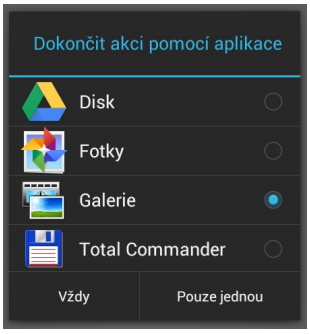
Obrázek 5.2. Volba jazyka čtení.



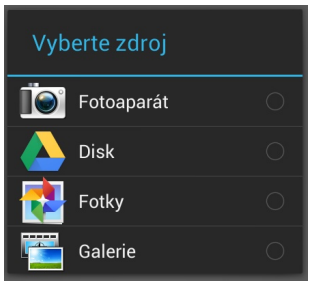
Obrázek 5.3. Výběr obrázku. Po dokončení aktivity výběru se přejde na zobrazení výsledků.

Při výběru jazyka se instanci `AsyncCteni` tato informace předá a okamžitě se začíná inicializovat Tesseract, což může trvat pár vteřin (to se děje v samostatném vlákně, takže uživatel může dále pracovat).

Tlačítko pro výběr obrázku vytváří složený `Intent`[1]) pro vyfocení obrázku (`ACTION_IMAGE_CAPTURE`) a načtení obrázku z telefonu (`ACTION_GET_CONTENT` s typem `image/*`). To má mj. tu vlastnost, že nelze pro příště zapamatovat vybranou akci, což je podle mě žádoucí, protože to má smysl spíše pro akce data *odesílající*, než *přijímající*.



Obrázek 5.4. Klasický `Intent` pro výběr obrázku – souboru.



Obrázek 5.5. Složený `Intent` použitý v aplikaci: možnost vybrat fotoaparát i načíst soubor, bez zapamatování volby.

5.3 Výsledek

Zpracování obrázku a zobrazení výsledku (vzhledem k asynchronnímu průběžnému zpracování) probíhá ve stejné aktivitě jako výběr jazyka a obrázku. Je to z toho důvodu,

[1]) `http://developer.android.com/reference/android/content/Intent.html#createChooser(...)`

aby již mohlo být `AsyncCteni` inicializované vybraným jazykem a takto je to jednodušší, než nějak předávat instanci mezi aktivitami. Uživatele se tato implementační drobnost vůbec nedotýká.

Základní koncepci obrazovky jsem zvolil tak, že uprostřed je zobrazený vstupní obrázek, přes který je vrstva indikující jeho zpracování. Text přečtených řádek se zobrazuje v tmavém pruhu vespod, který postupně může i překrývat obrázek. Pokud by bylo přečteno velké množství textu, může se stát, že bude přes celou obrazovku a uživatel jím může posouvat.

Dále jsem do dolního pravého rohu umístil tlačítko pro „sdílení“ načteného textu (Intent `ACTION_SEND` pro typ `text/plain`), například zkopírování do schránky, do zprávy nebo do překladače.

Pokud má uživatel v telefonu převod textu na řeč (TTS[1]) pro vybraný jazyk, tak zobrazím i tlačítko pro toto.

