VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

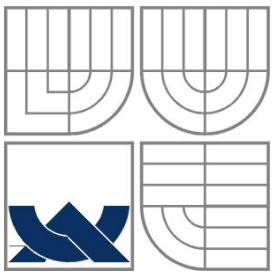
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ČTEČKA BRAILLOVA PÍSMA

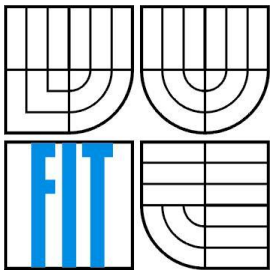
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE *Bc. PAVEL ZAPLETAL*
AUTHOR

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ČTEČKA BRAILLOVA PÍSMA

BRAILLE READER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — *Bc. PAVEL ZAPLETAL*
AUTHOR

VEDOUCÍ PRÁCE — *Ing. LUKÁŠ MARŠÍK*
SUPERVISOR

BRNO 2012

Zadání:

1. Seznamte se se základy zpracování obrazu a knihovnou OpenCV.
2. Prostudujte teorii Braillova písma a vytipujte vhodný postup pro automatické rozpoznání.
3. Navrhněte aplikaci umožňující rozpoznání Braillova písma ve fotografii a jeho převod na text.
4. Implementujte navrženou aplikaci.
5. Proveďte sérii vhodných testů pro vyhodnocení výkonnosti a spolehlivosti použitého řešení.
6. Diskutujte budoucí rozšíření a modifikaci programu.

Abstrakt

Práce se zabývá problematikou převodu Braillova písma zachyceného na fotografii do podoby běžného textu. Věnuje se specifikaci Braillova písma a jejím principům. Stručně jsou také popsány způsoby snímání obrazu. Rozebírá jednotlivé fáze zpracování obrázku a jejich implementaci v C++. V závěru jsou uvedeny možnosti budoucího pokračování projektu.

Abstract

The aim of work is conversion of a photographed Braille into ordinary text. On the beginning, there are described Braille specification and principles. In the next part are mentioned types of sensors for taking photos. The main part describes phases of image processing and their implementation in C++. At the end, there are listed possibilities of future development.

Klíčová slova

Braillovo písmo, CCD, CMOS, prahování, detekce bodů, C++, OpenCV, Qt

Keywords

Braille, CCD, CMOS, thresholding, detection of points, C++, OpenCV, Qt

Citace

Zapletal Pavel: Čtečka Braillova písma, diplomová práce, Brno, FIT VUT v Brně, 2012

Čtečka Braillova písma

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Lukáše Maršíka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Pavel Zapletal
16. května 2012

Poděkování

Děkuji vedoucímu práce za přínosné konzultace a pomoc v průběhu akademického roku.

Obsah

1 Úvod

Dnešní doba je charakteristická velkou poptávkou po informacích. Naštěstí díky rozvoji internetu je přístup k nim o dost ulehčen. Nicméně jako hlavní nositel informací byly po staletí knihy a další tištěné dokumenty a do jisté míry to platí i teď. Pro běžného člověka není problém zajít do knihovny nebo obchodu a opatřit si titul pojednávající o studovaném tématu. Nevidomí však mají tuhle možnost o něco složitější. Existují sice braillské knihovny, ale nabídka knih ve slepeckém písmu, není ani zdaleka tak bohatá. Nutno dodat, že situace se postupně zlepšuje. Překlad nových knih do Braillova písma neustále probíhá, je to však velice pozvolný proces. Na druhou stranu existuje i spousta tiskovin, které vychází v Braillově písmu a překládají se do latinky. Jednak kvůli vydání knižní verze anebo z důvodu uchovat materiál v elektronické podobě. Veškerý text je samozřejmé možné přepsat ručně znak po znaku a také se to tímto způsobem provádí. Nicméně pro převod běžného písma již dnes existují automatizované nástroje, které text z papírové podoby převedou s minimálním úsilím člověka. Proč tedy nevyužít podobný princip zpracování i k převodu Braillova písma. A právě to je motivace pro tvorbu tohoto projektu. Vytvořit aplikaci, která na obrázku detekuje Braillovo písmo a následně ho převede na běžný text.

Abychom však byli schopni překladač zkonstruovat, musíme se na začátku seznámit se způsobem, jakým takové zpracování probíhá. Celý průběh vždy zahrnuje několik fází. Nejdříve je nutné získat obraz předlohy obsahující písmo k překladu. Ten získáme buď pomocí fotoaparátu, pokud se jedná o menší množství textu nebo skenováním v případě většího množství textu, např. jednotlivé stránky z knih. Získaný obraz však z pravidla nebývá vhodný k okamžitému použití. Ve většině případů je určitým způsobem zašuměn či obsahuje jiný druh deformace. Proto je třeba před detekcí klíčových rysů v obraze všechny tyto vady v ideálním případě odstranit, nebo alespoň co nejvíce minimalizovat, aby nedocházelo ke špatnému určení sledovaných znaků. Teprve takto zpracovaný vstup jde do hlavní fáze, ve které jsou identifikovány klíčové části obrazu, což v našem případě znamená jednotlivé reliéfní body Braillova písma. Pochopení principů Braillova písma je tedy pro správnou detekci také velice důležité. Musíme vzít v potaz všechny jeho důležité parametry, abychom byli schopni jej na obrázku co nejlépe rozpoznat. Body mohou mít různou velikost, některé mohou být výraznější než jiné, celý text pak různě pootočen a s tím vším by si měla aplikace umět poradit. Celý její vývoj je v této práci rozdělen do několika kapitol, které postupně odkrývají veškerou zmíněnou problematiku.

Nejprve se kapitole 2 blíže seznámíme s Braillovým písmem. Řekneme si něco o jeho historii a vzniku. Podrobněji se budeme zabývat způsobem zápisu jednotlivých znaků a parametry, které musí písmo splňovat, aby bylo čitelné. Tyhle důležité informace nám pomohou ke správné detekci reliéfních bodů. Ukážeme si také nejběžnější metody tisku slepeckých dokumentů od historických až po současně používané. Jelikož budeme pracovat s obrazem, stručně si v kapitole 3 představíme technologie jeho snímání pomocí CMOS a CCD čipů. Probereme konstrukční řešení obou typů a nakonec porovnáme jejich výhody a nevýhody. Hlavní část práce se zabývá zpracováním obrazu, tedy návrhem jednotlivých fází aplikace, aby byla schopna překládat co nejširší spektrum fotografií a potom samozřejmě i její implementací. V kapitole 4 o návrhu si ukážeme, jak se vstupním obrazem pracovat, abychom z něj získali potřebné rysy. Jednotlivé kroky si zde podrobně představíme a jejich funkci ukážeme na konkrétních příkladech. Podle těchto fází pak budeme postupovat při programové realizaci v následující kapitole 5. Zde budeme tvořit z teoretického návrhu zpracování prakticky použitelnou aplikaci. Jednotlivé fáze tak postupně převedeme do zdrojového kódu dílčích funkcí, které dohromady uskuteční celý překlad. Využijeme při tom pomoc knihovny OpenCV obsahující řadu užitečných nástrojů, právě ke zpracování obrazu. U všech vytvořených metod si ukážeme schéma činnosti algoritmu a podrobně jej popíšeme. Zmínka bude i o nových datových typech, které

k výpočtu zavedeme. Okrajově se pak budeme zabývat tvorbou uživatelského rozhraní s využitím prostředků knihovny Qt. Předposlední kapitola 6 bude zjišťovat použitelnost představeného řešení. Provedeme sérii testů k ohodnocení aplikace s využitím rozdílných fotografií a skenovaných dokumentů. Určíme tak přibližnou úspěšnost překladu a tento údaj můžeme využít ke srovnání s podobně zaměřeným softwarem, který dnes na trhu je. V závěru potom celou práci shrneme a zamyslíme se nad jejím možným pokračováním, které by mohlo zvýšit úspěšnost překladu nebo přinést podporu pro zpracování dalších typů dokumentů.

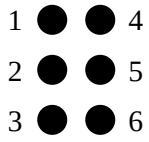
2 Braillovo písmo

Braillovo písmo, někdy označováno jako slepecké, je speciální druh písma pro nevidomé a slabozraké. Jedná se o skupiny bodů vyražené do papíru, které čtenář vnímá hmatem pomocí ukazováčku. Písmo nese název po svém objeviteli, kterým je francouzský učitel Louise Braille. Ten sám přišel o zrak po nešťastném úrazu ve 3 letech [1]. V mládí se seznámil s principy tzv. Barbierova písma, které v té době nevidomí používali. Využívala jej také francouzská armáda jako systém umožňující čtení ve tmě. Mělo však vážné nedostatky, a tak Braille koncem roku 1892 představil vlastní, mnohem jednodušší písmo, které daleko více vyhovovalo potřebám nevidomých [2].

Vznik písma byl v několika směrech revoluční. Přijetí toho systému ve Francii sice nebylo okamžité, ale díky poměrné snadné zapamatovatelnosti si neustále získávalo další příznivce. Do té doby bylo vzdělávání nevidomých velice problematické, jelikož neměli velké možnosti četby textu a přístup ke kultuře jim byl také odpírán. Vznik Braillova písma znamenal veliké zlepšení situace. Toto písmo bývá často dáváno jako příklad binárního kódu užívaného mimo počítače.

2.1 Formát znaků

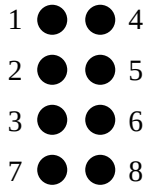
Jednotlivé znaky Braillova písma jsou tvořeny konfiguracemi šesti reliéfních bodů v matici se třemi řádky a dvěma sloupci, nazývaná braillská buňka (Obrázek 2.1). Na jednotlivé body se odkazuje číslicemi 1-6. Znak lze potom popsat čísly pozic, na kterých se nachází reliéfní bod.



Obrázek 2.1: Šestibodová braillská buňka

Na každé této pozici může anebo nemusí být umístěn jeden reliéfní bod. Tímto způsobem je možné vytvořit celkem $2^6 = 64$ různých znaků včetně mezery, která je reprezentována prázdnou buňkou. Zbývá tedy 63 použitelných kombinací pro ostatní znaky. Tento počet je však naprosto nedostačující na zapsání všech velkých i malých písmen abecedy, číslic, interpunkčních znamének a matematických symbolů. Proto se používají speciální znaky – prefixy, které mění význam jednoho nebo několika za nimi následujících znaků. Nejpoužívanější bývá změna velikosti písmen a prefix označující číslice.

Nutnost používání prefixů odpadá v osmibodové variantě Braillova písma, která byla vytvořena pro práci s počítačem (Obrázek 2.2). Rozšíření spočívá v přidání dalšího řádku, tedy bodů 7 a 8 pod stávající body 3 a 6. Tím vzrostl počet dostupných symbolů na 256 a je možné přímo zobrazit všechny znaky ASCII.



Obrázek 2.2: Osmibodová braillská buňka

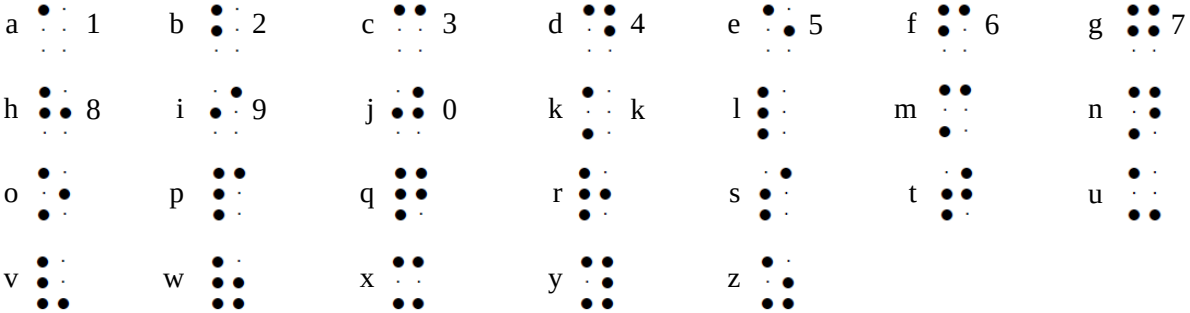
Tato varianta se však většinou používá jen na dynamických hmatových displejích zobrazujících text z části monitoru (Obrázek 2.3). Není příliš vhodná na tisk dokumentů a knih kvůli své velikosti. Jednak by narůstal objem tiskoviny a hlavně cit v posledním článku prstu, kterým se písmo čte, postupně od špičky ubývá. Čtvrtý řádek buňky se tak stává obtížněji rozeznatelný. Pro lepší čitelnost se v tisku stále využívá šestibodová varianta.

Obrázek 2.3: Braillský řádek

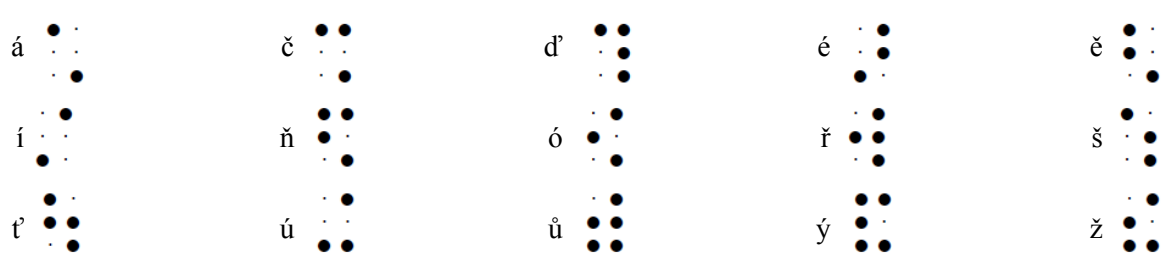
2.2 Česká abeceda

V současnosti používaná česká Braillova abeceda obsahuje 64 kombinací reliéfních bodů šestibodového písma. Základní písmena a-z mají reprezentaci shodnou s většinou jazyků používajících latinku. Drobné rozdíly spočívají v zápisu matematických symbolů a písmen s diakritikou, kterých má čeština ve srovnání s jinými jazyky mnohem více. Prázdný znak je vyhrazen pro mezeru, plný se využívá ke zlepšení orientace v textu.

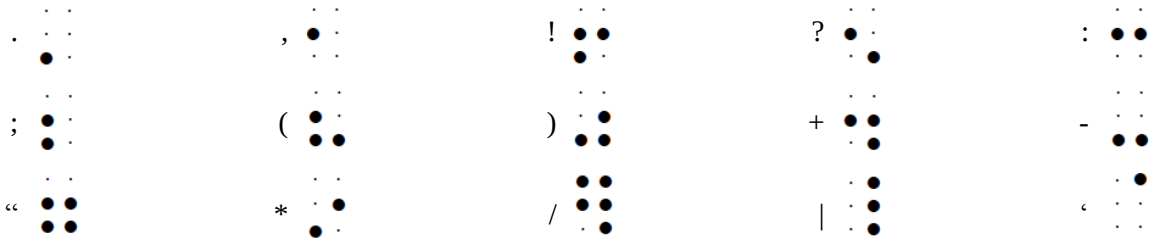
Podoba základních znaků české Braillovy abecedy:



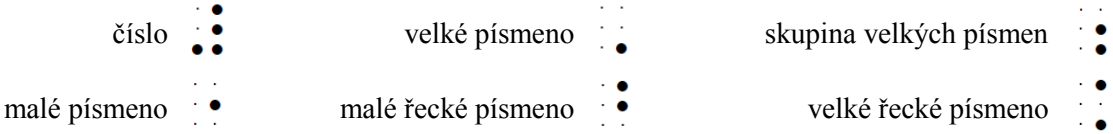
Znaky s diakritikou:



Interpunkce:



Prefixy:



2.3 Parametry písma

Při čtení hmatem vnímají nevidomí tvar konfigurace reliéfních bodů. Proto je třeba geometrické rozmístění pozic jednotlivých bodů důsledně dodržovat. Totéž platí i pro rozmístění braillských buněk na řádku a celých řádků na stránce. Pokud by se změnili vzdálenosti bodů nebo by byla konfigurace jiným způsobem pozměněna, písmo se stane nečitelné.

Parametry Braillova písma se sice mírně v různých zemích liší, nicméně standardní písmo by mělo splňovat následující kriteria [1]:

- průměr braillského bodu 1,5 mm
- horizontální i vertikální vzdálenost středů dvou sousedních bodů v téže buňce 2,5 mm
- vzdálenost středů dvou bodů na stejných pozicích v sousedních buňkách na řádku 6 mm
- vertikální vzdálenost středů dvou bodů na stejných pozicích v buňkách nad sebou 10 mm

Nutno dodat, že Braillovo písmo není zdaleka ideální, jak by se na první pohled mohlo zdát. Největší nevýhody se váží k tisku [3]. Braillské znaky jsou o mnoho větší než běžné znaky. Obvyklá velikost buňky je přibližně 5x7,5 mm. Při těchto rozměrech je možné zaznamenat na stránku formátu A4 zhruba 850 znaků. Běžného textu se však na stejnou stránku vejde asi 4500 znaků. Tiskoviny v Braillově písmu jsou tak přibližně 5x objemnější.

Další problém spočívá v typografii. Protože se na jeden řádek vejde kolem 30 znaků, často je nemožné použít běžné formátovací postupy jako např. tabulky. Místo toho je nutno psát veškeré informace sekvenčně, což jen dále navyšuje objem textu. S tím souvisí linearita zápisu. Výrazný problém, který se projevuje zejména při přepisu matematických a fyzikálních textů. Kvůli nutnosti používat prefixy pro čísla a znaménka mohou i jednoduché rovnice zabírat několik řádků. Určitý problém je také nejednotnost Braillova systému ve světě. Různé země používají různé normy písma, které se navzájem poměrně dost odlišují. Příkladem budiž česká a slovenská norma, ve které najdeme jen v základních písmenech abecedy 7 rozdílů.

2.4 Metody tisku

Současně s rozvojem písma se vyvíjeli také způsoby psaní a tisku. Tento vztah je závislý na konkrétních technických možnostech dané doby [2]. Začínalo se tak jednoduchými šablonami, které byly postupem času zdokonalovány. V tomto směru představovala vrchol ručního psaní tzv. Pražská tabulka. Rozvoj typografie poté přinesl rychlejší a pohodlnější možnost zápisu pomocí psacího stroje. Zatím nejrychlejší a nejmodernější metoda vzešla z rozmachu informačních technologií. Braillovo písmo je možné tisknout ve vysoké kvalitě a dokonce oboustranně na speciálních tiskárnách. V současné době se tedy k záznamu Braillova písma využívají tři základní způsoby [1]:

- ruční psaní bodátkem na šabloně (Pražská tabulka)
- na mechanických psacích strojích (Pichtův psací stroj)
- tiskem na počítačových tiskárnách

Pražská tabulka

Na ruční psaní se využívá speciální kovová šablona zvaná pražská tabulka a oblé bodátko (Obrázek 2.4). Tabulka se skládá ze dvou desek, mezi které se vloží papír. Horní deska má tvar mřížky s obdélníkovými okénky, které svou velikostí odpovídají rozměrům braillské buňky. Okraje okének jsou tvarované tak, aby se jednotlivé body daly dobře vypichovat. Spodní deska obsahuje řady vydutých plných znaků, což umožňuje bodátkem vytvarovat do papíru reliéfní body.

