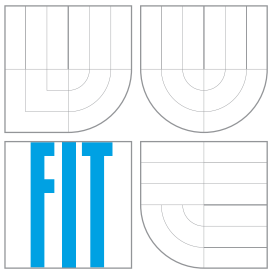
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

EDITOR KALIGRAFIE S ROZPOZNÁVÁNÍM JAPONSKÝCH ZNAKŮ

CALLIGRAPHY EDITOR WITH JAPANESE CHARACTER RECOGNITION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. PETR HORÁČEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. MIROSLAV ŠVUB
SUPERVISOR

BRNO 2009

Abstrakt

Práce řeší problematiku vytvoření aplikace pro podporu výuky japonských znaků. Její součástí je i stručný přehled vývoje a podob japonského písma. Diskutuje některá existující řešení a na základě jejich studia stanovuje požadavky na aplikaci. Rozebírá problémy, které je třeba v souvislosti s nimi řešit, a snaží se navrhnout možné postupy. Důležitou částí je rozpoznávání znaků. Následně práce popisuje zvolená řešení a jejich implementaci. Na závěr shrnuje a demonstruje dosažené výsledky a diskutuje možnosti dalšího vývoje systému.

Abstract

This work focuses on creating an application to support Japanese character learning. It also contains a brief overview of Japanese writing's history and evolution. Based on the study of existing options, this work sets the requirements for the application. It discusses problems and tries to find possible solutions. Character recognition is an important part. The work describes chosen solutions and their implementations. It ends by demonstrating achieved results and discussing options for further development of the system.

Klíčová slova

Japonské písmo, znaky kanji, výuková aplikace, kaligrafie, rozpoznávání znaků

Keywords

Japanese writing, kanji characters, learning application, calligraphy, character recognition

Citace

Petr Horáček: Editor kaligrafie s rozpoznáváním japonských znaků, diplomová práce, Brno, FIT VUT v Brně, 2009

Editor kaligrafie s rozpoznáváním japonských znaků

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Miroslava Švuba.

.....
Petr Horáček
24. května 2009

Obsah

Kapitola 1

Úvod

Tématem práce je editor kaligrafie s rozpoznáváním japonských znaků. První kapitola popisuje cíle, kterých chceme dosáhnout, a důvody, proč se problematikou zabýváme, a odkazuje na některá existující řešení.

Další kapitola se věnuje samotnému japonskému znakovému písmu. Stručně představuje jeho historii a vývoj a charakterizuje současný stav, s důrazem na oblasti přímo související s naší prací, jako jsou různá čtení znaků a zejména správný způsob psaní.

Ve třetí kapitole se zmíníme o technikám používaných v počítačové grafice, které se našeho tématu dotýkají. Pokusíme se v nich najít částí, kterých bychom mohli využít nebo se jimi inspirovat při tvorbě našeho systému.

Následující kapitola je věnována již samotnému vyvíjenému systému. Pokusíme se zde formulovat požadavky na aplikaci. Stanovíme problémy, jež budeme muset řešit, a představíme možné postupy. Vybrané metody podrobněji rozebereme.

Prostředky, použité pro realizaci systému, jsou zmíněny v další kapitole. Zde také představíme provedenou implementaci formou stručné programové dokumentace a vysvětlení jednotlivých částí.

V závěru zhodnotíme dosažené výsledky a navrhneme směry, jimiž by se mohl ubírat další vývoj v této oblasti.

Kapitola 2

Cíle a motivace

Cílem práce je vytvořit aplikaci pro podporu výuky japonských znaků *kanji*. Program by měl být schopen pracovat v několika režimech. Ve výukovém módu budou uživateli zobrazovány jednak samotné znaky, ale také správný postup jejich psaní a jejich různé významy a výslovnosti. Další režim bude sloužit k otestování naučených znalostí — potřebujeme tedy také, aby aplikace dokázala rozpoznat znak, který uživatel nakreslí např. pohybem myši. Kromě toho by program měl umožňovat přidávat nové znaky do databáze.