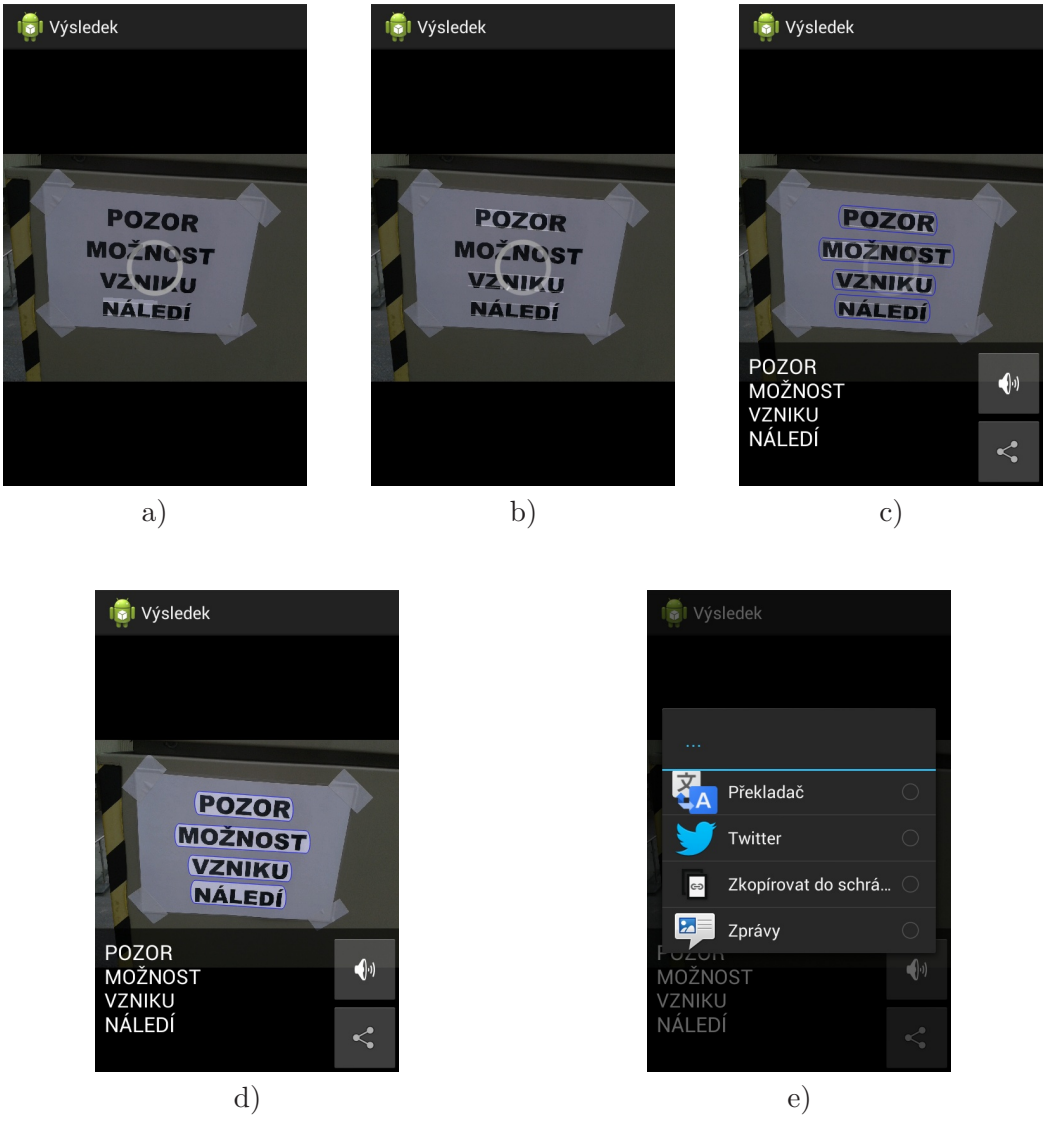
```
TextToSpeech tts = new TextToSpeech(...); // zjištění dostupnosti TTS
Locale tts_locale = new Locale(selectedLanguage.ISO_639_3);
int tss_avail = tts.isLanguageAvailable(tts_locale);
if(tss_avail != TextToSpeech.LANG_MISSING_DATA &&
   tss_avail != TextToSpeech.LANG_NOT_SUPPORTED){
   tts.selectedLanguage(tts_locale); // TTS dostupné
}
// přečtení textu nahlas
tts.speak(readString, TextToSpeech.QUEUE_FLUSH, null);
```

Díky asynchronnímu zpracování se přečtený text načítá postupně a také jsou v průběhu k disposici informace o zpracovávaných oblastech. Typicky probíhají fáze:

- čekání na dokončení segmentace (to trvá jen okamžik, protože se začíná nízkým rozlišením)
- postupné předávání řádek OCR (to v aplikaci sleduji pomocí přetížení `onReading()`), na základě geometrické informace o řádce zvýrazním danou oblast vstupního obrázku)
- získané výsledky OCR; pokud se zvýší `confidentCoverageCounter`[2]), tak zobrazím text a zvýrazním příslušné řádky v obrázku

Pro indikaci činnosti jsem na střed obrázku vložil `ProgressBar` (točící), kterému při prvních částečných výsledcích zvýším průhlednost a po dokončení čtení jej skryji.

[1]) `http://developer.android.com/reference/android/speech/tts/TextToSpeech.html`
[2]) Kapitola 4.4, zvýšení indikuje přečtenost celé scény

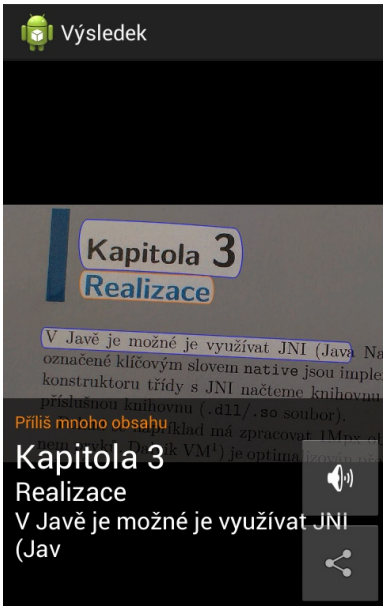


Obrázek 5.6. Zpracovávání: a,b) zvýraznění zpracovávaných řádek, c) částečné výsledky OCR, d) konečný výsledek e) sdílení textu