Bodátko se skládá z držátka a kovové jehly o délce 1 cm. Průměr jehly je 1 mm a s přičtením tloušťky papíru, odpovídá průměr velikosti reliéfního bodu. Držátko bývá vyrobeno ze dřeva a mívá nejčastěji trojúhelníkový tvar. Po vpichu se do papíru vytvaruje jamka, což znamená, že píšeme negativ. Aby bylo možné výsledný text přečíst, musí se stránka obrátit.

Obrázek 2.4: Pražská tabulka

Tabulky se vyrábí s různým počtem řádků a znaků na nich. Běžné jsou i varianty na oboustranné psaní, kdy je text na jedné straně umístěn mezi řádky na druhé straně. Vznikají sice větší mezery, ale protože je papír popsaný s obou stran, celkové množství textu na listu je podstatně vyšší. Širší mezery jsou rovněž pro začínající čtenáře pohodlnější.

Existují dokonce tabulky umožňující oboustranné mezibodové psaní, kdy jsou body na jedné straně umístěny přímo mezi body z druhé strany. Při používání taktových tabulek se nejprve napíše text z jedné strany, pak se papír otočí a pokračuje se druhou stranou. Tato metoda klade velké nároky na přesnost, aby nedocházelo k deformaci obrácených bodů.

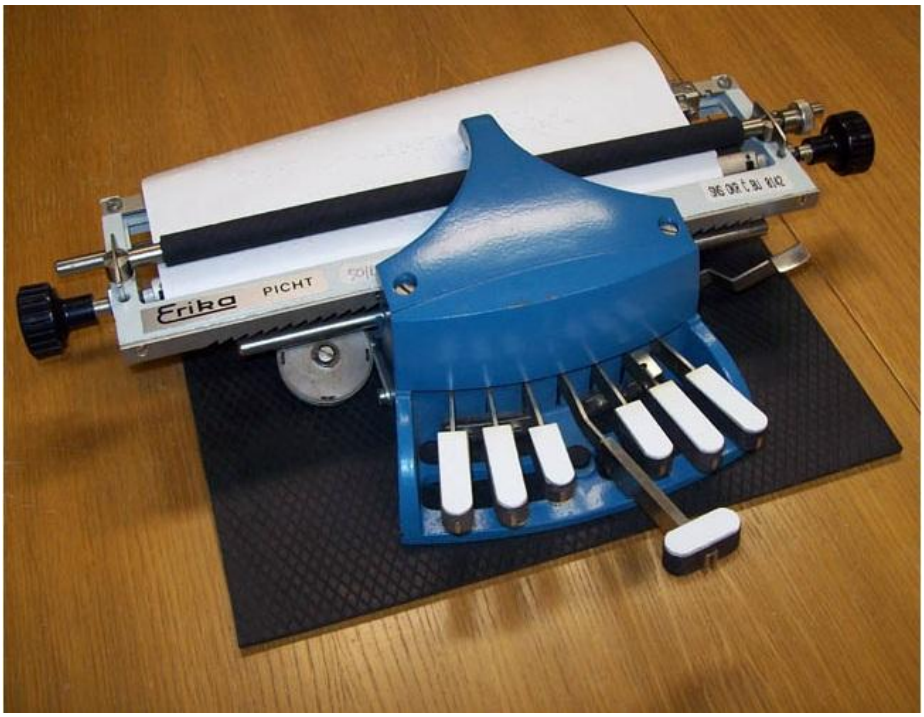
Pomocí tabulky zapíší zkušení pisatelé 10 až 15 slov za minutu. Tento způsob se však v současnosti používá už jen zřídka, především mezi staršími uživateli nebo na zápis krátkých poznámek. Na psaní delších textů se prakticky nepoužívají. V této úloze je zcela nahradily slepecké psací stroje a speciální tiskárny na Braillovo písmo.

Pichtův psací stroj

Slepecké psací stroje výrazně zvyšují rychlost a kvalitu psaní. U nás se nejčastěji používá Pichtův psací stroj (Obrázek 2.5), který sestrojil ředitel berlínského ústavu pro nevidomé Oskar Picht. Jeho stroj obsahuje 7 kláves. Šest je určeno pro psaní bodů a sedmá pro zápis mezery mezi slova. Prostřední klávesa, určená právě pro mezery, je prodloužená a její délka nastavitelná tak, aby se dala obsluhovat dlaní. Díky tomu je možné psát jen jednou rukou a druhá je k dispozici na čtení opisovaného textu, nebo na kontrolu napsaného textu. Na rozdíl od zápisu pomocí tabulky, je okamžitá kontrola možná, jelikož se body do papíru vpichují zespodu a tvoří se pozitivní reliéf.

Celé písmeno je možné napsat jedním současným stlačením příslušné kombinace kláves. Ve srovnání s pražskou tabulkou je rychlost psaní asi 4x vyšší. Pichtovy stroje se vyrábí ve třech variantách. První je pro samostatné psaní levou rukou, další pro psaní pravou a poslední pro psaní oběma rukama současně.

Kromě principu Pichtova stroje, kde se pohybuje válec s papírem a razící hlava stojí na místě, existuje i obrácená varianta. Tento konstrukční princip se nazývá Perkins Brailler. Stroje mají pohyblivou hlavu a nehybný válec s papírem. Jsou o něco menší, ale také dražší, jelikož jsou technicky náročnější.



Obrázek 2.5: Pichtův psací stroj

Braillské tiskárny

Počítačových tiskáren na Braillovo písmo je poměrně dost. Liší se především svým výkonem. Nejjednodušší verze tisknou asi 6 znaků za sekundu a pouze jednostranně. Nejmodernější tiskárny zvládají více než 400 znaků za sekundu a umožňují tisk z obou stran. Některé dovolují tisknout i grafiku sestavenou z reliéfních bodů. Jednodušší typy většinou používají skládaný, perforovaný papír, který po dokončení vlastního tisku vyžaduje ještě finální úpravy v podobě oddělení jednotlivých listů a ořezání okrajů. Výkonnější tiskárny jsou vybavené podavači jednotlivých papírů a ty nejvýkonnější odebírají papír přímo z role a automaticky ho řežou na požadovaný formát. Potištěné stránky se ukládají v pořadí, jak mají být svázané (Obrázek 2.6).

Braillské tiskárny představují velký pokrok ve výrobě knih a časopisů vycházejících v Braillově písmu. Rychlý tisk rozsáhlých textů má významný přínos i pro zabezpečení potřeb jednotlivců a jejich individuální vzdělávání, zaměstnání a společenské uplatnění.

Obrázek 2.6: Tiskárna Braillo 400 SR

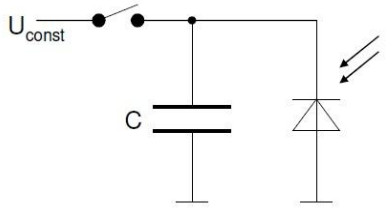
3 Snímání obrazu

Jak bylo řečeno v úvodu, práce se zabývá převodem Braillova písma z fotografie. Nyní si tedy přiblížíme současné metody zachycení snímku pro následné zpracování počítačem. O pořizování obrazu se starají obrazové senzory. Jejich vlastnosti a výrobní technologie mají značný vliv na výslednou kvalitu snímku, ať už se jedná o digitální fotoaparát nebo skener. Podívejme se tedy, jak jsou tyto senzory zkonstruovány a jaké mají parametry [4].

3.1 Senzory

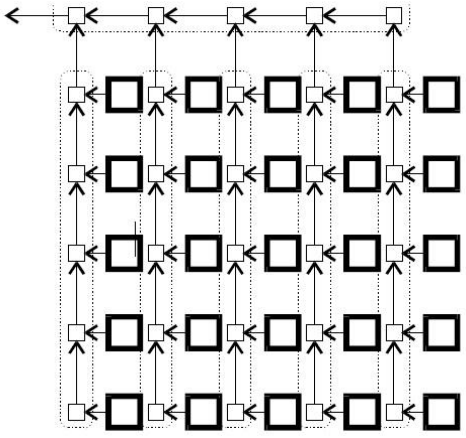
CCD senzor

CCD (Charge Coupled Device) senzory jsou polovodičové čipy, převádějící promítnutý obraz na elektrický náboj. Ten je pak dalšími strukturami přenesen na okraj čipu a nakonec převeden na napětí. Obraz je nejprve pořízen v elektrické podobě. Čip obsahuje struktury podobné kondenzátorům, které se před začátkem expozice nabijí konstantním nábojem. V průběhu expozice se napětí přestane přivádět. Plocha čipu je rozdělena na části zvané pixely. Vlivem dopadajícího světla se jednotlivé kondenzátory vybíjejí úměrně podle množství absorbované energie. Schematická podoba světlocitlivých buněk (pixelů) je znázorněna na obrázku (Obrázek 3.1).



Obrázek 3.1: Schéma světlocitlivé buňky

Po skončení expozice se náboj zbylý na kondenzátorech přesune do transportních registrů. Jsou to posuvné registry, které již nereagují na světlo a jsou schopné náboj beze změny posouvat. Pracují v soustavě uspořádaných řádků a sloupců (Obrázek 3.2) a postupně odsouvají náboj ze všech pixelů do místa, kde je integrován převodník náboje na napětí.

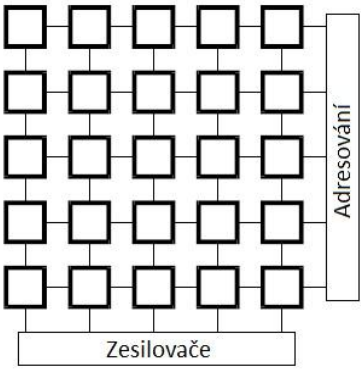


Obrázek 3.2: Struktura transportních registrů

CCD technologie umožňuje konstrukci senzorů, které exponují celý obraz současně a disponují velmi nízkým šumem. Nevýhodou však je relativně vysoká spotřeba, nutnost používat řadu napájecích napětí a nemožnost integrovat veškeré řídící obvody a A/D převodníky přímo do čipu senzoru. I přes to, že je CCD technologie poměrně stará, poskytuje nejkvalitnější možnosti snímání obrazu, které v současné době máme. CCD čipy se dnes vyrábí ve velikosti až 34 megapixelů. Hojně se využívají v digitálních fotoaparátech i videokamerách.

CMOS senzor

Senzory typu CMOS (Complementary Metal Oxid Semiconductor) obsahují podobné světlocitlivé struktury jako senzory CCD (Obrázek 3.1). Hodnota náboje je však převáděna na napětí přímo v jednotlivých buňkách, které jsou adresovatelné (Obrázek 3.3). Toto napětí je pak přenášeno na okraj čipu přes soustavu analogových multiplexorů a nakonec digitalizováno.



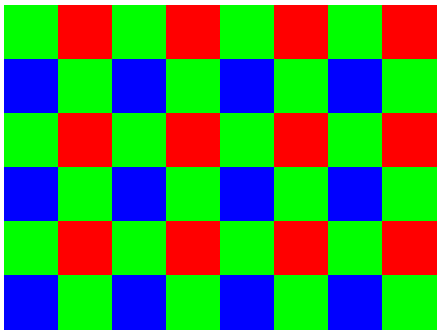
Obrázek 3.3: Schéma CMOS senzoru

Jelikož čtení pixelů probíhá sekvenčně a ne současně jako u CCD snímačů, je třeba tomu přizpůsobit expozici, pokud má být expoziční doba všech pixelů stejná. Proto odpojení kondenzátorů z konstantního napětí musí probíhat ve stejných intervalech, jako vyčítání obsahu. Je-li obraz promítaný na čip neměnný, není tento postup nutný. Jestliže se však obraz pohybuje, může být výsledek zkreslený. Kvůli tomu nejsou tyto senzory vhodné pro měřící účely.

Nevýhodou CMOS technologie je výskyt strukturálního šumu. Další vlastnosti jsou však příznivé. Na čip lze integrovat řídící elektroniku i převodníky. Je možné používat jediné napájecí napětí a mají výrazně nižší spotřebu. Množství čipů využívaných ve fotoaparátech je srovnatelné s počtem vyžívaných CCD čipů.

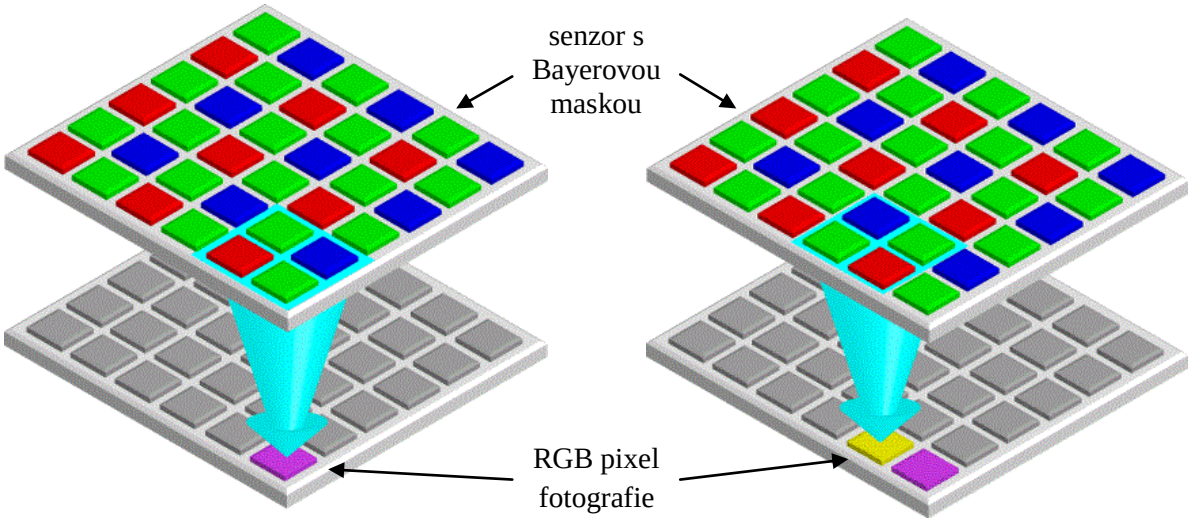
3.2 Digitální fotoaparáty

Digitální fotoaparát je zařízení pro záznam obrazu v digitální podobě. Existuje celá řada typů, které se liší svým zaměřením, velikostí, kvalitou a mnoha dalšími vlastnostmi. Ať už se jedná o fotoaparát v mobilu, klasický kompakt či vyspělou zrcadlovku, základní princip je u všech přístrojů stejný. Je třeba promítnout obraz na plochu senzoru a získanou intenzitu světla uložit v příslušném formátu. Jak víme, buňky senzoru zaznamenávají pouze množství světla, které na ně dopadá, čímž vzniká černobílý obraz. Aby vznik obraz barevný, jsou nad jednotlivými buňkami umístěny různě barevné filtry. Nejrozšířenější jsou odstíny červené, zelené a modré uspořádané do pravidelné struktury zvané Bayerova maska [5]. Většina výrobců dnes používá RGBG mozaiku, která je vidět na následujícím obrázku (Obrázek 3.4).



Obrázek 3.4: Příklad Bayerovy masky

Rozložení filtrů je 2:1:1 ve prospěch zelené kvůli vyšší citlivosti lidského oka. Jelikož každá buňka rozpozná jen jednu barvu, výslednou barvu je třeba spočítat, k čemuž se využívají hodnoty okolních bodů. Tento způsob označujeme jako Bayerova interpolace (Obrázek 3.5) a bývá používán ve většině digitálních fotoaparátů a skenerů.



Obrázek 3.5: Bayerova interpolace

Princip

Nyní se podíváme, jak fotoaparát pracuje od okamžiku stisknutí spouště do uložení snímku [6]. V klidovém stavu (Obrázek 3.6 vlevo), když se díváme do hledáčku, prochází světlo objektivem, v jehož středu je umístěna clona. Ta je otevřena na maximum, aby byl obraz v hledáčku co nejjasnější a senzory měli dostatek světla pro svoji činnost. Světlo dopadá na zrcátko skloněné v úhlu 45 stupňů a odráží ho na matnici, což je průhledné plátno, na kterém se promítne obraz a tak jej můžeme okem sledovat. Po průchodu objektivem je však převrácený a je třeba ho pomocí hranolu před hledáčkem otočit nazpět. Jakmile stiskneme spoušť (Obrázek 3.6 vpravo), poměry v přístroji se dramaticky změní. Obě zrcátka se sklopí vzhůru, takže přestanou clonit senzor a naopak zakryjí hledáček. Clona v objektivu se uzavře podle nastavené hodnoty a otevře se závěrka. Světlo tak může dopadat na senzor. Po uplynutí doby expozice je závěrka opět uzavřena, clona se roztáhne na maximum a zajistí tak přístup světla. Nakonec se obě zrcátka sklopí zase dolů, aby bylo možné sledovat v hledáčku nový obraz.

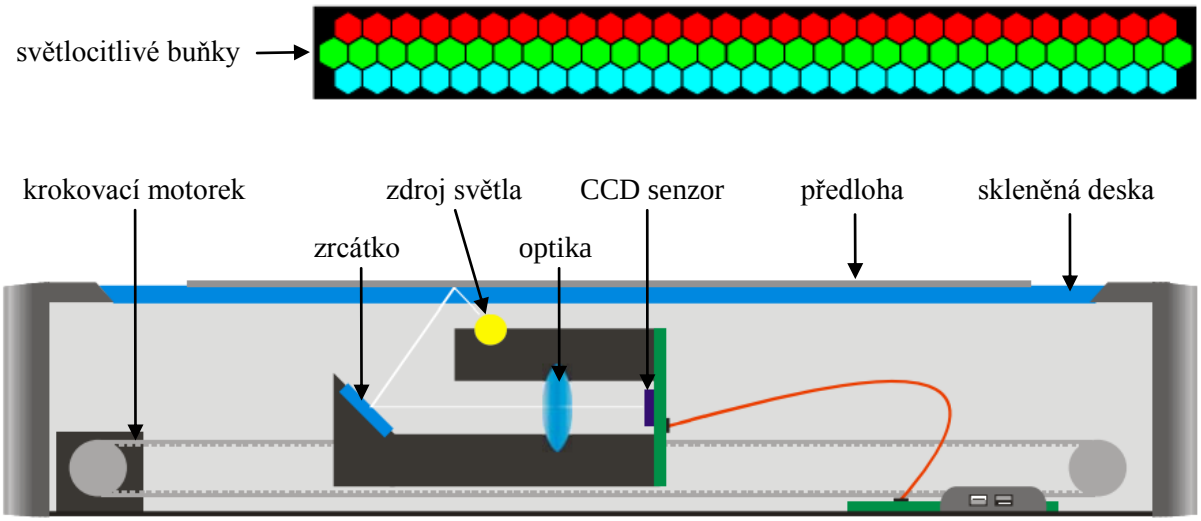
Obrázek 3.6: Konstrukce digitální zrcadlovky

3.3 Skenery

Jako skener obecně označujeme zařízení schopné něco snímat [7]. V našem případě hardwarové zařízení určené pro načítání obrazových dat. Dlouhou dobu se jednalo o základní periferii, jelikož představovala jediný způsob načtení fotografie do počítače. Dnes už je tento způsob na ústupu vzhledem k rozvoji digitálních fotoaparátů. Nicméně pro některé případy je stále vhodnější využít skener, zejména jedná-li se o dokumenty s textovým obsahem. Skenované stránky jsou rovné a stejnoměrně osvětlené, čehož bychom pouhým focením obtížně dosáhli. Skenery můžeme rozdělit do dvou základních skupin, a to bubnové a plošné. Z technického hlediska je však podstatnější rozdělení podle skenované předlohy na průsvitné (např. film či diapozitiv) a odrazové (fotografie nebo tištěné materiály). Některé typy skenerů vyžadují upevněné předlohy na podložku nebo do speciálního rámu, což označujeme jako skenovací montáž. Typickým příkladem je lepení předloh na buben u bubnového skeneru.