Hlavní důvody, proč jsem se rozhodl zabývat tímto problémem, jsou dva. V prvé řadě není v současné době stále dostatečně rozpracován. Ačkoliv existují aplikace s obdobnou funkčností, není jich příliš mnoho a obvykle mívají určité nedostatky, jichž se v rámci této práce snažíme vyvarovat.

Příkladem takového programu je *Yokozuna!* [8] (společnost stojící za jeho vývojem bohužel pravděpodobně zanikla). Obrázek 2.1 zachycuje tuto aplikaci v učebním režimu. Pro výuku může být poměrně užitečný, ale je zde např. poněkud potlačeno kaligrafické hledisko — znaky jsou zobrazovány pouze pomocí jednoduchých čar s konstantní tloušťkou — a ani rozpoznávání znaků se nezdá být dostatečně spolehlivé. Poměrně kvalitně jsou v této aplikaci naopak řešeny záležitosti přímo spojené s výukou, jako je rozdělení znaků do skupin podle významu a obtížnosti nebo obkreslování zobrazeneného znaku s možností nechat si napovědět, kde by měl začínat další tah. Těmito funkcemi bychom se mohli inspirovat.

Obrázek 2.1: Program *Yokozuna!* 1.1

Druhým významným důvodem pro volbu dané problematiky je můj osobní dlouhodobý zájem o japonskou kulturu a jazyk. Mimo jiné doufám, že mi tato práce může pomoci naučit se některé japonské znaky.

Kapitola 3

Japonské znakové písmo

Současné japonské písmo používá tři sady znaků.

Kanji jsou původem čínské znaky. Obvykle se jich používá pro zápis významové části, tj. podstatných jmen a kořenů přídavných jmen a sloves. Také japonská jména jsou ve většině případů psána pomocí těchto znaků.

Kromě *kanji* existuje tzv. *kana*. Jednotlivé znaky nemají vlastní význam, ale pouze danou výslovnost. Slova lze vytvářet jejich skládáním, podobně jako v případě písmen latinské abecedy. Znaky přibližně odpovídají slabikám v češtině, až na jedinou výjimku jejich čtení vždy zahrnuje buď pouze samohlásku, nebo souhlásku následovanou samohláskou. V současnosti jsou používány dvě skupiny znaků, *hiragana* a *katakana*, které se liší pouze svým vzhledem a stylisticky tím, v jaké situaci se užívají. Všechna slova psaná pomocí *kanji* lze zapsat také pomocí těchto systémů, viz níže.

Hiragana je v japonštině využita především pro koncovky sloves a přídavných jmen (určují tvar slova) a partikule (částice), které jsou velmi významnou a hojně používanou součástí jazyka. Na rozdíl od češtiny, kde částice vždy upravují význam celé věty, v japonštině existují různé druhy částic, jako např. pádové nebo kontextové. Mohou mít vliv na jednotlivá slova nebo slovní spojení, případně vymezovat vztahy mezi větnými členy atd. Systémem *hiragana* můžeme dále psát slova, jejichž *kanji* není běžné nebo je příliš složité, sami jej neznáme nebo máme důvod předpokládat, že by jej nemusel znát čtenář; pro některá slova navíc *kanji* vůbec neexistuje. Zápis pomocí znaků *hiragana* může posloužit také jako návod pro čtení znaků *kanji* při jejich výuce.

Katakana obvykle slouží k zápisu cizích slov a jmen, nebo slov technického a vědeckého charakteru. Také se používá pro citoslovce. Zápis v systému *katakana* můžeme zvolit i v případě, že chceme některá slova zvýraznit, podobně jako používáme např. italiku v českém textu.

V moderní japonštině se do určité míry užívá i latinské abecedy, například pro zkratky jako „CD“ nebo „NATO“. Ačkoliv existují znaky *kanji* pro čísla, v současnosti je běžnější při horizontálním způsobu psaní — tradiční způsob je po sloupcích shora dolů a zprava doleva, ale dnes již bývá obvyklejší psát po řádcích zleva doprava — používat arabské číslice.

V této práci se nás budou zajímat především znaky *kanji*.