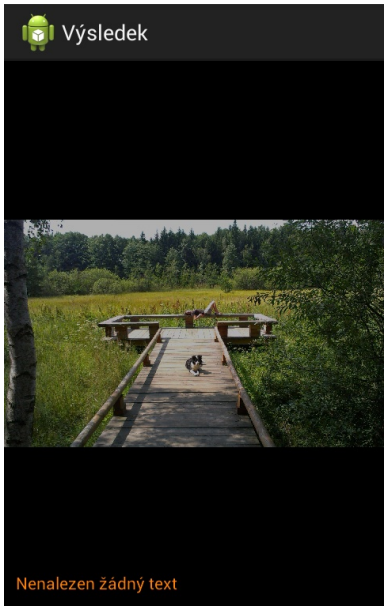
Pokud zpracování trvá příliš dlouho (sestavování tripletů, prototypů ...), tak se předčasně ukončí, aby alespoň nějaké výsledky byly uživateli předány a v UI se zobrazí oranžově hláška *Příliš mnoho obsahu*. Maximální časy pro jednotlivé fáze se definují ve `cteni.h`. V této práci jsem je experimentálně stanovil:

```
limitTLinesTriplTime = 200;      // čas na sestavování trojic (ms)
limitTLinesProtoCnt = 25;        // počet prototypů
limitTLinesSplitPerProtoCnt = 5; // počet řádek vzniklých z 1 prototypu
limitPrecistTime = 600;         // čas rektifikace řádek (všech dohromady)
```

Poté, co se v žádném zpracovávaném rozlišení nepodaří přečíst text, tak je zobrazena hláška *Nenalezen žádný text*.



Obrázek 5.7. Překročeny limity pro segmentaci: zpracování předčasně ukončeno.



Obrázek 5.8. Nenalezen žádný text.

5.3.1 Průběžné zobrazování

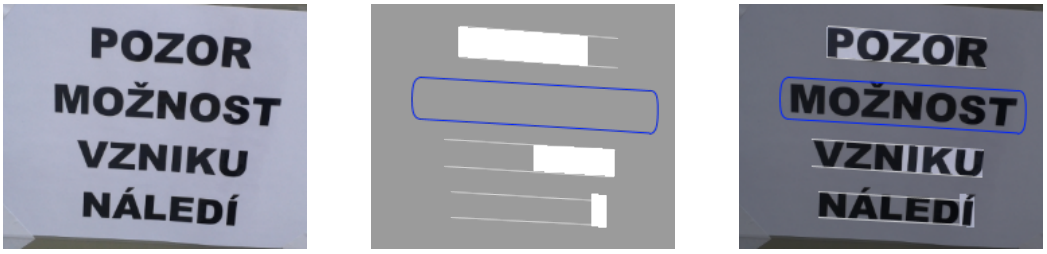
Aby uživatel viděl, že *se něco děje* a čekání mu lépe ubíhalo, tak před spuštěním OCR pro každý řádek se spustí v UI animace oblasti daného řádku: zvýrazní se horní a dolní dotažnice (t_2 a b_1) linkou bílé barvy a pro celou oblast řádku (mezi t_1 a b_2) se postupně zprůhlední překryvná vrstva tak, že levý okraj je v x_{min} a pravý okraj se rychlostí 100 px za vteřinu dotahuje na x_{max}, až je celý x-rozsah řádku takto zvýrazněný. Pak se levý okraj stahuje doprava stejnou rychlostí, až dosáhne hodnoty x_{max} a průhledná část zmizí, a bíle vykreslené linky jsou také skryty.

Řádky, jejichž text byl vybrán jako výstup OCR mají svou oblast obtaženou barevnou linkou, a pokud je již čtení dokončeno, tak jsou vidět skrz překryvnou vrstvu výrazně. Jejich oblast je o 30% nafouknuta, aby obrys nebyl nalepen na písmena.

Celé je to udělané pomocí `SurfaceView`[1]), které má tu vlastnost, že mu lze nastavit obsah pomocí `Canvasu`[2]). To se hodí, protože jiné *kreslící vrstvy* očekávají vykreslení každého snímku, což v tomto případě není úplně rychlá operace (vykreslení obrázku a přes něj poloprůhlednou vrstvu, obzvlášť když tou dobou intenzivně pracují další čtyři vlákna na čtení obrázku), takže by UI vlákno nebylo schopno svižně reagovat na uživatelské operace. Takto si v dalším vlákně počítám snímky animace rychlostí 15 FPS (nebo nižší, záleží jak telefon stíhá) a hotový snímek předám `SurfaceView`.

Ve skutečnosti mám `SurfaceView` dvě, v jednom je nakreslený původní obrázek (plátno je před nakreslením zmenšeno a posunuto tak, aby obrázek byl celý vidět) a překreslí se jen v případě změny velikosti obrazovky (tj. otočení). Nad ním je druhé `SurfaceView`, jehož plátno je posunuto stejně jako pro obrázek, ale je vyplněno černou barvou s průhledností 60 %. Podle stavu animace/zobrazení výsledku jsou vyplněny některé oblasti průhlednou barvou, což způsobí lepší viditelnost podloženého obrázku, tedy jakoby „zvýraznění“.