Princip

Funkce všech typů skenerů je v podstatě totožná. Zdroj světla postupně osvětluje předlohu. Světlo, které se odrazí (nebo projde v případě průsvitné předlohy) je zachyceno senzorem. U plošných skenerů se nejčastěji využívají CCD snímače. Ve světlocitlivých buňkách vzniká v důsledku osvícení elektrické napětí. Od světlých částí předlohy se odráží více světla, čímž získáme i vyšší napětí. Od tmavších ploch se naopak světla odrazí méně, tudíž je napětí nižší. Nakonec je napětí zpracováno pomocí A/D převodníku a obraz uložen v digitální podobě. U plošných skenerů je snímán najednou celý řádek předlohy, který se ukládá jako řádek pixelů rastrového obrazu. Senzor má tvar lišty, na které jsou umístěny 3 řady světlocitlivých buněk. Každá řada je opatřena jedním filtrem ze skupiny RGB a snímá tak pouze příslušnou barvu. Obrázek 3.7 představuje zjednodušené schéma plošného skeneru.



Obrázek 3.7: Konstrukce plošného skeneru

Parametry

Na hodnocení kvality skenerů ale i fotoaparátů a jejich výstupního obrazu se využívají různé parametry. Mezi ty nejdůležitější a objektivně vyčíslitelné řadíme velikost rozlišení, barevnou hloubku a minimální a maximální denzitu.

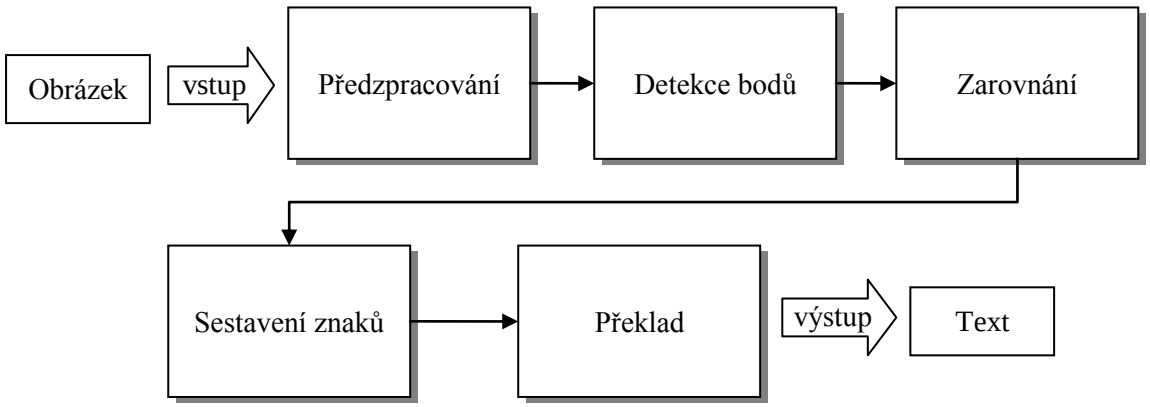
- Rozlišení – udává počet pixelů na palec (centimetr), které je skener schopen rozlišit. Můžeme říct, že čím vyšší hodnota, tím lépe. Jednotkou je ppi (pixel per inch = pixelů na palex) nebo dpi (dot per inch = bodů na palec). Bubnové skenery mají rozlišení obvykle 5000 – 15000 dpi, plošné kolem 2000 – 5000 dpi. Hodnota se často zadává ve tvaru násobku (např. 2000 x 2000 dpi). První údaj značí rozlišení snímače a druhý počet kroků posouvacího motorku.
- Barevná hloubka – označuje maximální počet barev, které je zařízení schopné zaznamenat. Udává se v bitech buď vztažená na jeden barevný kanál, nebo celkovou hodnotu. V současnosti je standardem hloubka 8 bitů na složku RGB (24b celkem), díky čemuž je možné rozeznat přibližně 16,7 milionu barev. Profesionální zařízení pracují s vyšší barevnou hloubkou a to až 16 bitů na kanál.
- Denzita – je termín, který udává optickou hustotu a značí míru propustnosti světla. Čím je vyšší, tím více světlo pohlcuje. Maximální a minimální denzita nám tedy určuje kvalitu snímání velice tmavých a velice světlých míst. U kvalitních přístrojů by měla denzita být alespoň 3. Ty nejkvalitnější pak dosahují hodnoty až 4,8.

Mimo tyto parametry se však k celkovému hodnocení přidává řada dalších, které nelze číselně ohodnotit. Jedná se např. o celkovou kvalitu snímání danou konstrukcí jednotlivých částí přístroje, rychlost zpracování, maximální formát snímané plochy a mnoho dalších vlastností.

4 Návrh

Dosud jsme se bavili pouze obecně. Nyní se podíváme trochu blíže na návrh programu, který bude schopen ze vstupní fotografie obsahující Braillovo písmo, dát na výstup odpovídající překlad v latince.

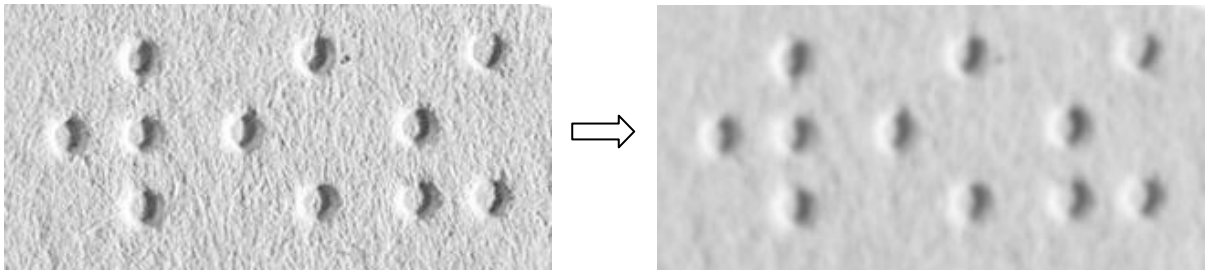
Při zpracování obrazu se aplikace standardně dělí na několik fází. První většinou bývá určité předzpracování, které má za úkol vyčištění vstupu od nežádoucího šumu, případně další opravy. Pak probíhá extrakce důležitých rysů, které v obraze sledujeme. Poté následuje se získanými daty nějaký výpočet a v poslední fázi dojde k zobrazení výsledků [8]. Pro náš konkrétní případ by mohla série požadovaných úkonů vypadat takto (Obrázek 4.1):



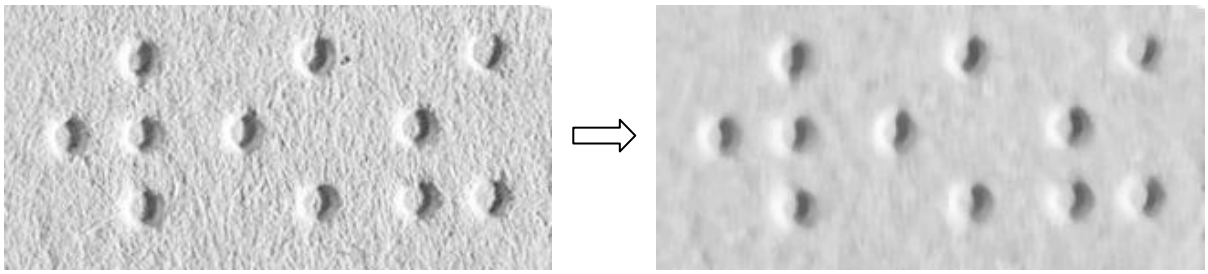
Obrázek 4.1: Jednotlivé fáze algoritmu

4.1 Předzpracování

Jelikož nikdy nevíme, jak kvalitní fotografii na vstupu dostaneme, je vice než vhodné ji nějakým způsobem ošetřit, abychom se vyhnuli pozdějším komplikacím. Pokud dostaneme fotografii, kde je Braillovo písmo zřetelné, nebyl by tento krok nutný. Např. nachází-li se na lesklém kovovém štítku někde u památníku. Problém by mohl nastat v případech, kdy fotografujeme text na starším hrubém papíru. V tomhle případě už jednotlivé body tolik zřetelné nejsou a samotný papír navíc ve výsledku způsobuje určitý druh šumu. Právě tyhle odchylky jednotlivých pixelů jdou odstranit slabým rozostřením (Obrázek 4.2) nebo ještě lépe použitím mediánového filtru (Obrázek 4.3).



Obrázek 4.2: Slabé gaussovské rozostření



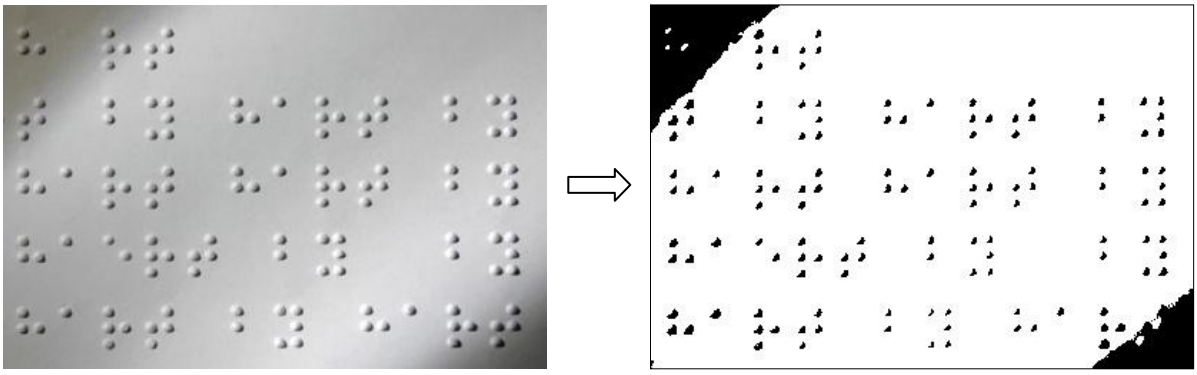
Obrázek 4.3: Aplikace mediánového filtru

4.2 Detekce bodů

Jedná se o klíčovou část návrhu. Na tom jak přesně a zda vůbec se podaří reliéfní body detekovat, závisí celé aplikace. Na předchozím obrázku (Obrázek 4.3) můžeme krásně vidět, že každý bod obsahuje zřetelně tmavší místo, což je způsobeno nerovnoměrným dopadem světla na povrch. A právě této skutečnosti jsem se rozhodl využít. Pro oddělení jednotlivých bodů od pozadí obrázku budu používat prahování [9].

Prahování převádí obraz ve stupních šedi do dvoubarevné podoby. Tmavá místa, která chceme získat, mají nízkou intenzitu a pozadí má naopak hodnotu vyšší. Podle tohoto kriteria můžeme rozdělit jednotlivé pixely do dvou různých skupin. Jen je nutné určit hodnotu odstínu, která bude skupiny oddělovat, tzv. práh. Pokud je hodnota prahu stejná pro celý obrázek, jedná se o tzv. globální prahování.

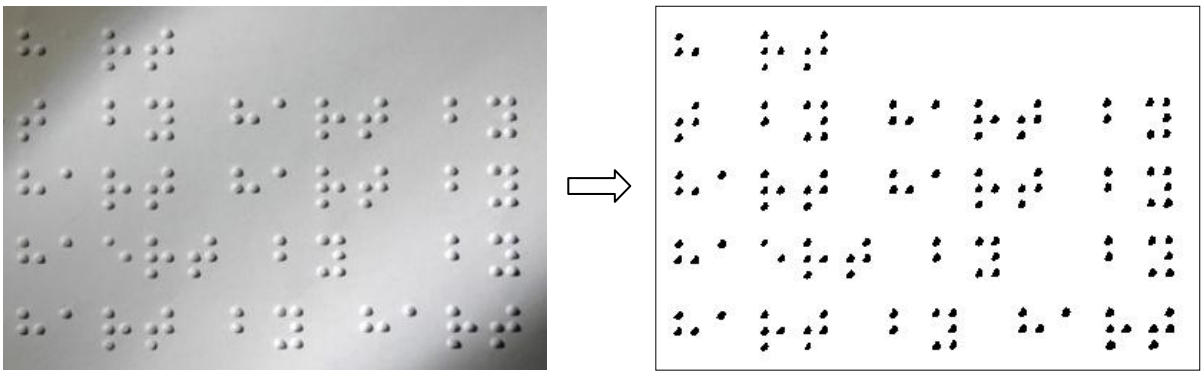
Poté lze snadno sestavit nový binární obraz. Hodnota každého vstupního bodu se porovná s prahem. Je-li vyšší, výstupní hodnota bude značit bílou barvu, bude-li nižší než práh, na výstupu se objeví černá barva. Tímto postupem získám masku, která bude na bílém pozadí obsahovat různé tmavé oblasti. V ideálním případě budou jednotlivé oblasti odpovídat reliéfním bodům. Problém však nastane, pokud je ve vstupním obraze příliš prudká změna v osvětlení. Potom by došlo ve tmavých částech ke ztrátě informace, jak je vidět na následujícím obrázku (Obrázek 4.4):



Obrázek 4.4: Výsledek prahování s globálním prahem

Toto nežádoucí chování je možné odstranit, pokud nebudeme používat stejnou hodnotu prahu pro celý obrázek. Místo toho se pro každý pixel spočítá individuální práh z jeho blízkého okolí. Tato varianta se označuje jako adaptivní prahování. Velikost okolí je však třeba dobře zvolit. Musí obsáhnout dostatečné množství pixelů, jinak by se prahová hodnota mohla nesprávně určit. Pokud by

bylo okolí příliš velké, mohla by být porušena základní myšlenka této metody, že osvětlení se kolem aktuálního bodu nemění a výsledek by byl stejný jako v případě globálního prahu. Obrázek 4.5 ukazuje aplikaci adaptivního prahu při správně zvolené velikosti okolí. Na první pohled je patrné, že tato metoda s různou intenzitou osvětlení v obrázku neměla potíže. Kvalita výstupu je ve srovnání s předchozí metodou o mnoho vyšší.



Obrázek 4.5: Výsledek adaptivního prahování

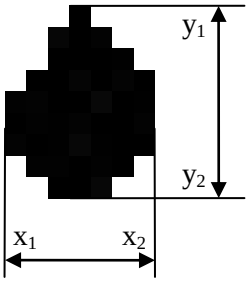
Jakmile budeme mít obraz s reliéfními body, zbývá ještě určit jejich souřadnice. Na tento úkol jsem se rozhodl upravit algoritmus pro semínkové vyplňování. Ten slouží pro obarvování uzavřených rastrových oblastí a funguje následovně:

1. Vlož počáteční bod (semínko) do fronty
2. Dokud není fronta prázdná, opakuj:
    a. Vyber semínko ze začátku fronty
    b. Pokud je bod uvnitř oblasti, nastav hodnotu pixelu a vlož do fronty sousední body

V našem případě však není potřeba nic obarvovat, ale získat přibližné souřadnice středů jednotlivých černých oblastí. Proto upravím krok 2.b z původního algoritmu, abych místo změny barvy vnitřního pixelu, uložil jeho pozici. Tím získám seznam všech bodů tvořících oblast. Pak už stačí projít seznam, nelézt nejmenší a největší hodnotu v obou osách a určit souřadnice středu $[x_S, y_S]$. Výpočet je vidět níže (Obrázek 4.6).

$$x_S = x_1 + (x_2 - x_1) / 2$$

$$y_S = y_1 + (y_2 - y_1) / 2$$

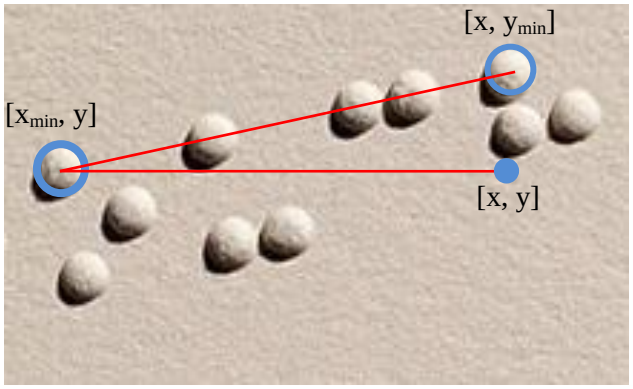


Obrázek 4.6: Výpočet středu oblasti

Aplikací takto upraveného algoritmu na celou masku postupně získám pozice všech reliéfních bodů detekovaných při prahování. Seznam souřadnic pak vstupuje do další fáze zpracování.