3.1 Historie a vývoj *kanji*

Čínské znaky se do Japonska dostaly nejprve na předmětech dovážených z Číny. Není jisté, kdy přesně sami Japonci začali ovládat klasickou čínštinu, první japonské dokumenty byly pravděpodobně napsány čínskými přistěhovalci. Důležité je, že od šestého století našeho letopočtu se začínají v čínských dokumentech psaných v Japonsku objevovat interference japonštiny, což ukazuje na rozšířené přijetí činských znaků v Japonsku.

Neexistují doklady existence žádné písemné formy samotné japonštiny před přijetím *kanji*. Původně byly texty psány a čteny čínsky. Později se ale objevují modifikace a doplňky, které umožňují upravit čínský text tak, aby jej bylo možné použít v rámci japonské gramatiky. Čínské znaky jsou poté používány i k zápisu japonských slov.

Dnešní *kanji* se již od tradičního čínského písma *hanzi* významně odlišuje. Existují znaky nově vytvořené v Japonsku, které čínština vůbec nemá (tzv. *kokuji* neboli národní znaky). Kromě toho některé znaky si sice zachovaly svou podobu, ale v japonštině dostaly zcela jiný význam, než měly původně (tzv. *kokkun*). V neposlední řadě došlo ke zjednodušení znaků z původní tradiční čínštiny.

3.2 Čtení znaků

Jeden znak může sloužit k zápisu více různých slov, resp. morfémů. Říkáme, že *kanji* má několik různých *čtení*. Rozhodnutí, které čtení má být užito, závisí na kontextu, zamýšleném významu, použití ve složeninách nebo i umístění ve větě. Čtení můžeme rozdělit do dvou základních skupin — čínské čtení *on'yomi* a japonské čtení *kun'yomi*.

3.2.1 *On'yomi*

Čínsko-japonské čtení, které vzniklo japonskou nápodobou původní čínské výslovnosti znaku. Protože některé znaky se dostaly do Japonska v různou dobu z různých částí Číny, mají několik různých *on'yomi*, a často i významů. Znaky vytvořené už přímo v Japonsku obvykle toto čtení nemají, ale existují i výjimky.

3.2.2 *Kun'yomi*

Japonské čtení *kun'yomi* je založeno na výslovnosti japonského slova, jehož význam se dostatečně blíží významu čínského znaku při jeho převzetí. Opět může jeden znak mít i více těchto čtení.

3.2.3 Jaké čtení použít?

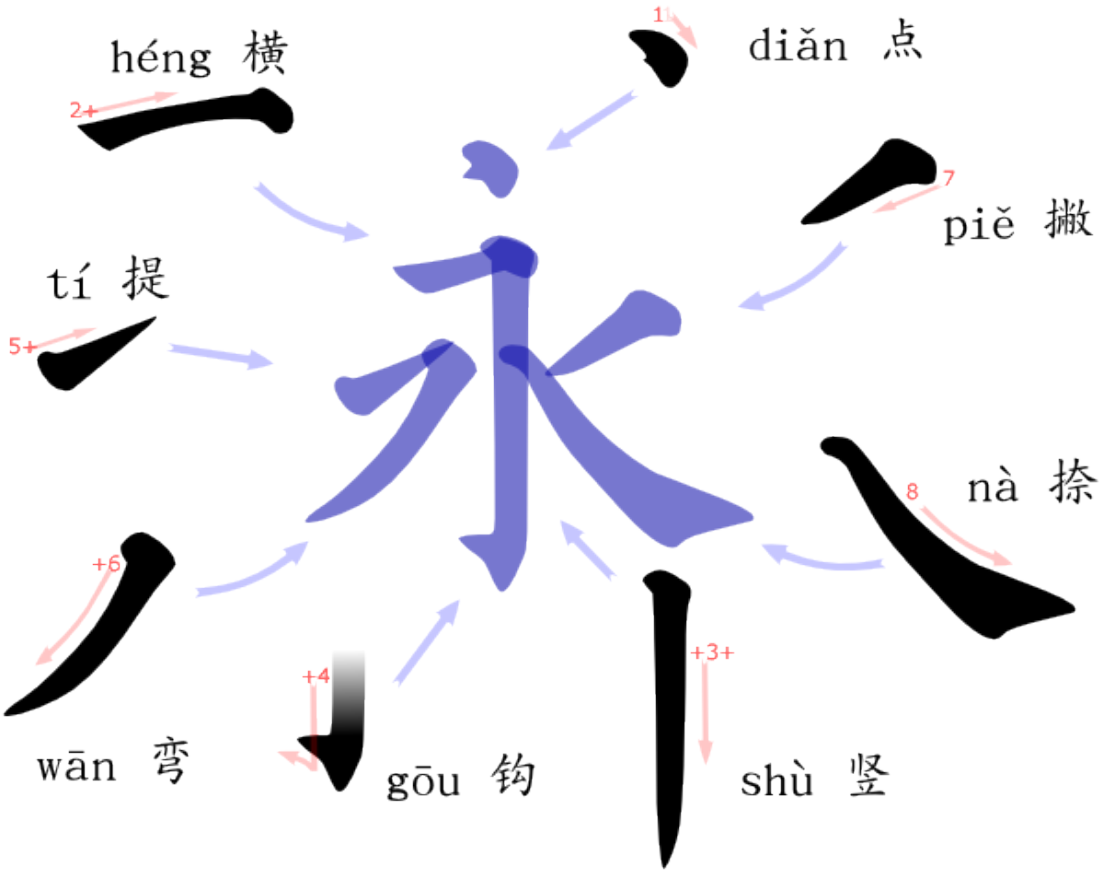
Poznat, kdy použít jaké čtení, často nemusí být snadné ani pro rodilého mluvčího. Existují sice obecná pravidla, ale výjimek je příliš mnoho, takže je obvykle nutná předchozí znalost.