[1]) `http://developer.android.com/reference/android/view/SurfaceView.html`
[2]) `http://developer.android.com/reference/android/view/SurfaceHolder.html#unlockCanvas...`



a) b) c)

Obrázek 5.9. Ukázka (výřez) obsahu obou `SurfaceView`: a) spodní vrstva (původní obrázek), b) vrchní vsrtva s animací[1]). Zvýraznění pro `POZOR` ve fázi *dorůstání*, `VZNIKU` a `NÁLEDÍ` v 2. fázi (*mizení*) a `MOŽNOST` v modrém rámečku, protože již bylo přečteno a je tedy ve spodním pruhu GUI s výsledky[2]), c) vrstvy (a) a (b) zobrazené přes sebe – toto vidí uživatel.

5.4 Stahování jazyků

Uživatel má po spuštění volbu stažení dalších jazykových balíčků (z internetu). Pokud aplikace nemá žádné (první spuštění), tak je automaticky spuštěna tato aktivita.

Zdroj stahování je určen adresou v řetězci `download_baseurl` (dále budu psát `URL`). Nejprve se z adresy `URL/_.json` stáhne informace o dostupných jazycích ve formátu JSON, jehož schéma[3]) je:

```
{
  "description": "schema souboru _.json",
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "lang": {
        "type": "string",
        "description": "ISO 639-3 kod, ke stazeni jako lang.zip"
      },
      "crc32": {
        "type":"string",
        "description": "HEX(crc32) souboru lang.zip/lang.lng4"
      }
    },
    "required": ["lang", "crc32"]
  }
}
```

a porovnají se CRC32 otisky balíčků (resp. pouze jazykových modelů[4]), uložených ve složce `dataDir`) s hodnotami `crc32` pro jazyky dle `_.json`.

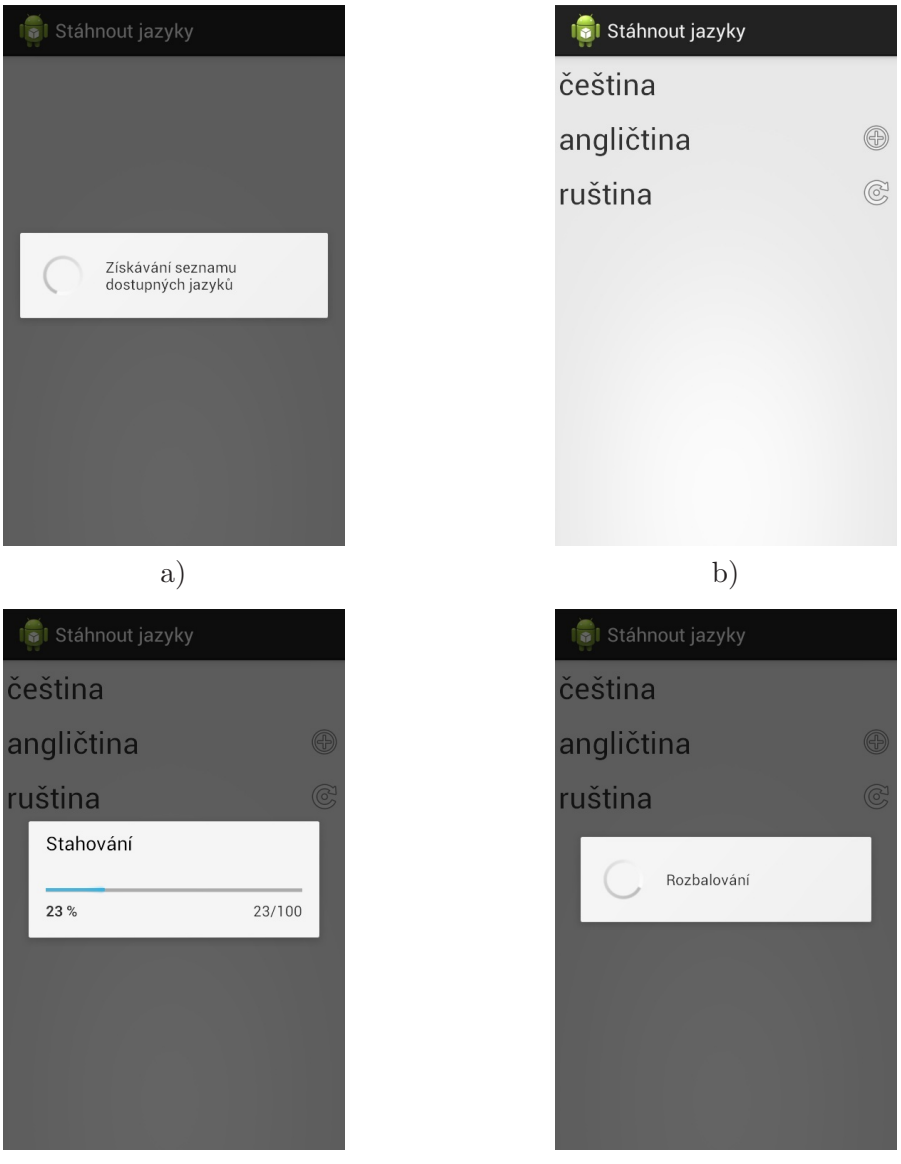
[1]) Poloprůhledný obrázek – bílé oblasti jsou ve skutečnosti 100% průhledné, a *šedá* plocha je černá barva z 60 % průhledná.
[2]) Pouze ilustrativní příklad, pro tento obrázek by se ve skutečnosti jako přečtené vyhodnotily všechny řádky společně – až po zvýšení `confidentCoverageCounter`
[3]) `http://json-schema.org`
[4]) Není potřeba kryptograficky bezpečný součet. Pokud by model zůstal a měl se aktualizovat jen soubor pro Tesseract, není problém změnit triviálně i model, protože jeho poslední položkou je datum (viz. kapitola 3.8), které se nikde neuplatňuje, ale CRC souboru změní.