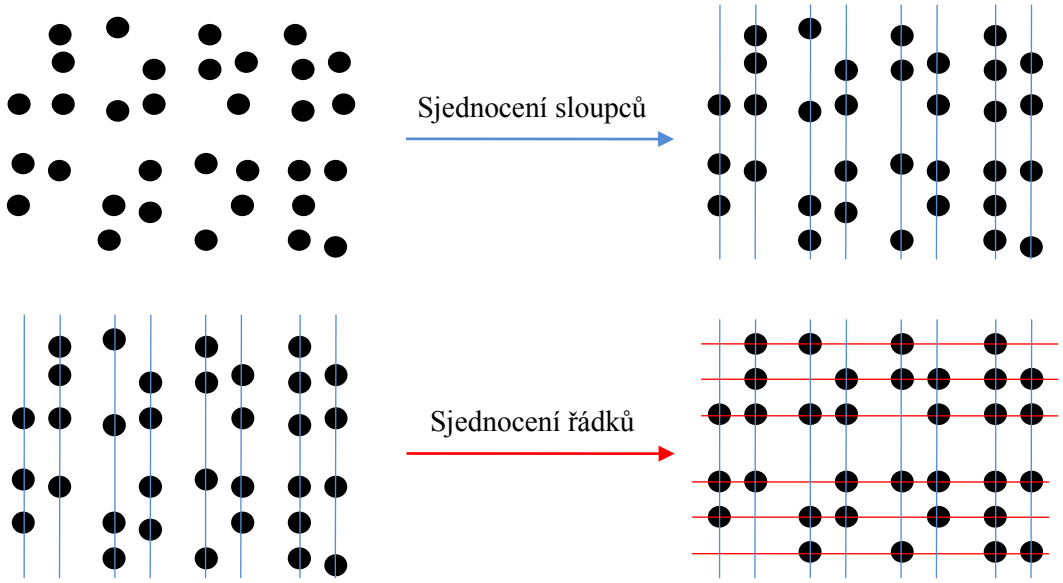
4.3 Zarovnání

Nemůžeme předpokládat, že uspořádání reliéfních bodů na fotografii bude vždy přesně podél osy x a y. Ve většině případů je text více či méně šikmo. Proto je nutné body nejdříve pootočit, aby byl jejich odklon od směru osy co nejmenší. Pro určení odchylky projdu seznam souřadnic a budu hledat extrémy v obou osách, jak ukazuje Obrázek 4.7. Z těchto hodnot určím úhel, o který je třeba všechny body pootočit. Tímto způsobem se provede první fáze srovnání, při které se bude hýbat s celým obrazem.



Obrázek 4.7: Zjištění úhlu odklonu

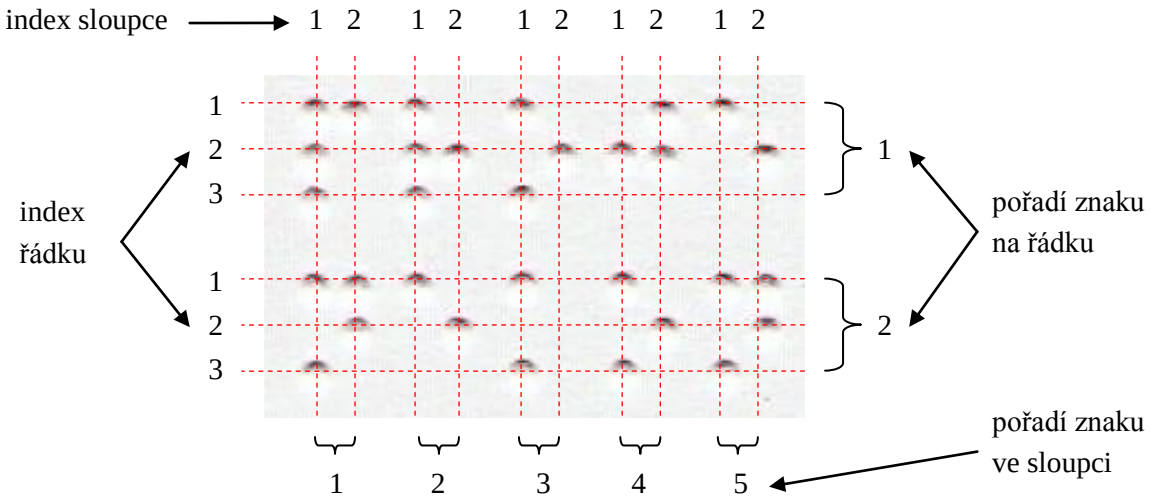
Poté co jsou body otočeny tak, aby odklon jejich roviny v řádku od osy x byl co nejmenší, nastává druhá část zarovnání. X-ová souřadnice bodů ve sloupcích a y-ová v řádcích se bude lehce lišit. I když je obrázek v první fázi na pohled naprosto přesně pootočen, nikdy nebudou souřadnice řádků a sloupců jednotné. Mírná odchylka v řádech několika pixelů tak představuje další překážku pro přesné určení znaků. Proto je třeba před dalším zpracováním jednotlivé řádky a sloupce sjednotit na stejnou hodnotu. Nejjednodušší způsob jak to udělat, je vzít souřadnice každého bodu ve sloupci, spočítat průměrnou hodnotu a tuto novou souřadnici pak přiřadit všem bodům v příslušném sloupci. Tento postup se pak aplikuje i na souřadnici y pro jednotlivé řádky a reliéfní body tak budou zarovnané do mřížky, jak je vidět na Obrázek 4.8.



Obrázek 4.8: Zarovnání souřadnic

4.4 Sestavení znaků

Když budou body přesně zarovnané, je možné na základě vzdáleností mezi nimi sestavit jednotlivé braillské buňky. Jinými slovy je třeba každý bod zařadit do odpovídajícího znaku. Víme, že buňku tvoří vždy 2 sloupce a 3 řádky a každá pozice v ní je určena číslem 1-6. Pokud jednotlivé souřadnice ve sloupcích a řádcích označíme pořadovým indexem, bude možné všechny body přiřadit do správné buňky. Způsob jak to provést, je na základě porovnání mezer mezi sousedními souřadnicemi. Menší vzdálenost značí, že body patří do stejné skupiny, větší naopak, že se jedná o bod z nové buňky. Označení indexy je vidět na Obrázek 4.9 (vlevo).



Obrázek 4.9: Indexování souřadnic

K absolutní identifikaci jednotlivých znaků by však samotné rozdělení souřadnic do skupin ještě nestačilo. Tímto určíme jen polohu bodů v buňce. Dále je potřeba určit pozici buňky v obrázku, což vede na další číslování. Tentokrát budeme označovat každé 2 sloupce a 3 řádky pořadovým indexem, který zajistí identifikaci celých buněk. Zjednodušeně můžeme říct, že počítadlo znaků na řádku a ve sloupci zvýšíme pokaždé, když narazíme na jedničku v indexování souřadnic. Situaci znázorňuje Obrázek 4.9 (vpravo).

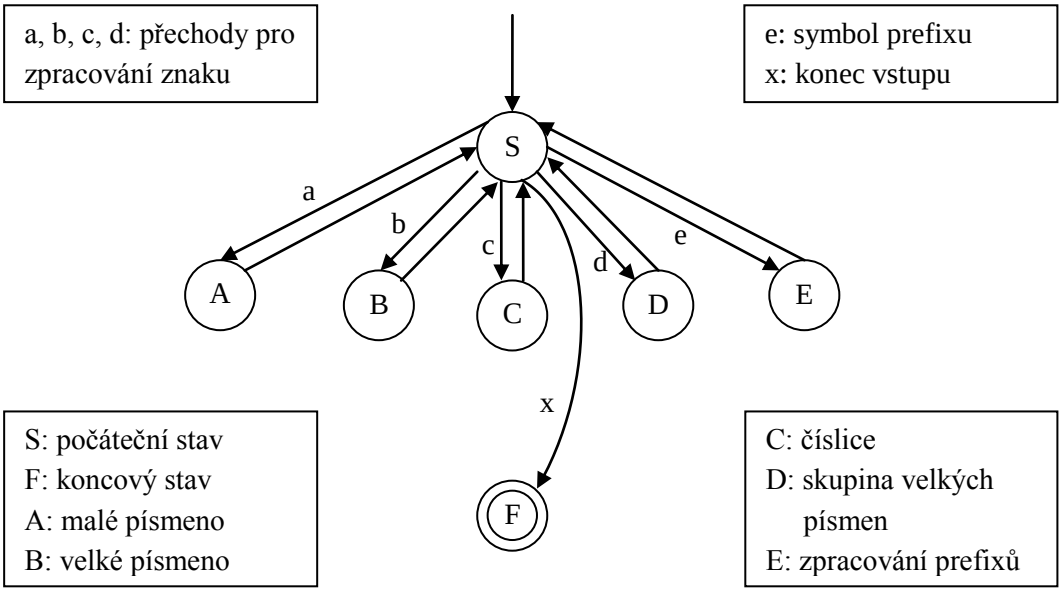
Po označení všech řádků a sloupců potřebnými indexy, můžeme vytvořit ze seznamu bodů jednotlivé braillské buňky. Zatím byl každý bod určen svou souřadnicí x a y, která značila jeho polohu v obrázku. Pro překlad je však nutné brát symboly jako komplexní celek šestice bodů a ne každý samostatně. Ve výsledku tak bude každý znak reprezentován jako posloupnost šesti binárních hodnot, kde 1 značí, že na dané pozici je reliéfní bod a 0 nikoliv. Pokud vyjdeme z předchozího obrázku, zpracování by vypadalo následovně:

1. Pořadí znaku ve sloupci a na řádku identifikuje buňku, se kterou se bude pracovat. Body jsou díky sekvenčnímu průchodu obrázkem uloženy po řádcích. Jako první tak přijde na řadu bod, který patří do znaku (1, 1)
2. Indexy tohoto bodu jsou též [1, 1]. Tato konfigurace tedy změní počáteční hodnotu znaku z “000000” na “100000”. Druhý v pořadí je bod ze stejné buňky s indexy [2, 1]. Index sloupce 2 udává, že se jedná o druhou polovinu znaku a provedeme změnu z “100000” na “100100”. Další v pořadí je bod nové buňky s indexy [1, 1]. Provedeme opět změnu z počáteční hodnoty “000000” na “100000” atd.

Tímto způsobem projdeme celý seznam vstupních bodů, díky čemuž se postupně zpřesňují počáteční hodnoty “000000” všech znaků na podobu jednotlivých braillských buněk. Díky tomuto převodu máme k dispozici první textový zápis znaků, se kterým už je mnohem snazší manipulace než s pouhým výčtem souřadnic značící reliéfní body.

4.5 Překlad

Po získání seznamu s jednotlivými znaky následuje poslední fáze algoritmu a to samotný překlad do latinky. Pravidla pro zápis Braillova písma jsou dána standardem a podrobně popsány. Na jejich základě je možné sestrojit automat, který bude znaky číst a generovat odpovídající výstup. Zjednodušené schéma automatu je na Obrázek 4.10.



Obrázek 4.10: Překladový automat

Každý symbol na vstupu reprezentuje buď konkrétní znak, nebo značí definovanou změnu následujícího (prefix). V případě malých písmen abecedy se ihned přejde do stavu A, ve kterém se jeho hodnota vloží do výstupního řetězce. Poté následuje návrat do stavu S, kde pokračuje čtení dalšího symbolu. Stavy B, C a D nejsou dostupné přímo, ale pouze v případě, že aktuálnímu znaku předcházel některý prefix. Zpracování prefixů a nastavení příznaku jeho aktivity provedeme ve stavu E, pokud se takový symbol vyskytne ve vstupní posloupnosti. Jeho aktivací se ovlivní, do kterého stavu budou směřovat následující symboly, a tedy jaké znaky budou přidávány do výstupního řetězce. Po přečtení celé vstupní sekvence přejdeme do stavu F, čímž se překlad ukončí.

5 Implementace

Aplikace je psána v jazyce C++ s využitím různých knihoven. Její vývoj můžeme rozdělit na dva hlavní celky. Jedná se o část pro detekci reliéfních bodů a překlad písma, při které budu využívat prostředky ke zpracování obrazu z knihovny OpenCV [8]. Druhá část pak zahrnuje uživatelské rozhraní, v jehož vývoji mi bude pomáhat další knihovna, a sice Qt [10]. Oba zmíněné nástroje jsou dnes poměrně hojně využívané a přeložitelné na mnoha platformách.

5.1 Třída Cbraille

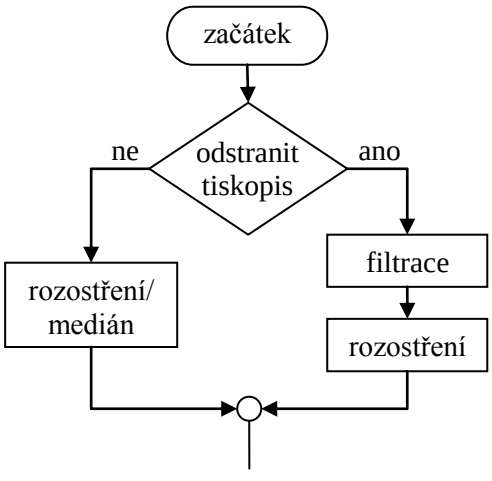
Obsahuje metody pro postupné zpracování obrázku od inicializace po překlad. Obsah jednotlivých metod vychází z fází probraných v návrhu. Dále jsou zde definovány nové datové typy a atributy pro uchování mezivýsledků a přehlednější výpočet. Nyní si představíme nejdůležitější součásti třídy podrobněji.

Metoda init

Veřejná metoda `void init(IplImage* image, int threshold, int gauss = 2, int form = 0, bool grid = false)` slouží k počátečnímu nastavení atributů důležitých pro výpočet. Je třeba ji zavolat jako první před zahájením překladu, jelikož se přes jeji první parametr `image` předává fotografie ke zpracování. Další 3 parametry `threshold`, `gauss` a `form` slouží k nastavení prahování. Konkrétně jestli se jedná o fotografii nebo oskenovanou stránku, velikost rozmazání nebo okolí mediánového filtru a zda vstupní dokument obsahuje přes braillské buňky natištěný význam jednotlivých znaků. Poslední parametr `grid` udává, jestli jsou body uspořádané tak, že v celém obsahu nechybí žádný řádek ani sloupec, tedy že se jedná o pravidelnou mřížku (bývá u většiny skenovaných dokumentů).

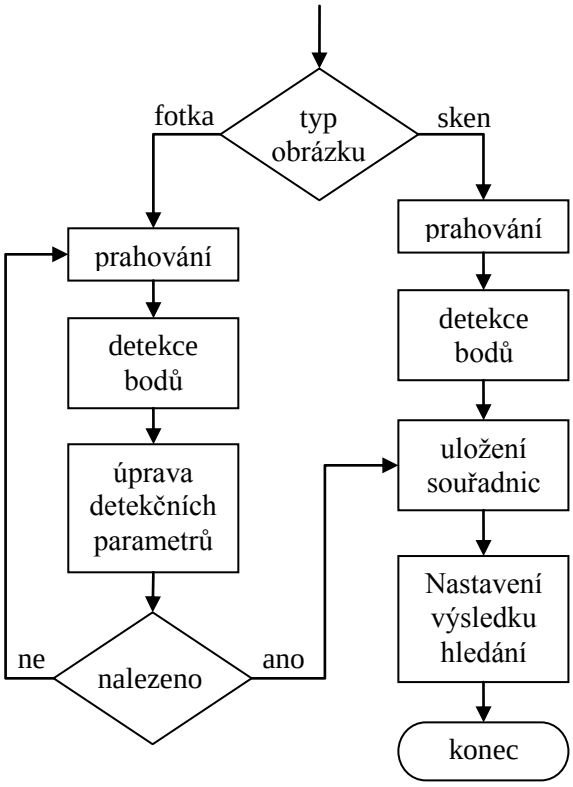
Metoda start

Veřejná metoda `bool start()` implementuje části návrhu podle kapitol 4.1 a 4.2. Její funkce je znázorněna v Algoritmus 5.1. Na začátku je oproti návrhu přidána ještě navíc možnost odstranění tiskopisu. K materiálům s vytištěným překladem jsem se dostal až později, proto nejsou v návrhu uvedeny. Nicméně jejich zpracování je přidáno a řídí se podle parametru `form` zadaného při inicializaci. Pokud je jeho hodnota 0, provede se pouze aplikace gaussovského rozostření na vstupní obraz voláním funkce `cvSmooth(…)` k redukci šumu. Jinak je volána funkce `IplImage *filterImage(IplImage* image, int p1, int p2)`, která odstraní ze vstupního obrázku všechny hodnoty odstínu nacházející se mimo rozsah zadaný parametry `p1` a `p2`.



U většiny dokumentů s výrazným černým písmem, to znamená odfiltrovat nejnižší intenzitu, což představuje hodnoty pixelů zhruba od 0 do 100. Hodnoty 101 – 255 pak poskytují informace důležité pro určení reliéfních bodů, které je třeba zachovat.

Podle typu obrázku, který zvolí uživatel před zahájením operace, rozhodneme, do které větve vstoupíme. Tato hodnota je uložena v parametru `threshold` z inicializační metody. Počáteční úkony jsou pro oba typy dokumentů stejné, jen se provádí s mírně odlišným nastavením. Jako první přichází adaptivní prahování. K tomu využijeme funkci `cvAdaptiveThreshold(…)`, která převede obraz do binární podoby a poté vstupuje do funkce `bool findPoints(IplImage* input, vector<CvRect> &p, int last)`, v níž se obrázek prochází po řádcích a hledá se černý pixel, značící reliéfní bod. Jádro funkce vystihuje následující segment kódu:



Algoritmus 5.1: Detekce reliéfních bodů

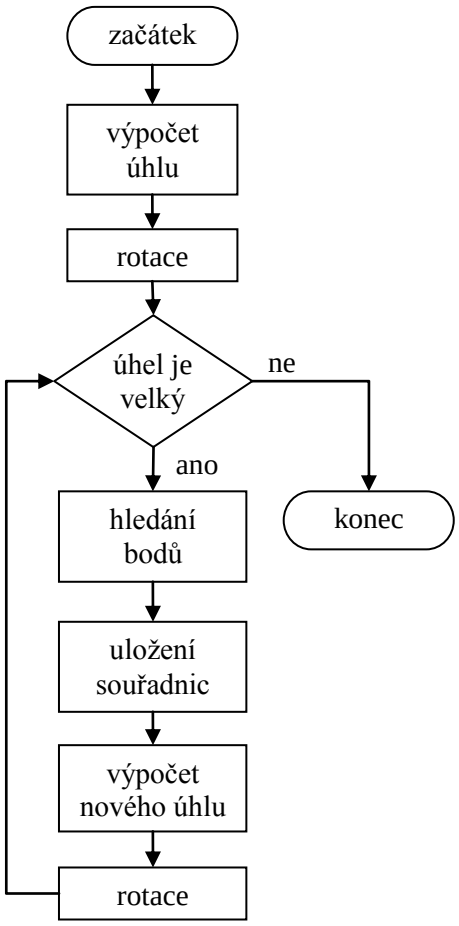
```
for (int y=0; y<height; y++)
   for (int x=0; x<width; x++) {
      // je-li bod cerny, spustime na nej seminkove hledani
      if (input->imageData[y * width + x] == 0) {
         seedFind(x, y, input, mask, thresh, count, max, p);
      }
   }
```

Uvedená funkce `void seedFind(int x, int y, IplImage* input, IplImage* output, int thresh, int &count, CvPoint max, vector<CvRect> &points)` vychází ze standardního semínkového vyplňování se změnami popsanými v návrhu v kapitole 4.2. Paremtry `x` a `y` jsou souřadnice počátečního bodu, `input` je vstupní obrázek, `mask` obsahuje záznam o již prohledaných pixelech, abychom zabránili duplicitnímu vkládání do fronty, `thresh` udává minimum pixelů, které považujeme za reliéfní bod, `count` vrací celkový počet pixelů tvořící bod, `max` označuje nejvyšší indexovatelnou položku v obrázku a do vektoru `points` se ukládají nalezené body.