Poměrně často platí, že *kanji*, jež se vyskytuje v textu samostatně, čteme pomocí *kun'yomi*. Naopak pro znaky, které jsou součástí složenin, obvykle použijeme *on'yomi*. V některých případech to ovšem může být i jinak, nelze se tedy na tuto pomůcku obecně spoléhat. Dalším problémem je, že znak může mít více čtení v rámci jedné z těchto kategorií. Může se tedy např. stát, že stejný znak se čte v různých složeninách jinak, přitom všechna tato čtení mohou spadat do skupiny *on'yomi*. Jediným spolehlivým způsobem tedy zůstává naučit se kromě jednotlivých čtení znaku také to, v jaké situaci se které z nich použije.

3.3 Psaní, základní tahy

Pro japonské znaky je základem osm jednoduchých tahů a okolo třiceti složených — tento počet se může v různých zdrojích mírně lišit, např. v [2] je definováno těchto tahů třicet, v [6] pouze dvacet devět. Některé tahy, které naleznete v jednom zdroji, v jiném mohou chybět, a naopak. Složené tahy se skládají ze dvou nebo více uvedených tahů základních.

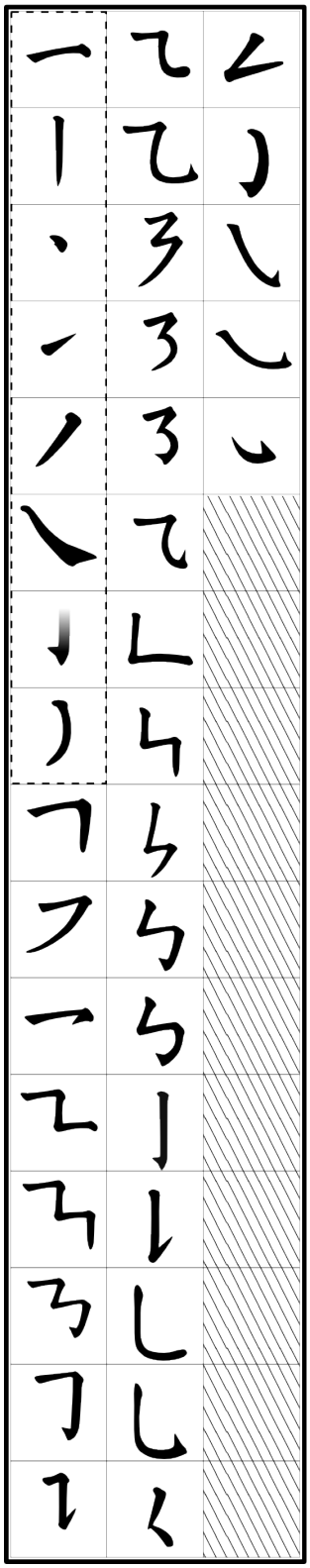
Základní tahy jsou dobře patrné ve znaku *yong* (věčnost), zachyceném na obrázku 3.1. Na tomtéž obrázku můžeme vidět také směr jejich psaní (znázorněný červenou šipkou) a jejich názvy. Složené tahy viz obrázek 3.2.