Jazyky v telefonu a jazyky ke stažení jsou sloučeny do zobrazeného seznamu. U těch, které nejsou dosud stažené, nebo nemají shodný CRC32, je nabídnuta volba stáhnout nový, respektive stáhnout aktualizovaný balíček.

Do telefonu se stáhne (do složky `dataDir`) soubor z `URL/xyz.zip` (kde `xyz` je ISO 639-3 kód jazyka dle `_.json`) a rozbalí se. Je tedy vyžadováno, aby struktura souboru byla:

```
xyz.zip
|-- xyz.lng4
\-- tessdata
    \-- xyz.traineddata
```

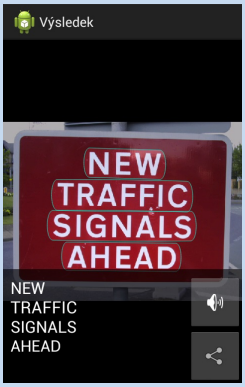
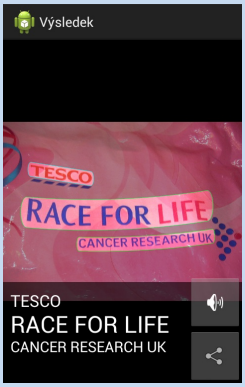
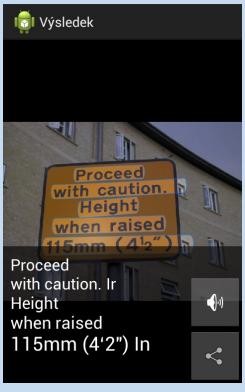
Poté se `xyz.zip` smaže a jazyk `xyz` je v tuto chvíli možno použít pro čtení. Stažený jazyk je při návratu na vstupní obrazovku vybrán jako aktivní.

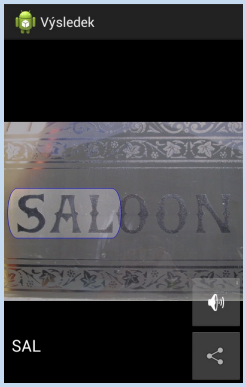
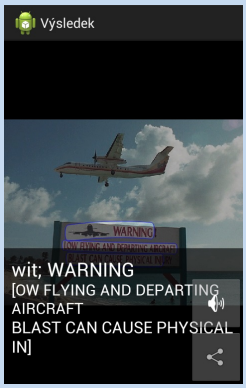
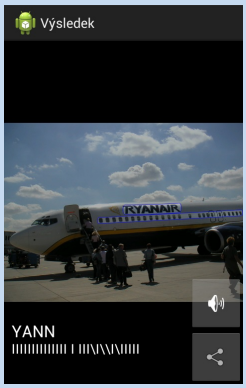


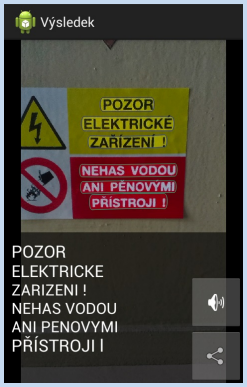
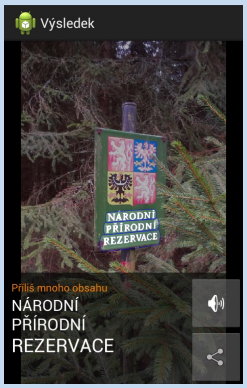
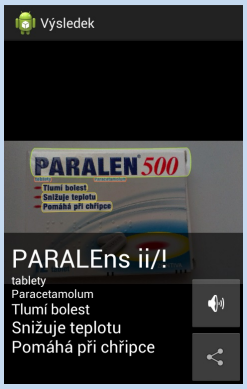
Obrázek 5.10. Stahování balíčků: a) získávání `_.json`, b) seznam jazyků. Čeština je v telefonu, angličtina není, a ruštinu je možno aktualizovat. c) Po výběru jazyka se stahuje ZIP soubor, d) rozbalování balíčku.