Uvedený postup je využíván pro oba typy vstupního obrázku. Pro skenovaný dokument se však provádí jen jednou, jelikož jeho kvalita a kontrast jsou poměrně vysoké a je možné přednastavit parametry pro prahování. Při hledání bodů na neznámé fotografii je třeba postupovat iteračně a postupně snižovat práh, dokud nejsou nalezeny oblasti, které se svojí velikostí neliší o více než 100 procent. To je také hlavní podmínka k ukončení cyklu. Poté už následuje shodné zakončení. Středy nalezených bodů uložíme do atributu `vector<CvPoint> pts` a pokud jsou nalezeny alespoň 2, je výsledek funkce `bool start()` nastaven na `true`, v opačném případě na `false`, čímž je metoda pro nalezení reliéfních bodů ukončena.

Metoda equalizePoints

Veřejná metoda `vector<CvPoint> equalizePoints()` se stará o první fázi zarovnání podle kapitoly 4.3. Hlavní úkol je zjistit úhel, o který je rovina braillských řádků odkloněna od osy x a následně o tento úhel obrázek pootočit. Diagram činností je vidět v Algoritmus 5.2. Nejdříve je třeba spočítat úhel odklonu, a pokud bude větší než stanovené minimum, otočit prahovací obrázek. O to se postará funkce `float rotateImage(IplImage* &input, vector<CvPoint> pts, float ratio)`, kde `input` je vstupní obrázek, `pts` seznam nalezených bodů z předchozího kroku a `ratio` udává, o jakou část úhlu se otočení provede. Když se obrázek otáčí postupně, výsledky jsou přesnější než při otočení o celkový úhel, který je určen z počátečních souřadnic.



Algoritmus 5.2: Vyrovnání bodů

V případě náklonu dle Obrázek 4.7 jde o nalezení posledního bodu neklesající posloupnosti, jelikož body se detekují od vrchu dolů, první je vždy v obrázku nejvýš. Od něj pak spustíme hledání sekvence, jak ukazuje následující kód:

```
for (unsigned i=1; i<pts.size(); i++)
{
    if (pts[i].x < pts[i-1].x) {
        minXup = i;
    }
    else break;
}
```

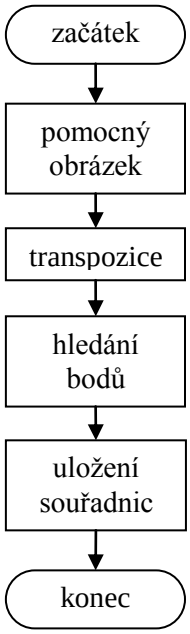
Do proměnné `minXup` uložíme index druhého bodu, ze kterého určíme trojúhelník pro výpočet úhlu. Otočení provedeme v případě, že je úhel větší než nastavený limit 0,3 stupně. Menší se již v obrázku neprojeví a navíc s takto drobnou odchylkou si poradí druhá fáze zarovnání, o které bude řeč ještě později. Vlastní rotace je provedena pomocí standardní funkce OpenCV `warpAffine(…)`. Pokud je tedy úhel dostatečně velký a otočení proběhne, spustíme na upraveném obrázku znovu detekci reliéfních bodů – metoda `start()`. Tím získáme nový seznam souřadnic. Z nalezených bodů je vypočten aktuální úhel odklonu a celý proces se opakuje, dokud není splněna podmínka pro ukončení cyklu, že `abs(angle) < minAngle`. Absolutní hodnota je v podmínce potřeba, jelikož jsme si představili pouze situaci, kdy je odklon směrem nahoru, tedy kladná hodnota úhlu. Analogicky se postupuje, mají-li řádky sklon dolů. Hledání pak probíhá opačným směrem a získaný úhel má tudíž zápornou hodnotu. Po splnění podmínky funkce končí, jejím výstupem je seznam souřadnic `vector<CvPoint>` s novými hodnotami.

Metoda detectColumns

Metodu `vector<CvPoint> detectColumns()` můžeme označit za pomocnou, nicméně její funkce je pro další zpracování velmi důležitá. Při hledání bodů se obrázek prochází po řádcích, jak je

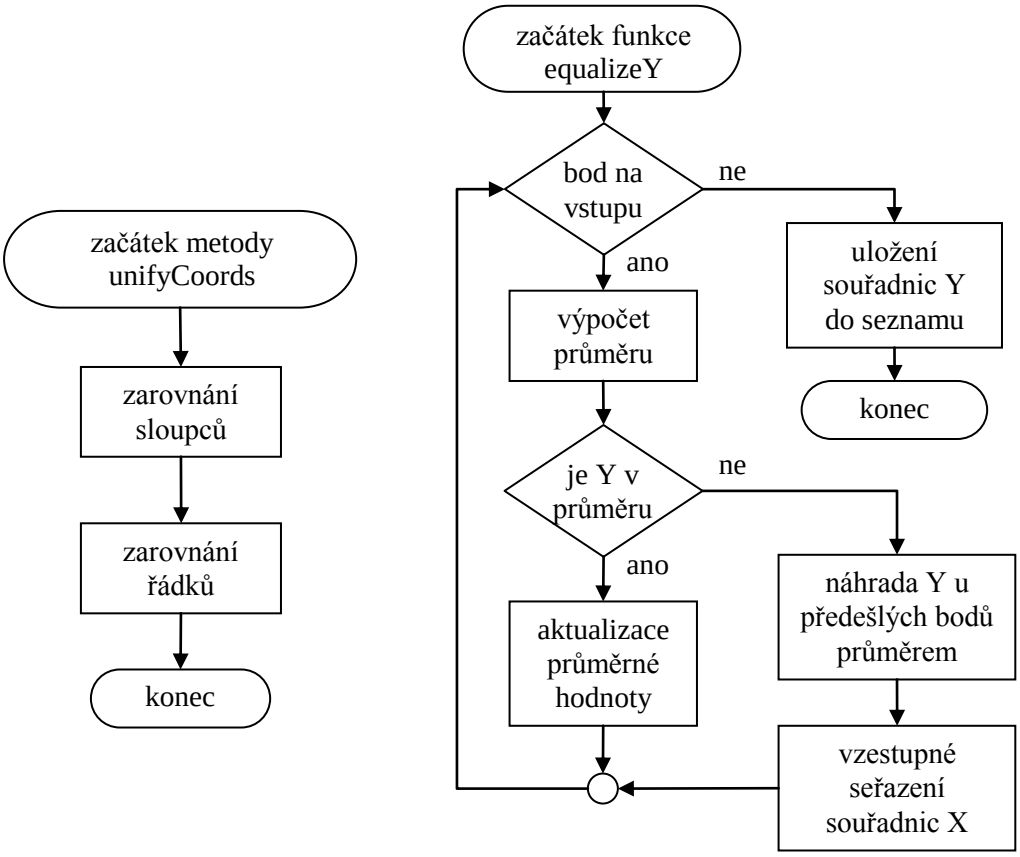
popsáno v metodě `start()`. Takto získaný seznam je možné rozdělit na části reprezentující jednotlivé řádky. Ovšem rozdělení podle sloupců by bylo značně problematické, když jsou souřadnice bodů mírně odlišné. K provedení této akce by bylo dobré mít seznam souřadnic uložený podle sloupců. Nabízí se tak řešení prohledat obrázek znovu, ale tentokrát po sloupcích. Aby se nemuselo zasahovat do stávající metody, která při hledání postupuje po řádcích, provedeme nejprve transpozici původního obrázku. Celý postup je vidět v Algoritmus 5.3.

Pomocný obrázek pro uložení transpozice vytvoříme funkcí `IplImage* cvCreateImage()` s přehozením rozměrů výšky a šířky. Do něj pak pomocí funkce `void cvTranspose(…)` uložíme transponovaný obrázek vzniklý prahováním. Díky tomu již není třeba znovu prahování aplikovat a můžeme ihned zavolat metodu pro detekci bodů `bool findPoints()`. Od tohoto okamžiku už následuje klasický průběh popsaný výše. Nalezené souřadnice jsou uloženy do dalšího atributu třídy `vector<CvPoint> pts2` a tím je funkce ukončena. Oba seznamy pak budou dále využity k dokončení procesu zarovnání.



Algoritmus 5.3: Hledání sloupců

Metoda unifyCoords



Algoritmus 5.4: Zarovnání souřadnic

Na úvodním schématu vlevo (Algoritmus 5.4) je posloupnost činností metody `vector<CvPoint> unifyCoords()`. Jedná se o dva kroky, které má na starost jediná funkce `void equalizeY(vector<CvPoint> &pts, int diametr, vector<int> &columns)`, zobrazená v Algoritmus 5.4 napravo. Parametr `pts` je seznam detekovaných bodů, `diametr` udává průměrnou velikost reliéfního bodu v pixelech. Přes poslední parametr `columns` je vrácen seznam s nově určenými souřadnicemi sloupců (řádků). Poprvé je volána se seznamem `pts2`, kde jsou uloženy body po sloupcích. Je třeba si uvědomit, že kvůli předchozí transpozici se z původních sloupců stanou řádky a naopak. Díky tomu je opět možné použít tu samou funkci pro zpracování obou seznamů. V jednom případě se skutečně zarovnává souřadnice Y, ve druhém však X. Na první pohled vypadá, že děláme dvakrát totéž a může být matoucí, že se proměnná v obou případech jmenuje `columns`. Jednou se však zpracovávají sloupce, podruhé řádky.

Hlavní smyčka funkce je tvořena cyklem `for` přes všechny body na vstupu. Jednotlivé body jsou porovnány s průměrnou hodnotou a podle výsledku rozhodneme o dalším postupu. Tuto činnost provádí následující úsek kódu:

```
int avgY = pts[0].y;
int countY = 1;

for (unsigned i=1; i<pts.size(); i++) {
     if (pts[i].y < avgY + diametr) {
          avgY = avgY + pts[i].y;
          countY++;
          avgY = avgY / 2;
     }
```

Proměnná `avgY` slouží k uchování průměrné hodnoty. Počáteční inicializace je provedena prvním bodem na vstupu. Proměnná `countY` je počítadlo bodů, které splní podmínku, že jeho souřadnice Y je menší než průměrná hodnota zvýšená o velikost jednoho bodu. Tím vzniká přiměřená tolerance pro správné určení, zda bod spadá do aktuálního řádku nebo je již mimo. Pokud ano, je hodnota zařazena do průměru a zvýšeno počítadlo. Jestli podmínku nesplňuje, pokračuje se následujícím kódem:

```
     else {
          for (int j=countY; j>0; j--) {
               // nahrada stare hodnoty
               pts[i-j].y = avgY;
          }
          bubble(pts, i-countY, i);
          // nasteveni pocatecnich hodnot pro dalsi radek
          avgY = pts[i].y;
          countY = 1;
     }
} // cyklus for
```

Když je na vstupu bod z dalšího řádku, je třeba provést aktualizaci souřadnice Y u předchozích bodů tvořících řádek. O náhradu se postará další cyklus `for`. Poté už jen body seřadíme podle souřadnice X od nejmenší po největší, aby byl řádek v takovém pořadí, v jakém je na fotografii. O setřídění se postará funkce `void bubble(vector<CvPoint> &p, unsigned x1, unsigned x2)`, implementující klasickou metodu bubble sort. Tato varianta je pro daný účel velice vhodná, jelikož body jsou již z velké části seřazeny. Parametry `x1` a `x2` udávají počáteční a koncový index bodů v seznamu, kterých se operace týká. Po skončení cyklu tak máme ve vektoru `pts` seřazené sekvence

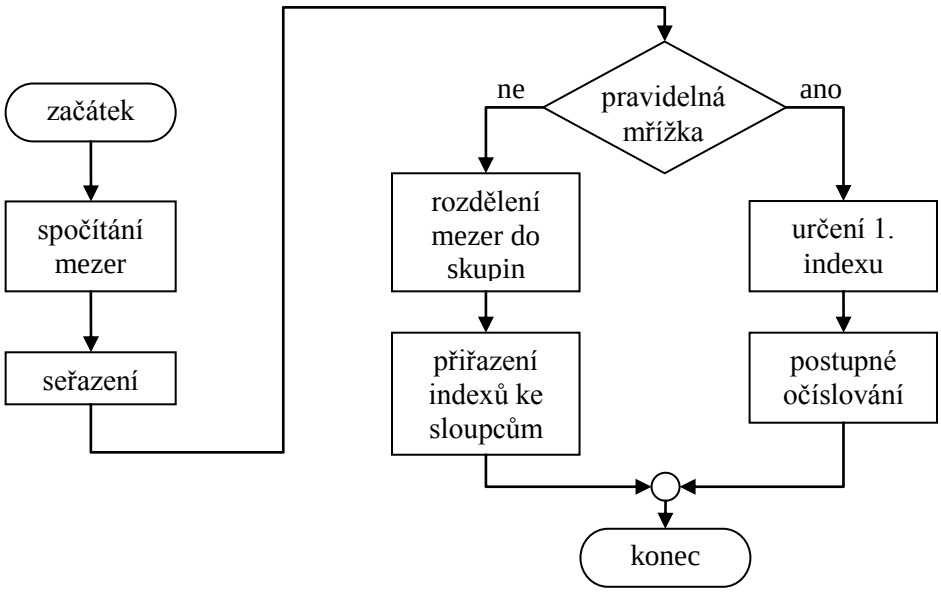
bodů, které odpovídají jednotlivým řádkům v obrázku. Pro potřeby dalšího zpracování ještě zbývá naplnit seznam `sloupce` aktuálními hodnotami souřadnic Y:

```
for (unsigned i=0; i<pts.size(); i++) {
    // detekce zmeny Y souradnice = novy radek
    if ((i<pts.size()-1) && (pts[i].y != pts[i+1].y)) {
        sloupce.push_back(pts[i].y);
    }
}
```

Jelikož máme body v řádku srovnané na stejnou souřadnici, je možné k určení dalšího řádku využít změnu souřadnice Y, kterou následně uložíme do výstupního vektoru `sloupce`. Touto akcí je funkce `equalizeY(…)` ukončena. Pro ukončení celé metody `unifyCoords()` však musí proběhnout ještě jednou pro zarovnání řádků, tedy s parametrem `pts`.

Metoda calcSpaces

Metoda `void calcSpaces()` provádí indexaci souřadnic podle návrhu v kapitole 4.4. Její zjednodušené schéma zobrazuje Algoritmus 5.5.



Algoritmus 5.5: Výpočet mezer a indexování

Na vstup přichází seznam sloupců, vytvořený předchozí metodou, nazvaný `unifiedColumns`.
Ten je v cyklu zpracován a spočítané mezery jsou uloženy do nového seznamu, jak ukazuje následující zápis:

```
vector<CvPoint> spaces;

// pocitani rozdilu
for (unsigned i=1; i<unifiedColumns.size(); i++) {
    space = unifiedColumns[i] - unifiedColumns[i-1];
    spaces.push_back(cvPoint(space, 0));
}

bubble(spaces, 0, spaces.size());
```

Průchodem přes všechny sloupce počítáme rozdíly sousedních souřadnic a výsledky průběžně ukládáme do vektoru `spaces`. Ten je typu `CvPoint`, i když je použita jen jedna položka. Důvodem je, abychom mohli zavolat řadící funkci `bubble(…)`, která byla určena pro předchozí případ, kde bylo nutné seřadit právě typ `CvPoint`, jinak by samozřejmě stačil vektor typu `int`. Po seřazení o dalším postupu rozhodneme podle zvoleného typu zpracování zadaného uživatelem. Konkrétně se jedná o parametr `grid` z metody `init(…)`, probrané na začátku kapitoly.

Tvoří-li body pravidelnou mřížku, tedy nechybí žádný sloupec, je možné jednotlivé souřadnice očíslovat postupně. Jen je třeba určit, zda bude indexování začínat 0 nebo 1. To provedeme porovnáním první mezery se získaným seznamem. Proto bylo třeba jej počítat i v tomto případě. Algoritmus pracuje následovně:

```
space = unifiedColumns[1] - unifiedColumns[0];

for (unsigned i=0; i<spaces.size(); i++) {
     if (space == spaces[i].x) {
          order = i;
          break;
     }
}
```

Do proměnné `space` nejdříve uložíme velikost první mezery. Poté hodnotu postupně porovnáme s hodnotami v seznamu. Až nalezneme shodu, uložíme si index mezery do proměnné `order` a cyklus končí. Získanou hodnotu pak využijeme k rozhodnutí, kterým indexem začne číslování sloupců. Víme, že seznam mezer je vzestupně uspořádaný, proto v první polovině jsou krátké mezery odpovídající vzdálenosti mezi reliéfními body v buňce. Ve druhé polovině se vyskytují vzdálenosti větší, tzn. mezi buňkami. Rozhodnutí o počátečním indexu vyjadřuje tento kód:

```
int index = 0;
if (order > spaces.size()/2) index = 1;
```

Jednoduše nastavíme index na nulu. Pokud však počáteční mezera spadá až do druhé poloviny seznamu, znamená to, že se jedná o dlouhou mezeru oddělující buňky a tedy, že první bod je ve druhém sloupci. Proto změníme index na jedničku. Poté už se indexy až do konce pravidelně střídají. S každou změnou na 0 také zvýšíme počítadlo znaků, které označuje skupinu dvou sloupců, jak je uvedeno v návrhu.

```
vector<CvRect> indexedCols;
order = 0;

for (unsigned i=0; i<unifiedColumns.size(); i++) {
     indexedCols.push_back(cvRect(unifiedColumns[i], order,
     index, 0));

     if (index == 0) {
          index = 1;
     }
     else {
          index = 0;
          order++;
     }
}
```

Souřadnice sloupců spolu s indexy ukládáme do vektoru `indexedCols`. Jelikož potřebujeme uchovat 3 hodnoty, použijeme standardní typ `CvRect`, který dovoluje uložit hodnoty 4. Poslední položka tedy zůstane nevyužitá a na její místo budeme pokaždé vkládat nulu. Proměnnou `order`

využijeme znovu, tentokrát jako počítadlo znaků, proto počáteční inicializace na nulu. Dále už následuje cyklus přes všechny sloupce, ve kterém probíhá ukládání všech potřebných hodnot do vektoru v tomto pořadí: souřadnice sloupce, pořadí znaku a index sloupce. Po každém uložení se nakonec aktualizují indexy pro příští průchod. Tímto cyklem je větev algoritmu označená jako pravidelná mřížka ukončena. Nyní si popíšeme průchod druhou větví, tedy možnost, že některé sloupce v obrázku chybí a je třeba určit indexy podle okolních bodů.