Kapitola 6
Výsledky

V této kapitole popíšu chování aplikace na sadě ICDAR 2003[9] (anglické texty) a na vlastních fotografiích pořízených telefonem. Obrázky z ICDAR 2003 mají rozlišení kolem 1 Mpx a výsledek je velmi rychle dostupný (do deseti vteřin, na telefonu HTC Desire X). Oproti tomu z telefonu jsou 4Mpx a nejsou tak ostré, takže zpracování trvá déle (desítky vteřin).

v aplikaci	vstup	přečtený text chyby/chybí	poznámka
		NEW TRAFFIC SIGNALS AHEAD	Bez problému; ostrá fotka.
		TESCO RACE FOR LIFE CANCER RESEARCH UK	Prohnuté řádky; je potřeba pochválit OCR, protože ani po narovnání řádek nejsou písmenka zcela rovně.
		Proceed with caution. lr Height when raised 115mm (4'2") ln	Struktura oken zahrnuta do řádků. Písmeno v horním indexu chybně přečteno.

v aplikaci	vstup	přečtený text chyby/chybí	poznámka
		SALOON	Příliš nízký kontrast písmen, málo stabilní extremální oblasti ignorovány.
		wit; WARNING [OW FLYING AND DEPARTING AIRCRAFT BLAST CAN CAUSE PHYSICAL IN]URY	Čtení piktogramu. Problém OCR (písmena L, J). Problém segmentace (poslední řádek nenalezen celý).
		RYANNR |||||||||||||...	Netypický font (OCR považuje AI za N a ignoruje R). Struktura oken považována za řádek.
		ZAKAZ KOURENI	Vynechána diakritika, protože v projekci z původního obrázku není zahrnuta (dotažnice t_1 a t_2 splývají).

v aplikaci	vstup	přečtený text chyby/chybí	poznámka
		POZOR ELEKTRICKE ZARIZENI ! NEHAS VODOU ANI PENOVYMI PŘÍSTROJI |	Opět chybí diakritika. Vykřičník přečten jako |.
		NÁRODNÍ PŘÍRODNÍ REZERVACE	Ve větvičkách nalezeno mnoho pozdezřelých oblastí (*Příliš mnoho obsahu*), ale text rozpoznán správně.
		PARALEns ii/! tablety Paracetamolum Tlumí bolest Snižuje teplotu Pomáhá při chřipce 12 tablet ZENTIVA	Problém se změnou fontu v názvu (kurzíva). Nenalezené řádky. *(vstup otočen kvůli sazbě)*

Tabulka 6.1. Ukázky výsledků čtení, popis problémů v poznámkách. Pro prohlížení obrázků doporučuji zvětšené PDF.

V použité metodě lokalizace textu[2–3] je udávána 67% úspěšnost[1]) na sadě ICDAR 2003. Oproti tomu na sadě Street View Text Dataset (z Google Street View) má úspěšnost 33 % (s poznámkou, že je problém s anotacemi v sadě).

V mojí aplikaci pozoruji stejný jev[2]): obrázky, které jsou ostré a oříznuté (tj. text je v nich leitmotiv, například z ICDAR 2003 nebo fotobank), jsou přečteny rychle a spolehlivě. Oproti tomu obrázky získané telefonem typicky obsahují z velké části okolí scény, jsou neostré a zašuměné. Proto jejich zpracování trvá déle a není tak spolehlivé.

[1]) f-measure, je kombinací *precision* (podíl skutečných řádek s textem mezi všemi nalezenými řádky) a *recall* (procento nalezených řádek s textem)

[2]) Měřením konkrétní přesnosti jsem se ovšem nezabýval.

Zlepšení výsledků by pomohlo předzpracování obrázku jako například doostření a odstranění šumu (to už je částečně zajištěno zpracováním v nižším rozlišení). Dále jsem pozoroval, že při focení zachytím mnohem větší okolí a samotný text se nachází někde uprostřed, nepodstatné okraje zabírají i 70 % plochy obrázku. To by šlo zlepšit tím, že by se pro focení nepoužívala systémová aplikace, ale vlastní, která by umožňovala rovnou obraz oříznout.

6.1 Porovnání aplikací

Na několika obrázcích ze sady ICDAR 2003 jsem provedl porovnání existujících aplikací. Načíst vstup z obrázku umí pouze navržená aplikace a Google Translate. Pro WordLens a Bing Translator jsem obrázek snímal z monitoru a snažil jsem se o co nejlepší výstup. Všechny aplikace byly nastaveny na čtení anglického jazyka.

			
	BEWARE ALLIGATOR TEETH	WARNING LOW FLYING AND DEPARTING AIRCRAFT BLAST CAN CAUSE PHYSICAL INJURY	Customer Freephone Private Hire Service
navržená aplikace	BEWA’’_9 ALLIG’KTOR TEETH	wit; WARNING [OW FLYING AND DEPARTING AIRCRAFT BLAST CAN CAUSE PHYSICAL IN]URY	Customer Freephone Private Hire Service
WordLens	JO MI BEWARE ALLIGATOR TEETH	WARNING LOW FLYING AND DEPARTING AIRCRAFT BLAST CAN CAUSE PHYSICAL LAJURY	CUSTOMER FREEMAN PRIVATE HIRE SERVICE
Bing Translator	BEWARE ALLIdÅTOR TEETH	.4 WARNING LOW FLYING AND DEPARTM AIRCRAFT BUST CAN CAUSE PHYSICAI INJURY	Customer Freephone Private Hire Service
Google Translate	BEWARE ALLIGATOR TEETH	WARNING LOW FLYING AND DEPARTING AIRCRAFT BLAST CAN CAUSE PHYSICAL INJURY	Customer Freephone Private Hire Service

			
	RIVERSIDE WALK	INVESTMENTS PENSIONS MORTGAGES	TESCO VALUE Washing up liquid
navržená aplikace	[I VERSIDE WALK	INVESIMENTS PENSIONS MORTGAGES	illllll TESCO VALuE Washing UI "quid
WordLens	ROVERRIDE WALK	INVESTMENTS PENSIONS MORTGAGES	ILLICIT TESCO VALUE Washing up liquid
Bing Translator	RIVERSIDE A WALK 1	INVESTMENTS PENSIONS* 'MORTGAGES*	TESCO VALUE Washing up liquid
Google Translate	RIVERSIDE WALK	INVESTMENTS PENSIONS. MORTGAGES	TESCO VALUE Washing up liquid

Tabulka 6.2. Porovnání výstupů existujících aplikací. Červeně jsou nalezená, leč špatně přečtená písmena (popřípadě nalezená písmena tam, kde žádná nejsou). Červeně podbarvená písmena nebyla aplikací vůbec lokalizována.

Ze srovnání vyplývá náročnost úlohy čtení textu z obrázků, protože žádná aplikace nemá na vybraných obrázcích 100% úspěšnost. Zároveň není obrázku, který by všechny aplikace přečetly spohehlivě, a přitom se nejedná o nějaké divoké scény.

- **Navržená aplikace** má problémy při zpracování špatně segmentovaného textu, a také pracuje v porovnání s ostatními aplikacemi nejpomaleji (zpracovává obrázek cca 10 vteřin). Oproti tomu je poměrně úspěšná v lokalizaci textu.
- **WordLens** je uzpůsobena na čtení textu cedulí a zde to potvrzuje. Při zpracování neznámého slova nebo hůře kontrastního textu, má problémy a prezentuje nesouvisející výsledky.
- **Bing Translator** má nejuspokojivější výsledky z offline aplikací. Příjemné je to, že je zobrazuje v reálném čase přímo v náhledu kamery.
- **Google Translate** má nejlepší výsledky, ovšem vzhledem ke zpracování obrázku na serveru není na stejné startovní čáře.

Kapitola 7
Závěr

V této práci jsem vytvořil aplikaci pro OS Android, která v obrázku reálné scény nalezne text a přečte jej, bez potřeby připojení k internetu. Nejprve jsem musel vyřešit lokalizaci řádek textu: zvolil jsem metodu podle Neumann a Matas[3]. Tu jsem naprogramoval v C++, protože je v telefonu řádově rychlejší než Java a lze lépe pracovat s pamětí. Pro zpracování 1 Mpx potřebuje 40 MB paměti.

Pro rozpoznání nalezených řádek s textem jsem použil open source Tesseract OCR. Vzhledem k časové náročnosti rozpoznávání textu jsem navrhl a implementoval asynchronní proces, který obrázek zpracovává nejprve v nižším rozlišení a heuristicky volí, které řádky se budou číst OCR. Takto jsou během několika vteřin k disposici první výsledky, jež v případě zřetelné scény jsou už finálním výstupem. Celý proces typicky trvá desítky vteřin, a to v závislosti na počtu nalezených řádků. Tento proces jsem vyčlenil do samostatné knihovny, kterou lze dále snadno použít v jiných projektech.

Dále jsem tuto knihovnu použil v samotné uživatelské aplikaci, jež má jednoduché rozhraní: uživatel zvolí jazyk čteného textu, vybere nebo vyfotí obrázek a tento je zpracován. Proces zpracování je průběžně vizualizován animací nalezených textových řádek a nalezený text je možné nechat nahlas přečíst[1]), nebo předat jiným aplikacím, jako například překladač nebo zprávy. Aplikace také řeší stahování jazykových balíčků z internetu.

Aplikace má dobré výsledky na obrázcích, které jsou ostré, kontrastní a text je dostatečně velký. Takové jsou například některé obrázky ze sady ICDAR 2003, obrázky cedulí z fotobank, nebo i obrázky vyfocené mnou, pokud jsem se soustředil, aby byly dostatečně kvalitní. Pokud ovšem vyfotím text ne-úplně-zblízka, tedy je poměrně malý, často i rozmazaný vlivem nekvalitní optiky, tak takový text není úspěšně rozpoznán, nebo je rozpoznán až po dlouhé době. Tato vlastnost vyplývá i z výsledků úspěšnosti použité metody lokalizace na datové sadě obrázků z Google Street View oproti ICDAR 2003.

Vzhledem k tomu jsem se rozhodl upustit od původního záměru, totiž že bych v rámci této práce publikoval hotovou aplikaci veřejně.