V tomto případě je nutné nejprve získaný seznam s hodnotami mezer rozdělit do skupin, podle kterých budeme později rozhodovat, jaký index přidělit. Rozdělení se řídí tímto kódem:

```
vector<CvPoint> spaceIndex;
spaceIndex.push_back(cvPoint(0,0));
int group = 0;

for (unsigned i=0; i<spaces.size()-1; i++) {
    if (spaces[i+1].x - spaces[i].x > spaces[i+1].x * 0.15) {
        spaces[i].y = group;
        spaceIndex[group].y = i;
        spaceIndex.push_back(cvPoint(i+1,0));
        group++;
    }
    else {
        spaces[i].y = group;
        spaceIndex[group].y = i;
    }
}
```

Vektor `spaceIndex` slouží k uložení počátečního a koncového indexu jednotlivých skupin. Ihned na začátku je do něj vložena inicializační hodnota, prozatím dvě 0. První je v pořádku, indexování samozřejmě nulou začíná. Koncový index bude v průběhu výpočtu upraven, jakmile bude znám. Další proměnná `group`, je index aktuální skupiny mezer, dala by se označit i za počítadlo. Poté už spustíme cyklus přes všechny mezery. Postupně porovnáváme, jestli se sousední hodnoty neliší o více než 15 procent. Pokud ano, následující mezera už patří do další skupiny. Označíme aktuální mezeru současnou skupinou a do proměnné `spaceIndex` vložíme index ukončující skupinu. Zároveň přidáme i druhý index otvírající novou skupinu o jedničku vyšší. Zmíněná hodnota 15 procent je hranice, která odděluje jednotlivé skupiny. Tato hodnota vznikla na základě experimentování a dávala nejlepší výsledky. Pokud podmínka není splněna, jedná se tedy pořád o stejnou skupinu a pouze přiřadíme odpovídající značky do proměnných. Po ukončení cyklu tak máme k dispozici seznam `spaces`, kde ve složce `x` je uložena hodnota mezery a ve složce `y`, do které skupiny daná mezera patří. 0 označuje nejmenší mezery, tedy takové, které oddělují jednotlivé reliéfní body v buňce. 1 patří mezerám o něco větším, které od sebe oddělují jednotlivé buňky. Tohle jsou dvě základní skupiny, které se musí vyskytovat vždy. Potom v závislosti na obrázku přidáváme ještě další, které označují větší mezery. To jsou případy, kdy chybí některý bod nebo celá buňka, popřípadě i několik bodů za sebou. Tyto skupiny se dál číslují 2, 3, 4, atd. Dokud se hodnoty liší o více než uvedených 15 procent.

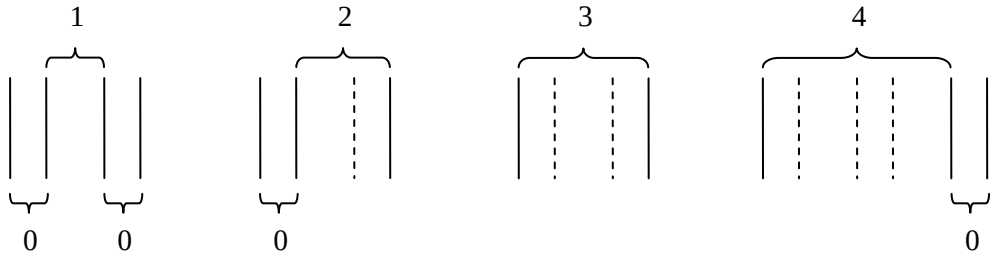
Posledním úkolem metody je označení jednotlivých souřadnic indexem znaku a sloupce. V předchozím případě bylo možné indexy pravidelně střídat, jelikož žádná souřadnice nechyběla. U fotografií, kde se vyskytují různé nepravidelnosti, je však pravděpodobné, že indexy budou několikrát za sebou stejné z důvodu chybějícího sloupce. Proto je třeba každou souřadnici ohodnotit na základě

porovnání mezery předcházejícího a následujícího sloupce. Tím získáme přehled o tom, na které pozici se aktuální sloupec nachází. Tento fragment kódu ukazuje porovnávání předcházejících mezer:

```
vector<CvRect> indexedCols;
int index = 0;
int diff;

for (unsigned i=0; i<unifiedColumns.size()-1; i++) {
  if (i>0) {
    diff = unifiedColumns[i] - unifiedColumns[i-1];
    for (unsigned j=0; j<spaceIndex.size(); j++) {
      // porovnani vzdalenosti
      if (diff >= spaces[spaceIndex[j].x].x && diff <=
      spaces[spaceIndex[j].y].x) {
        if (j == 0) index = 1;
        else
          if (j == 1) index = 0;
          else
            if (j == 2) index = indexedCols[indexedCols.size()-1
                                ].width;
            else
              if (j == 3) index = !indexedCols[indexedCols.size()-2
                                  ].width;
              ...
        break;
  } } }
  ...
  indexedCols.push_back(cvRect(unifiedColumns[i], order, index,
  0));
}
```

Proměnné `index` a `indexedCols` plní stejnou funkci, jako v případě popsaném dříve, `diff` udává velikost mezery mezi aktuálním a předchozím sloupcem. V cyklu postupně procházíme jednotlivé souřadnice, kde rozdílem dvou po sobě jdoucích sloupců získáme mezeru mezi nimi. Tu pomocí dalšího cyklu hledáme v seznamu mezer. V případě shody, udává řídící proměnná `j` skupinu, do které daná mezera patří. Podle ní pak pomocí `if – else` konstrukcí rozhodneme, jaký index přidělit. Významy jednotlivých skupin byly již částečně popsány. Pro lepší pochopení jsou ještě vyznačeny na Obrázek 5.1. Plné čáry reprezentují existující sloupce, čárkované chybějící.



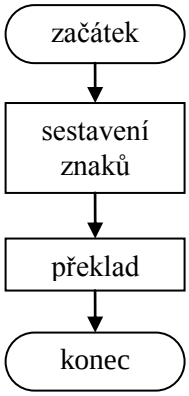
Obrázek 5.1: Rozdělení skupin u indexů

Jak víme, indexy 0 a 1 udávají, o který sloupec v braillské buňce se jedná. Nyní se podívejme na význam jednotlivých přiřazení v uvedeném kódu. Porovnáváme vždy mezeru mezi aktuálním sloupcem a bezprostředně předcházejícím. Pokud tedy daná mezera patří do skupiny 0 (odpovídá podmínce j == 0), znamená to, že předchozí bod je vzdálen o mezeru oddělující body v buňce. Aktuální sloupec je tedy až druhý a tak nastavíme index na jedničku (číslováno od nuly). V dalším porovnání se jedná o skupinu 1, což značí, že předchozí bod je vzdálen na mezeru mezi buňkami. Aktuální souřadnice tedy představuje první sloupec buňky, proto nastavíme index na 0. Ve skupině 2 už se jedná o chybějící souřadnici. Znamená to tedy, že předchůdce je vzdálen ob jeden sloupec a hodnotu indexu natavíme právě tak, jak byla zvolena u něj. Jestliže mezera spadá do skupiny 3, chybějí mezi aktuálním bodem a předchozím sloupce 2. V takovém případě nastavíme index na opačnou hodnotu, než je poslední určený index. Zpracování dalších skupin probíhá analogicky, uvedená část kódu je pouze pro ilustraci. Mimo porovnávání předchozích sloupců probíhá ještě druhá smyčka, která porovnává následníky. Pokud jsou mezery mezi souřadnicemi větší je třeba někdy oba způsoby kombinovat. Kompletní zápis metody je možné nalézt ve zdrojovém kódu, který je součástí přílohy.

Na konci smyčky opět ukládáme oba zjištěné indexy, tedy sloupce a znaku, do vektoru `indexedCols`. Získáme tedy totožný formát výstupu jako v případě první větve. Popsaná metoda patří k nejsložitějším v celém průběhu zpracování obrázku. Na základě očíslování jednotlivých souřadnic je však už poměrně snadné sestavit braillské buňky, které jsou pro převod nezbytné.

Metoda translate

Poslední ze série metod pro zpracování vstupu je `string translate()`. Jak je vidět z definice, návratová hodnota již není žádný seznam se souřadnicemi, ale hotový překlad v textové podobě. Celé předcházející zpracování je touto metodou zakončeno. Posloupnost její činnosti je vidět v Algoritmus 5.6. Nejprve z vytvořených seznamů s indexy sloupců a řádků sestavíme jednotlivé braillské buňky, které mají podobu šesti binárních hodnot, jak je popsáno v kapitole 4.4. Znaky budou ukládány do vektoru `chars` s položkami `Tchar`:



Algoritmus 5.6: Překlad

```
typedef struct {
    char code[7];
    byte value;
} Tchar;
```

Proměnná `code` uchovává binární podobu, proměnná `value` udává číselnou hodnotu, kterou reprezentuje kód ve dvojkové soustavě. Sestavení znaků probíhá následovně:

```
int indexX = 0;
int indexY = 0;
int indexZ = 0;

for (unsigned i=0; i<pts.size(); i++) {

    for (unsigned j=0; j<indexedCols.size(); j++) {
        if (pts[i].x == indexedCols[j].x) {
            indexY = indexedCols[j].y;
            indexZ = indexedCols[j].width * 3;
            break;
```

```
            }
        }
        for (unsigned j=0; j<indexedRows.size(); j++) {
            if (pts[i].y == indexedRows[j].x) {
                indexX = indexedRows[j].y;
                indexZ = indexZ + indexedRows[j].height;
                break;
            }
        }
        chars[indexX][indexY].code[indexZ] = '1';
    }
```

Indexy X a Y jsou použity k identifikaci buňky v obrázku. Index Z udává pozici bodu, který se bude aktualizovat – měnit na jedničku. Všech šest pozic v položce `code` je zpočátku přednastaveno na nulu. V cyklu jsou postupně zpracovány jednotlivé body. Jeho souřadnice X je porovnána se seznamem sloupců. Po nalezení shody si uložíme odpovídající indexy. Totéž provedeme i pro souřadnici Y, kterou porovnáváme se seznamem řádků. Nakonec jsou všechny získané indexy využity k určení absolutní pozice ve znaku, který se bude aktualizovat. Po zpracování všech bodů získáme seznam s posloupností šesti binárních hodnot reprezentujících textový zápis Braillova písma. Ten už stačí poslat do překladového automatu a celý proces dokončit. K tomu využijeme další datové typy. `Tprefixes` pro identifikaci prefixů a `Talphabet` s významem jednotlivých symbolů po aplikaci prefixů.

```
typedef enum {
    NONE,
    CAPITAL_LETTER,
    GROUP_CAPITALS,
    LETTER,
    NUMBER,
    CAPITAL_GREEK,
    GREEK_LETTER
} Tprefixes;
```

```
typedef struct {
    char letter[3];
    char capital[3];
    char number[3];
    Tprefixes prefix;
} Talphabet;
```

Struktura překladového automatu je následující:

```
vector<Talphabet> ab;
Tprefixes prefix;
prefix = ab[bin8(chars[0][0].code)].prefix;
for (unsigned i=0; i<chars.size(); i++) {
    for (unsigned j=0; j<chars[0].size(); j++) {
        switch (prefix) {
            case NONE: ...
            case CAPITAL_LETTER: ...
            case GROUP_CAPITALS: ...
            case LETTER: ...
            case NUMBER: ...
            ...
        }
    }
}
```

Vektor `ab` obsahuje význam jednotlivých symbolů. Ten je v programu nastaven na českou abecedu, je však možné ho změnit na novou definici uloženou v XML souboru (viz příloha XML soubor s definicí abecedy). Každý znak je přístupný přes číslo, které udává jeho binární podoba. Proto je do hranatých závorek indexace vložena funkce `byte bin8(const char *code)`, která převádí právě hodnotu z dvojkové soustavy do dekadické. Proměnná `prefix` slouží k uložení prefixu aktuálního znaku, podle kterého se provádí rozhodnutí o dalším postupu. Jak vidíme je ihned nastaven na hodnotu prvního znaku ze vstupu. Poté je spuštěn cyklus, kde porovnáváme prefix každého symbolu a v příslušné větvi přidáme do výstupního řetězce odpovídající znak z abecedy. Pro ilustraci se podívejme na konkrétní zpracování čísla, ostatní varianty pracují analogicky.

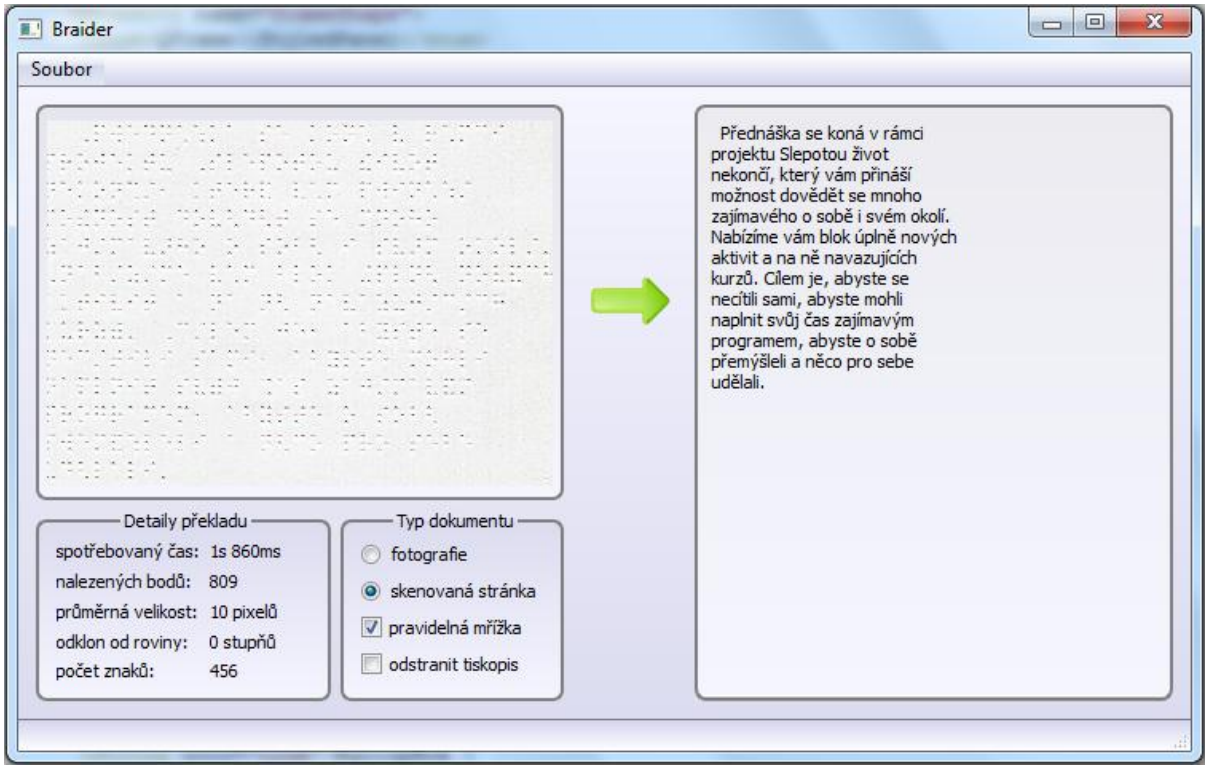
```
case NUMBER:
  if (ab[chars[i][j].value].prefix == NONE) {
    if (strcmp(ab[chars[i][j].value].number, "N") != 0) {
      sprintf(output, "%s", ab[chars[i][j].value].number);
    }
    else {
      sprintf(output, "%s", ab[chars[i][j].value].letter);
      prefix = NONE;
    }
    trans.append(output);
  }
  else prefix = ab[chars[i][j].value].prefix;
  break;
```

Nejprve zkontrolujeme, zda symbol ze vstupu je nositel nějaké hodnoty, nebo se jedná o prefix. V tom případě jej nastavíme do proměnné `prefix`, podle kterého bude zpracování následujícího znaku řízeno. Pokud symbol žádný prefix neobsahuje, je to tedy konečný znak a zbývá jen určit jeho význam. Protože se jedná o zpracování čísel, na začátku porovnáme, zda skutečně o číselnou hodnotu jde. Jako číslice jsou využity pouze písmena A-J. Srovnání na `"N"` ve funkci `strcmp` udává, že je číselná hodnota nedefinovaná. Pak už zbývá jen do proměnné `output` nastavit znak, který přidáme do výstupního řetězce a větev je ukončena. Po proběhnutí hlavního cyklu tak máme v atributu `string trans` kompletní přeložený obsah Braillova písma z obrázku.

Touto akcí končí celé zpracování, jehož průběh jsme si zde postupně přiblížili. Třída `Cbralile` obsahuje ještě další pomocné metody, které jsou použity při výpočtu. Většinou se však jedná o elementární úkoly, jejichž význam není natolik zásadní, abychom se jimi podrobně zabývali. Veškeré detaily je možné nalézt v příslušném zdrojovém kódu definice třídy, kde jsou všechny dostupné funkce popsány.

5.2 Třída MainWindow

Třída `MainWindow` je vytvořena s využitím prostředků Qt pro tvorbu uživatelského rozhraní. Je odvozena od základní třídy `QMainWindow`, do které přidává navíc další atributy a jednotlivé grafické prvky definující vzhled aplikace. Jsou to zejména `QLabel` k zobrazení vybraného obrázku a `QPlainTextEdit`, do kterého je vypsán výsledný překlad. Pro účely nastavení slouží `QGroupBox` s položkami `QRadioButton` a `QCheckBox`. Dále pak je to `QMenuBar` nabízející volbu uživatelských akcí a stavový řádek `QStatusBar`, informující o aktuální situaci. Podobu hlavního okna aplikace je možné vidět na následujícím obrázku (Obrázek 5.2).



Obrázek 5.2: Hlavní okno aplikace

Mezi klíčové atributy třídy patří `QString trans`, do kterého je uložen výsledek překladu. `IplImage *img` slouží pro načtení vstupního obrázku. Dále instance třídy `Cbraille *cb`, pomocí níž je obrázek voláním jednotlivých metod zpracován. Poslední zmíněnou položkou je `BrailleThread *th`. Jedná se o výpočetní vlákno, ve kterém probíhá překlad. Jelikož zpracování trvá řádově sekundy, bylo nutné jej zavést, jelikož by aplikace během výpočtu nereagovala na žádné podněty. Z důvodu neaktivní smyčky zpráv by tak zůstala po celou dobu „zamrzlá“. O třídě `BrailleThread` si povíme více později.

Co se týče implementovaných metod, nejdůležitější je událost na stisknutí zelené šipky (uprostřed), která zahájí překlad - `void on_button_trans_clicked()`. Její tělo je přibližně takovéto:

```
if (!img)
    label_image->setText(QString::fromUtf8("Obrázek se nepodařilo
    otevřít!"));
else {
    ui->text_translation->clear();
    ...
    if (ui->rb_photo->isChecked()) threshType = 0;
    if (ui->rb_scan->isChecked()) threshType = 1;
    ...
    cb->init(img, threshType, gauss, grid, form);
    th->init(cb);
    th->start();
    ...
}
```

Nejdříve proběhne test, zda je dispozici platný obrázek. Pokud ne, upozorníme uživatele zobrazením krátké zprávy. Dále už probíhají jednotlivé kroky algoritmu. Smažeme předchozí text z pole pro

překlad `text_translation`. Podle zvolených radiových tlačítek nastavíme typ prahování – 0 pro zpracování fotografie a 1 pro skenovaný dokument. Následuje zavolání metody `init(…)`, pomocí které dojde k inicializaci všech potřebných atributů třídy `Cbraille`. Definovanou instanci `cb` předáme výpočetnímu vláknu `th` a pak jej spustíme. Díky tomu probíhá zpracování samostatně a aplikace zůstává po celou dobu aktivní.

O ukončení překladu jsme informováni díky signálu `StateChange(int)`, který je v průběhu výpočtu postupně generován se zvyšujícím parametrem. Jako reakce na zachycený signál je vyvolána funkce `void on_StateChange(int state)`, kde je pro každý stav připravena adekvátní odpověď. Ve většině případů se jedná o výpis zprávy do stavového řádku nebo právě o zobrazení přeloženého textu. Zde je malá ukázka příslušného kódu:

```
switch (state) {
    case 0:
        ui->statusBar->showMessage(...);
        break;
    case 1:
        ui->statusBar->showMessage(...);
        break;
    ...
    case 6: {
        trans = cb->getTranslation().c_str();
        ui->text_translation->setPlainText(
            QString::fromUtf8(trans.toStdString().c_str()));
    ...
}
```

Stavy 0-5 způsobí aktualizaci stavového řádku aktuální fází zpracování. Stav 6 je indikace, že byl překlad dokončen. V tomhle případě provedeme zobrazení přeloženého textu. Metoda `string getTranslation()` dává na výstup obsah proměnné `trans` ze třídy `Cbraille`, kde je uložen výsledek. Ten pak přeformátujeme a zobrazíme v textovém poli, jak je vidět na posledním řádku. Tím je celá operace dokončena a aplikace čeká na další uživatelskou akci.

5.3 Třída BrailleThread

O třídě `BrailleThread` jsme si řekli, že zajistí provedení překladu. Je odvozena od základní třídy `QThread`, kde je přetížena metod `void run()`. Právě v ní je uveden kód, který se má v novém vlákně provést. V našem případě jde o celý výpočet od detekce bodů po provedení překladu. Díky navržené třídě `Cbraille`, je tak možné snadno celé zpracování do nového vlákna přenést. Stačí jen předat konkrétní instanci a poté volat jednotlivé metody ve správném pořadí. Tělo funkce `run()` vypadá následovně:

```
emit StateChange(0);            // detekce reliéfních bodů
if (cb->start()) {

    emit StateChange(1);        // otočení obrázku
    cb->equalizePoints();

    emit StateChange(2);        // průchod transponovaným obrazem
    cb->detectColumns();
```

```
        emit StateChange(3);        // sjednocení řádků a sloupců
        cb->unifyCoords();

        emit StateChange(4);        // určení mezer a indexů
        cb->calcSpaces();

        emit StateChange(5);        // probíhá překlad
        trans = cb->translate().c_str();

        emit StateChange(6);        // konec výpočtu
    }
    else
        emit StateChange(7);        // body nenalezeny
```

Jak vidíme, s každým voláním výpočetní metody je také emitován signál `StateChange(int)`, který informuje aplikaci o průběhu výpočtu. Reakci na signál jsme si již popsali. Uživatel tak může sledovat postup zpracování, což přináší jistý přehled a jistotu, že překlad skutečně běží.

6 Testování

Nyní se zaměříme na otestování aplikace. Na vstupu postupně vystřídáme různé materiály a budeme zaznamenávat výsledky. Zejména kolik bodů se nepodařilo správně určit, abychom mohli spočítat přibližnou úspěšnost detekce. Testování rozdělíme do tří skupin podle obsahu. První skupinu představují fotografie s menším množstvím bodů a rozdílnou kvalitou. Druhá skupina obsahuje předlohy s velkým množstvím bodů, které byly pořízeny skenováním stránek z literatury pro nevidomé. V poslední skupině jsou zařazeny materiály obsahující přes body vytištěný význam jednotlivých znaků.

6.1 Skupina 1

Zde budeme sledovat úspěšnost při hledání bodů. Vstupní fotografie nemají srovnatelnou kvalitu, liší se barvou pozadí i velikostí bodů (viz Obrázky B.1 a B.2). V některých případech nejsou ani rovnoměrně osvětleny. Rozmanitosti je dost a právě u tohoto testu se přesvědčíme, jak univerzální detektor jsme vytvořili.

soubor	nalezeno bodů	chybně určeno	velikost bodu	úspěšnost
foto1.jpg	16	0	18 pixelů	100 %
foto2.jpg	13	0	52 pixelů	100 %
foto3.jpg	58	1	18 pixelů	98,3 %
foto4.jpg	248	0	3 pixely	100 %
foto5.jpg	139	2	4 pixely	98,6 %
foto6.jpg	465	1	7 pixelů	99,8 %

Jak vidíme, průměrná úspěšnost detekce se pohybuje kolem 99 procent. V některých případech se nepodařilo správně určit reliéfní bod z důvodu slabého kontrastu, způsobeného pravděpodobně mechanickou deformací. Pokud jsou však body nepoškozené, rozpoznání probíhá velice přesně pro rozdílné fotografie s různě velkými body. Vzhledem k původnímu zaměření aplikace právě na detekci bodů v předem neznámém vstupu, je výsledek tohoto testu velice příznivý.

6.2 Skupina 2

V této skupině jsou oskenované celé stránky nebo části stránek z knih či jiných materiálů psaných Braillovým písmem (Obrázek B.3). Jejich kvalita je poměrně vysoká a obsahují velké množství reliéfních bodů.

soubor	nalezeno bodů	potřebný čas	velikost bodu	překlad
sken1.jpg	809	1,83 s	10 pixelů	100 %
sken2.jpg	545	1,42 s	10 pixelů	100 %
sken3.jpg	646	1,44 s	10 pixelů	100 %
sken4.jpg	1231	2,86 s	10 pixelů	100 %
sken5.jpg	1045	2,37 s	10 pixelů	100 %

Díky přesnému skenování nebyl problém se správnou detekcí reliéfních bodů. Nastavení skeneru bylo pro všechny případy shodné, proto je také velikost jednotlivých bodů vždy 10 pixelů a úspěšnost překladu stoprocentní. U tohoto testu byla spíše sledována doba spotřebovaná výpočtem. Jak vidíme, zpracovat plně využitou stránka A4 trvá přibližně 3 sekundy, což je poměrně dost. Nicméně vysoká přesnost tento nedostatek značně kompenzuje. Skenované dokumenty s dostatečnou kvalitou je tedy možné bez problému zpracovávat.

6.3 Skupina 3

Nyní zkusíme několik testů s potištěnými dokumenty (Obrázek B.4). Aplikace nebyla původně navržena pro zpracování tohoto typu materiálu, ale jistá podpora je díky jednoduchému filtru k dispozici. Dochází však k jistému oslabení klíčových znaků a proto není výsledek překladu tolik přesný jako v předchozích případech.

soubor	nalezeno bodů	neurčeno	velikost bodu	úspěšnost
tisk1.jpg	627	51	8 pixelů	92,5 %
tisk2.jpg	439	5	9 pixelů	98,9 %
tisk3.jpg	333	2	9 pixelů	99,4 %
tisk4.jpg	442	8	9 pixelů	98,2 %

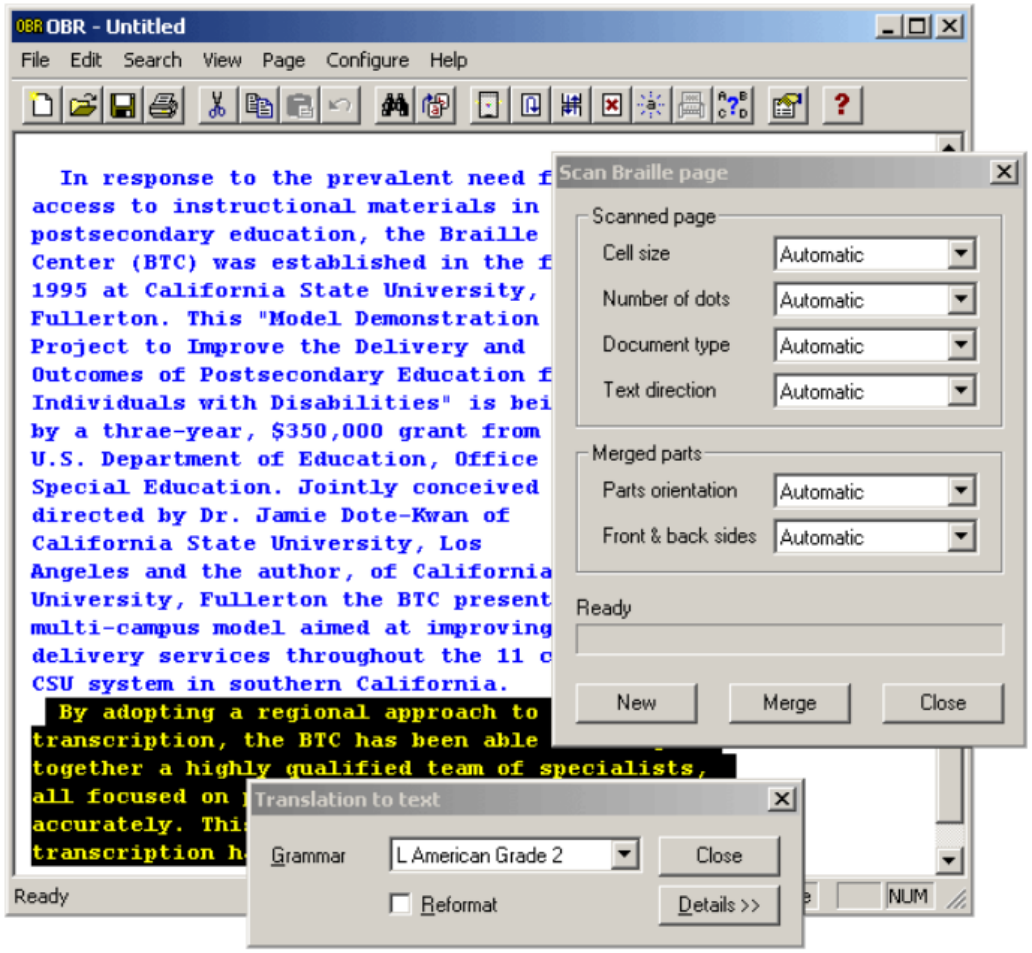
Průměrná úspěšnost rozpoznání bodů činí asi 97 procent. Na první pohled se nemusí zdát výsledek tak špatný, ale pokud chybí více než 30 bodů, znamená to zhruba i 30 špatně přeložených znaků. Pokud se vyskytne v jednom slovu více těchto překlepů, stává se nečitelné a kratší text pak nemusí dávat smysl. Na druhou stranu méně než 10 chyb v celém obsahu většinou neznamená výraznější komplikaci a text bývá poměrně dobře čitelný a s minimálním úsilím opravitelný. Můžeme tedy říct, že i tento typ předlohy je ve většině případů možné aplikací zpracovat.

7 Porovnání aplikací

Když známe přibližné kvality našeho programu, můžeme se podívat na existující software pro rozpoznávání Braillova písma a provést malé srovnání. Produktů zaměřených právě na tuto oblast zpracování písma není mnoho, nabízejí však velice přesné výsledky. Jako prvního a zároveň nejvýznamnějšího zástupce bych uvedl aplikaci OBR [11] od české firmy Neovision s.r.o. a druhý příklad z dílny ruské laboratoře ElecGeste a jejich software Braille Reader VDL [12].

7.1 OBR

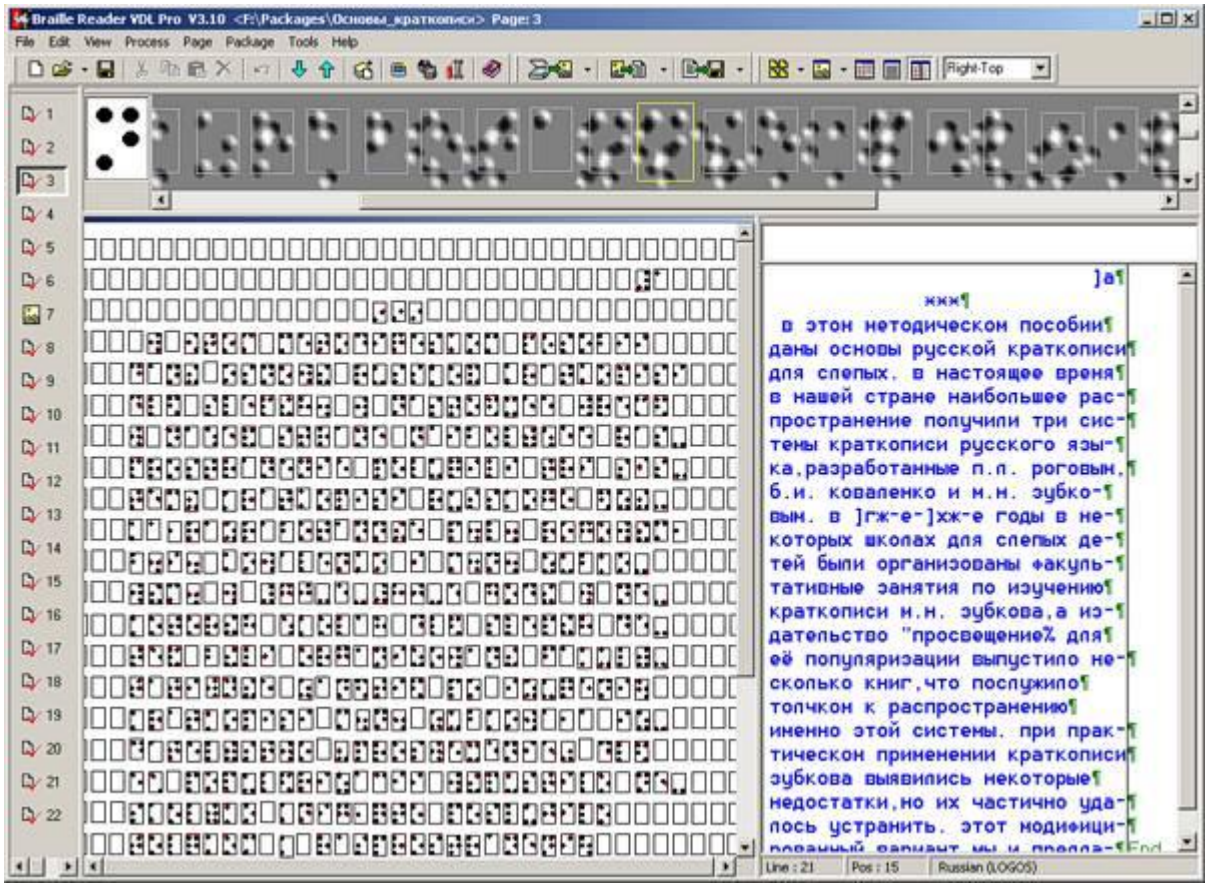
OBR neboli Optical Braille Recognition je první software na světě k rozpoznávání Braillovo písma s pomocí obyčejného skeneru připojeného k počítači. Zvládá čtení jednostranných i oboustranných předloh. Kvalita rozpoznávání je pro nepoškozené Braillské dokumenty vynikající. Dokonce i pro starší a poškozené tiskoviny je chybovost velmi nízká. Výrobce uvádí průměrnou úspěšnost rozpoznání 99,98 procent, což je skutečně mimořádná hodnota vzhledem k faktu, že software dovoluje zpracovat i oboustranné dokumenty. Vysoká kvalita ovšem není zadarmo. Základní verze aplikace vyjde na 895 eur. Rozšířená verze je ještě o 200 euro dražší a přidává navíc podporu pro A3 skenery a rychlejší překlad. Ve zpracování skenovaných dokumentů tak OBR (Obrázek 7.1) patří ke špičce ve své kategorii.



Obrázek 7.1: Vzhled aplikace OBR

7.2 Braille Reader VDL

I tato aplikace je primárně určena ke zpracování Braillských knih. Dovoluje rozpoznání jednostranných i oboustranných dokumentů a také nabízí přímé propojení se skenerem. Přesnou procentuální hodnotu úspěšnosti rozpoznání výrobce neuvádí (předpokládám však hodnotu nižší než v případě OBR, kdyby byla vyšší, určitě by se s ní výrobce pochlubil). Na druhou stranu aplikace umožňuje tzv. interaktivní režim, ve kterém je možné ručně opravit špatně určené nebo nerozpoznané body a tím získat korektní výsledek překladu. Uživatelské rozhraní Braille Reader VDL je na následujícím obrázku (Obrázek 7.2).



Obrázek 7.2: Vzhled aplikace Braille Reader VDL

Uvedené aplikace jsem bohužel neměl možnost osobně vyzkoušet. Obě jsou však zaměřené na rozpoznávání skenovaných předloh, stejně jako další software pro rozpoznání Braillova písma. Přímé srovnání s produktem zpracovávajícím fotografie s různě velkými a natočenými body, jako se zabývá tato práce, proto není možná.

8 Závěr

V úvodu práce byly jasně definovány cíle, kterých je třeba dosáhnout. Jednotlivým požadavkům zadání je zde věnován prostor adekvátní jejich důležitosti a považuji tak všechny body za úspěšně splněné. Prvním z cílů projektu bylo seznámení s principy Braillova písma. Ke splnění toho bodu jsem prostudoval různé zdroje a nasbíral teoretické znalosti o způsobech zápisu, parametrech písma a dalších vlastnostech, jak je uvedeno v kapitole 2. Dalším cílem bylo proniknutí do problematiky zpracování obrazu a knihovny OpenCV. Tomuto tématu je věnován prostor v kapitole 3, kde je stručně probrána teorie snímání obrazu a dále nepřímo v kapitole 5, ve které je prakticky využita knihovna OpenCV při tvorbě detektoru reliéfních bodů. Třetí cíl zadání stanovil návrh aplikace k rozpoznání písma ve fotografii. Tento bod je splněn v kapitole 4, kde jsou názorně představeny jednotlivé fáze zpracování obrázku od detekce písma až po překlad. Dalším požadavkem bylo provedení implementace podle představeného návrhu. Tímto bodem se důkladně zabývá kapitola 5, ve které jsou podrobně popsány algoritmy klíčových funkcí implementované v jazyku C++. Pro lepší představu jsou zde i schémata jejich činnosti. Menší prostor je naopak věnován uživatelskému rozhraní, které nebylo v této práci prioritou. Kapitola 6 se věnuje poslednímu bodu zadání, kterým bylo provedení testů k vyhodnocení použitelnosti. Testování je rozděleno na skupiny podle typu zdroje, abychom mohli určit výsledky jednotlivě. Bohužel požadované materiály nejsou zcela běžné dostupné, proto všechny testy musely proběhnout s poměrně malým množstvím vstupních vzorků. Pro vyhodnocení byl však tento počet dostatečný.