Náročnost řešené úlohy potvrzuje také malé množství aplikací třetích stran, které převádí obrázek na text. Jejich srovnání v kapitole 6.1 ukázalo, že na vybraných obrázcích ze sady ICDAR 2003 nemá žádné offline řešení výrazně lepší úspěšnost.

7.1 Práce do budoucna

Při implementaci jsem narazil na mnoho problémů, jež jsem víceméně úspěšně vyřešil. Vyplynulo mi ovšem několik zajímavých nápadů, které jsem zatím nerealizoval:

- Pro zrychlení procesu bych chtěl použít jinou metodu rozpoznání (než externí OCR), protože nalezené řádky už mám segmentované na jednotlivá písmena. Zde vy vlastní řešení mohlo být výrazně rychlejší[2]).

[1]) pokud má uživatel v telefonu příslušný jazyk v modulu TTS – převod textu na řeč

[2]) Ostatně takto je to popsáno i v původní metodě, ze které jsem se ovšem rozhodl použít pouze část pro lokalizaci, protože dle mé zkušenosti by takový klasifikátor vyžadoval příliš mnoho paměti.

- Předzpracování obrázku, jako je digitální doostření a odstranění šumu. Dále identifikace okrajových částí obrázku, kde se text nenachází – což by mohlo vést ke zmenšení vstupu popsané metody, respektive ke zpracování ve vyšším rozlišení.
- Při samotném získávání obrázku umožnit uživateli omezit se při hledání textu jen na nějakou oblast, například zvětšení u fotoaparátu nebo ořez obrázku.
- Objemově efektivnější reprezentace jazykového modelu. Nyní je to markovův řetězec 4. řádu a pro uložení potřebuje řádově n^4 bajtů, kde n je velikost abecedy. Pro jazyky používající latinku je to 2,5 MB.

Literatura

[1] Alessandro Bissacco, Mark Cummins, Yuval Netzer a Hartmut Neven. *PhotoOCR: Reading Text in Uncontrolled Conditions.* In: *The IEEE International Conference on Computer Vision (ICCV)*. 2013.

[2] Lukáš Neumann a Jiří Matas. *Text Localization in Real-World Images Using Efficiently Pruned Exhaustive Search.* In: *Document Analysis and Recognition (ICDAR), 2011 International Conference on*. IEEE Computer Society Offices, 2001 L Street N.W., Suite 700 Washington, DC 20036-4928, United States: IEEE Computer Society Conference Publishing Services, 2011. 687–691. ISBN 1520-5363.

[3] Lukáš Neumann a Jiří Matas. *Real-time scene text localization and recognition.* In: *2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, June 16-21, 2012*. 2012. 3538–3545.
`http://dx.doi.org/10.1109/CVPR.2012.6248097`.

[4] Google. *Tesseract OCR*.
`https://code.google.com/p/tesseract-ocr/`.

[5] Jiri Matas, Ondrej Chum, Martin Urban a Tomás Pajdla. Robust wide-baseline stereo from maximally stable extremal regions. *Image and vision computing*. 2004, 22 (10), 761–767.

[6] Alexandru Niculescu-mizil a Rich Caruana. *Obtaining Calibrated Probabilities from Boosting.* In: *In: Proc. 21st Conference on Uncertainty in Artificial Intelligence (UAI '05), AUAI Press*. AUAI Press, 2005.

[7] Martin A. Fischler a Robert C. Bolles. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Commun. ACM*. 1981, 24 (6), 381–395. DOI 10.1145/358669.358692.

[8] A. Giloni a M. Padberg. Least Trimmed Squares Regression, Least Median Squares Regression, and Mathematical Programming. *Math. Comput. Model.*. 2002, 35 (9-10), 1043–1060. DOI 10.1016/S0895-7177(02)00069-9.

[9] Lucas Panaretos Sosa, S. M. Lucas, A. Panaretos, L. Sosa, A. Tang, S. Wong a R. Young. *ICDAR 2003 Robust Reading Competitions.* In: *In Proceedings of the Seventh International Conference on Document Analysis and Recognition*. IEEE Press, 2003. 682–687.

[10] Leonid Batyuk, Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Ahmet Camtepe a Sahin Albayrak. *Developing and Benchmarking Native Linux Applications on Android*. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. 2009.
`http://dx.doi.org/10.1007/978-3-642-01802-2_28`.

Příloha A
Obsah přiloženého DVD

Přiložené DVD obsahuje složky:

- **text** – Text této práce (PDF, TEX, ilustrace)
- **workspace** – Eclipse workspace s projekty
  - **AndroidSegmentace** – Knihovna pro zpracování obrázku (podle kapitoly 3) a asynchronní implementace (podle kapitoly 4). Má závislost na `tess-two`[1])
  - **AndroidCteni** – Grafická aplikace z kapitoly 5
- **dist** – Výsledné APK
  - **lang** – Jazykové balíčky (čeština, angličtina, ruština)

[1]) `https://github.com/rmtheis/tess-two`