Přínosem práce je tak aplikace, která poměrně dobře plní svůj účel pro primární zdroj obrázků, pro jaké byla navržena. Tedy fotografie obsahující menší množství textu, s výraznějšími body. Není zde kladena žádná podmínka na rozlišení ani jejich kvalitu, avšak k úspěšné detekci je třeba alespoň minimální odlišnost pozadí a reliéfních bodů. Ani u skenovaných dokumentů není úspěšnost překladu špatná. Jejich kvalita však musí být vyšší. Při menším kontrastu či fyzicky poškozeném zdroji dochází k chybnému určování bodů a tedy i následný překlad obsahuje chyby. Pokud se navíc nějaký kaz detekuje jako bod mimo mřížku, překlad nemusí vůbec proběhnout z důvodu nemožnosti určit jednotlivé sloupce. Tuto slabinu je však možné odstranit ruční úpravou obrázku, jak bylo v jednom případě z testu nutné. Také u dokumentů s vytištěným významem jednotlivých symbolů dochází kvůli nutnosti jeho odstranění k degradaci obrazu a ztrátě určité informace. Detekce reliéfních bodů je potom neúplná, což se opět projeví ve výsledku jako chybně přeložené znaky. Nicméně i přes tyto nedostatky dosahuje aplikace poměrně slušné úspěšnosti. Navíc je k dispozici zdarma, což může být při vysokých cenách profesionálních produktů jen další plus pro někoho, kdo by potřeboval výjimečně přeložit pár stránek.

Co se týče pokračování v projektu, možných vylepšení je hned několik. V první řadě úprava detektoru. A to nejen pro vyšší přesnost a úspěšnost v rozpoznávání testovaných zdrojů, ale i v zavedení podpory zejména pro oboustranné dokumenty. Ty nebyly v testování vůbec použity, jelikož při detekci bodů je v tomto případě nutné zohlednit další parametry, které stávající detektor zatím nemá. Další možnost pro rozšíření použitelnosti představuje schopnost detekce bodů na barevně nejednotném podkladu. Tím by bylo možné rozpoznávat Braillovo písmo např. na obalech od léků. V úvahu připadá i větší podpora pro skenované dokumenty, zejména automatické spojování po sobě jdoucích stránek, které by umožnilo přeložit celou úsek jen zvolením první a poslední strany. Jako poslední bod k vylepšení bych uvedl rychlost celého zpracování. Optimalizací všech fází překladu by bylo možné kompletní zpracování ještě o něco urychlit.

Literatura

[1] Braillovo písmo. *Únia nevidiacich a slabozrakých Slovenska* [online]. 3. 1. 2012 [cit. 2012-01-03]. Dostupné z: http://unss.sk/sk/braillovo-pismo

[2] SMÝKAL, Josef. Pohled do dějin slepeckého písma. *PhDr. Josef Smýkal* [online]. © 1999–2011 [cit. 2012-01-03]. Dostupné z: http://smykal.ecn.cz/publikace/kniha08t.htm

[3] SYNEK, Svatopluk. Braillovo slepecké písmo. *Hendikep.cz* [online]. 3. 9. 2010 [cit. 2012-01-03]. Dostupné z: http://www.hendikep.cz/index.php?option=com_content&view=article&id=288%3Abraillovo-slepecke-pismo&catid=21%3Azrak-kompenzace

[4] ZEMČÍK, Pavel a Michal ŠPANĚL. *Zpracování obrazu*. Brno, 2006. Studijní opora. Vysoké učení technické v Brně, Fakulta informačních technologií.

[5] PIHAN, Roman. Obrazové problémy digitální fotografie. *Fotografovani.cz* [online]. 10. 4. 2009 [cit. 2012-04-26]. Dostupné z: http://www.fotografovani.cz/art/fotech_df/rom_trouble4.html.

[6] PIHAN, Roman. Rozumíme DSLR - 1. Základní konstrukce. *Fotografovani.cz* [online]. 16. 8. 2006 [cit. 2012-04-26]. Dostupné z: http://www.fotografovani.cz/art/fozak_df/rom_dslr1.html.

[7] DROBEK, Milan. Skenery a skenování. *SOUT Přelouč* [online]. 2010 [cit. 2012-04-26]. Dostupné z: http://www.sout-prelouc.cz/stranky/polygrafie_grafika_drobek/dokumenty/maturita10/graf_18_sken.pdf

[8] BRADSKI, Gary R a Adrian KAEHLER. *Learning OpenCV: Computer Vision with the OpenCV Library.* Sebastopol: O'Reilly, c2008, 555 s, ISBN 978-0-596-51613-0.

[9] GENČÚR, Martin. *Nástroj na zpracování fotografovaného textu*. Brno, 2007. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

[10] BLANCHETTE, Jasmin a Mark SUMMERFIELD. *C++ GUI programming with Qt 4*. 2nd ed. Upper Saddle River: Prentice-Hall, 2008, 718 s. ISBN 0-13-235416-0.

[11] *Neovision s.r.o.* [online]. [cit. 2012-04-30]. Dostupné z: http://www.neovision.cz/prods/obr

[12] *ElecGeste* [online]. [cit. 2012-04-30]. Dostupné z: http://www.elecgeste.ru/index.php?lng=english&sec=blind&mod=products&rid=24

Seznam příloh

Příloha A

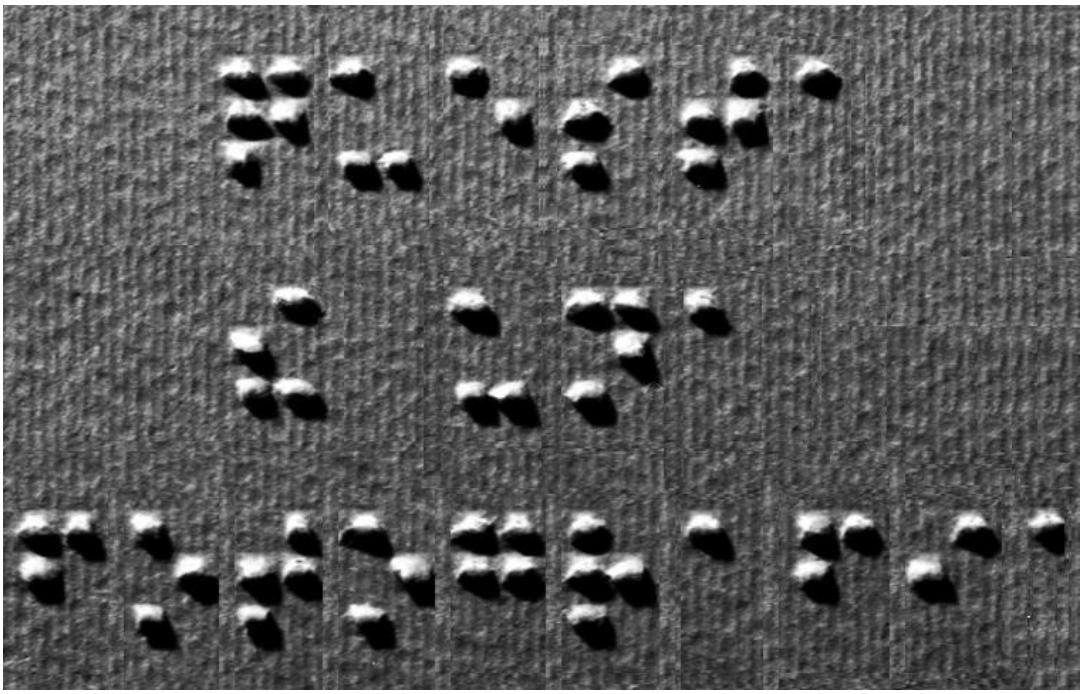
Obsah CD

Součástí této práce je CD, obsahující kompletní zdrojové kódy aplikace i technické zprávy. Způsob organizace souborů na médiu je následující:

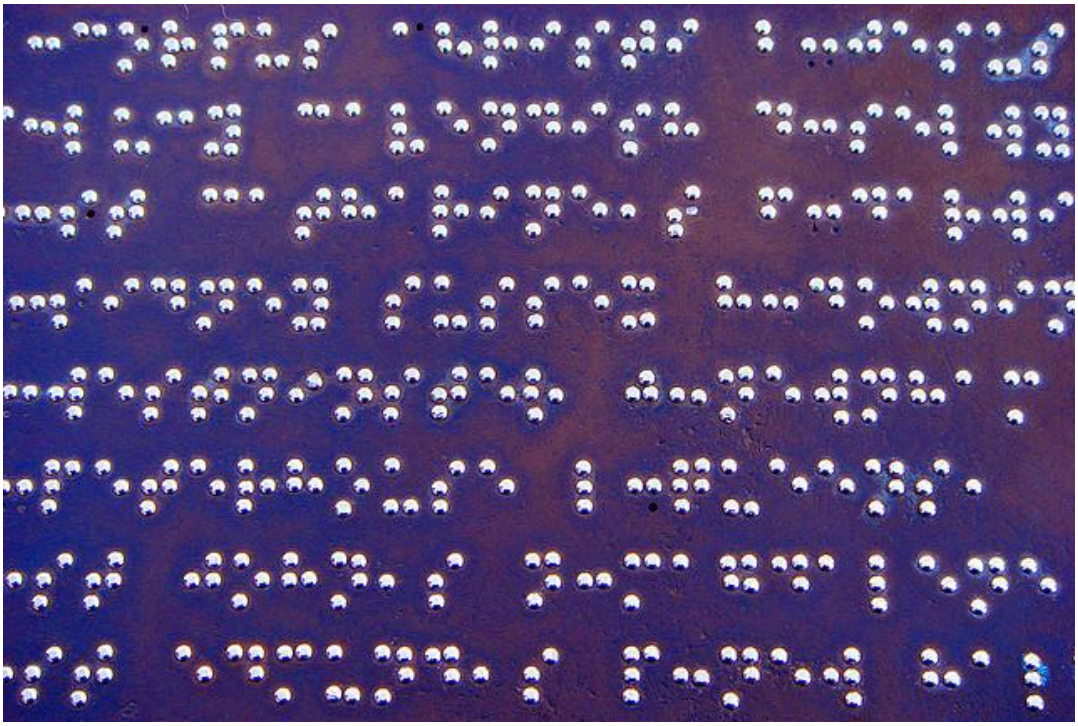
1. adresář `src` obsahuje:
   - zdrojové kódy projektu pro Qt Creator 2.4.
   - dokument Microsoft Word `DP.doc`, ve kterém je napsána tato práce.
2. adresář `test` obsahuje:
   - `skupina1` – fotografie pro 1. test
   - `skupina2` – skenované stránky pro 2. test
   - `skupina3` – předlohy pro 3. test
3. adresář `braider` obsahuje spustitelnou aplikaci pro systém Windows.
4. `DP.pdf` – technická zpráva ve formátu PDF.
5. `abecedaCZ.xml` – soubor s definicí české abecedy
6. `abecedaEN.xml` – soubor s definicí anglické abecedy

Příloha B

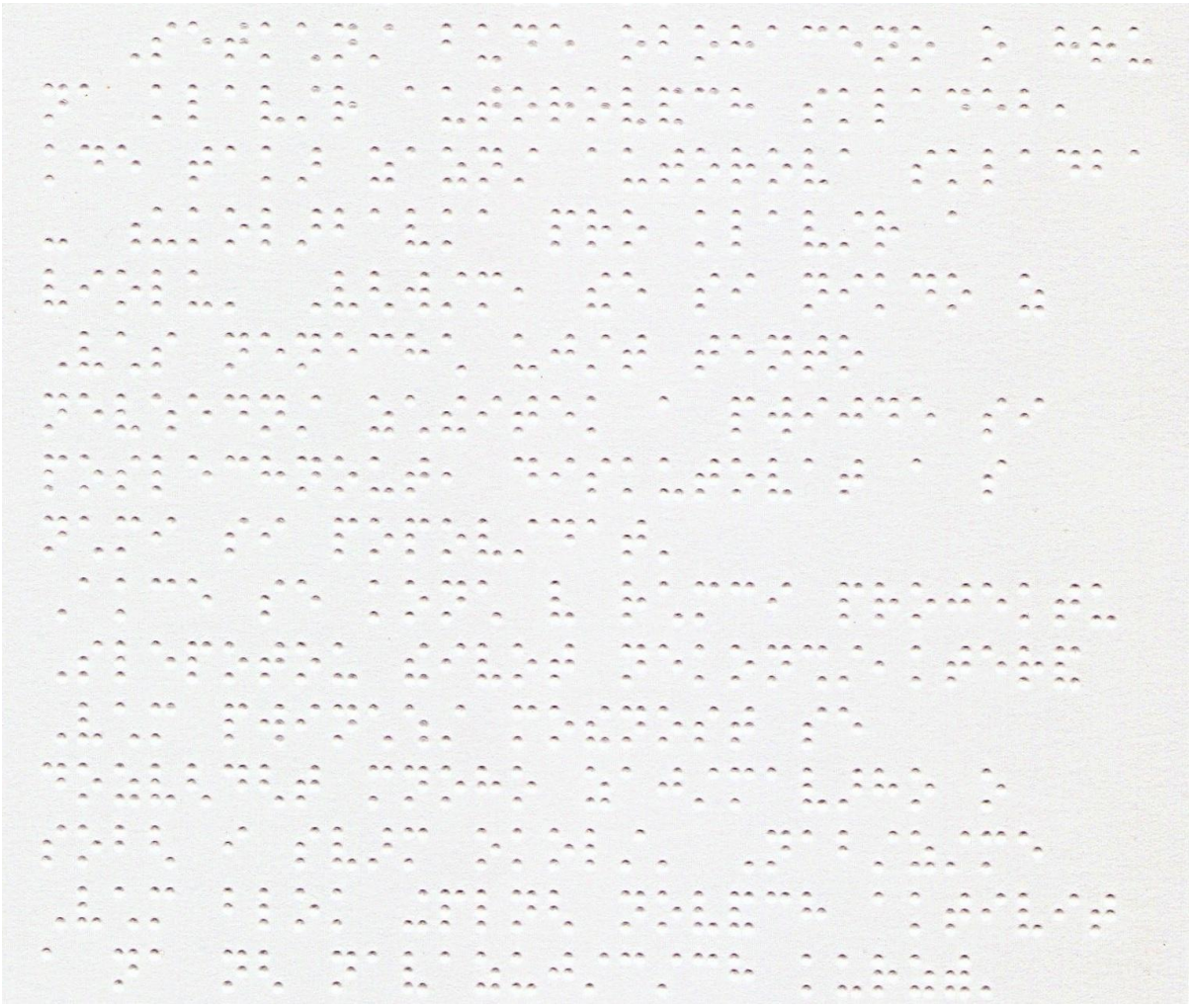
Ukázka testovacích dat



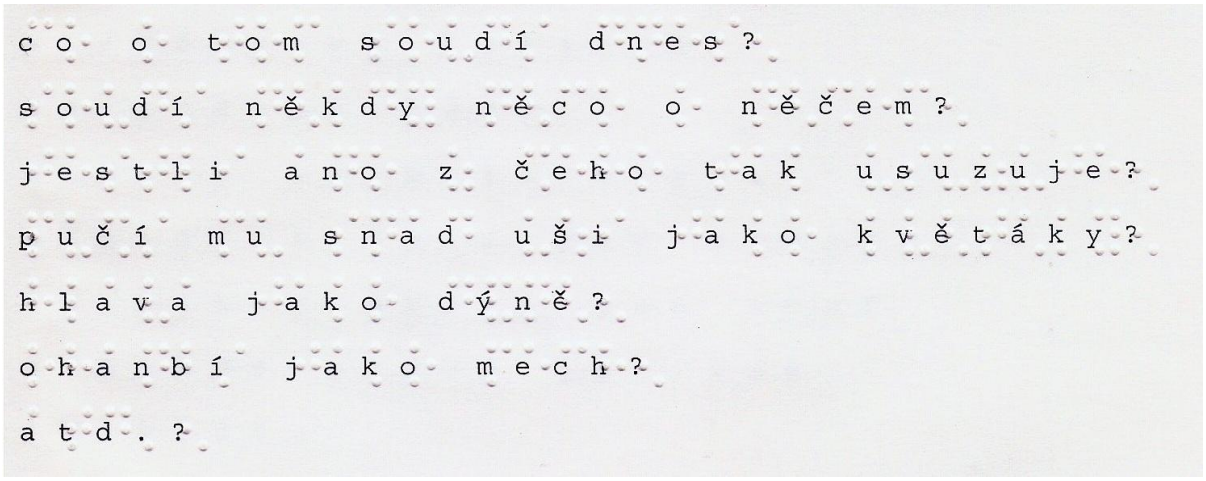
Obrázek B.1: Soubor foto3.jpg



Obrázek B.2: Soubor foto6.jpg



Obrázek B.3: Soubor sken5.jpg



Obrázek B.4: Soubor tisk3.jpg

Příloha C

XML soubor s definicí abecedy

```
<alphabet id="čeština">
  <char bin="000000" small=" "/>
  <char bin="100000" small="a" number="1"/>
  <char bin="110000" small="b" number="2"/>
  <char bin="100100" small="c" number="3"/>
  <char bin="100110" small="d" number="4"/>
  <char bin="100010" small="e" number="5"/>
  <char bin="110100" small="f" number="6"/>
  <char bin="110110" small="g" number="7"/>
  <char bin="110010" small="h" number="8"/>
  <char bin="010100" small="i" number="9"/>
  <char bin="010110" small="j" number="0"/>
  <char bin="101000" small="k"/>
  ...
  <char bin="101011" small="z"/>

  <char bin="100001" small="á" capital="Á"/>
  <char bin="100101" small="č" capital="Č"/>
  <char bin="100111" small="ď" capital="Ď"/>
  <char bin="001110" small="é" capital="É"/>
  <char bin="110001" small="ě" capital="Ě"/>
  <char bin="001100" small="í" capital="Í"/>
  <char bin="110101" small="ň" capital="Ň"/>
  <char bin="010101" small="ó" capital="Ó"/>
  <char bin="010111" small="ř" capital="Ř"/>
  <char bin="100011" small="š" capital="Š"/>
  <char bin="110011" small="ť" capital="Ť"/>
  <char bin="001101" small="ú" capital="Ú"/>
  <char bin="011111" small="ů"/>
  <char bin="111101" small="ý" capital="Ý"/>
  <char bin="011101" small="ž" capital="Ž"/>

  <char bin="010010" small=":"/>
  <char bin="011000" small=";"/>
  <char bin="001001" small="-"/>
  <char bin="010011" small="+"/>
  <char bin="110111" small="/"/>
  <char bin="010001" small="?"/>
  <char bin="011010" small="!"/>
  ...
  <char bin="000001" prefix="CAPITAL_LETTER"/>
  <char bin="000011" prefix="GROUP_CAPITALS"/>
  <char bin="000010" prefix="LETTER"/>
  <char bin="001111" prefix="NUMBER"/>
  <char bin="000101" prefix="CAPITAL_GREEK"/>
  <char bin="000110" prefix="GREEK_LETTER"/>
<alphabet/>
```