Obrázek 3.1: Osm základních tahů

Pro psaní znaku platí přesná pravidla. Velmi důležité je dodržovat pořadí tahů. Existuje několik obecných rad, které nám mohou pomoci (je ale třeba si uvědomit, že pořadí závisí vždy na konkrétním znaku):

1. Znak píšeme zleva doprava a shora dolů.
2. Pokud se tahy kříží, horizontální má přednost před vertikálním.
3. Tahy, které znak přetínají vertikálně, se píší jako poslední.
4. Diagonální tahy směřující zprava doleva přecházejí opačné.



CJK strokes :

8 basic strokes :

一	CJK STROKE H	横	hēng
丨	CJK STROKE S	竖	shù
丶	CJK STROKE D	点	diǎn
丿	CJK STROKE T	提	tí
乀	CJK STROKE P	撇	piě
乁	CJK STROKE N	捺	nà
亅	CJK STROKE G	鈎	gōu
乛	CJK STROKE W	彎	wān

29 other complex strokes :

㇁	CJK STROKE HZ
㇇	CJK STROKE HP
㇋	CJK STROKE HG
㇌	CJK STROKE HZZ
㇍	CJK STROKE HZZZ
㇎	CJK STROKE HZZZG
㇉	CJK STROKE HZG
㇊	CJK STROKE HZT
㇈	CJK STROKE HZW
㇏	CJK STROKE HZWG
㇄	CJK STROKE HZZP
㇅	CJK STROKE HPW
㇆	CJK STROKE HPWG
㇃	CJK STROKE HWG
㇂	CJK STROKE SZ
㇀	CJK STROKE SZZ
㇇	CJK STROKE SZP
㇀	CJK STROKE SZZG
㇀	CJK STROKE SZZWG
㇋	CJK STROKE SG
㇉	CJK STROKE ST
㇂	CJK STROKE SW
㇂	CJK STROKE SWG
㇌	CJK STROKE PD
㇄	CJK STROKE PZ
㇅	CJK STROKE W
㇈	CJK STROKE XG
㇏	CJK STROKE BXG
㇆	CJK STROKE WG

Obrázek 3.2: Přehled základních a složených tahů podle [6]

5. Přednost mají vertikální tahy vniřní před vnějšími.
6. Vnější uzavírající tahy se píší před vnitřními tahy, ovšem spodní uzavírající tah se píše až nakonec.
7. V případě uzavírajících tahů začínáme levým vertikálním tahem.
8. Jako úplně poslední píšeme malé tahy („tečky“).

Ačkoliv by se na první pohled mohlo zdát dodržování správného pořadí tahů jako zbytečná komplikace, není to samoúčelné. Ve skutečnosti obvykle bývá podstatně jednodušší naučit se znak tímto způsobem, než kdybychom se například pokoušeli zapamatovat si pouze jeho celkový vzhled.

Podrobnější návod a ilustrační obrázky viz [5].

3.4 Radikály

V různých znacích *kanji* můžeme najít určité společné podčásti. To jsou tzv. radikály (bushu). Každý znak obsahuje radikál, a také samotný radikál je znakem. Radikály nám mohou významně pomoci určit obecnou povahu znaku. Můžeme podle nich usuzovat na původ znaku, skupinu, do níž spadá, význam nebo výslovnost. Také slovníky *kanji* často organizují znaky podle radikálů. Celkový počet radikálů je dvě stě čtrnáct, ale pro běžnou potřebu není nutné naučit se naprosto všechny, postačí nám některé významnější (ani rodilí mluvčí nemusejí vždy znát veškeré radikály). Můžeme je obecně rozdělit do sedmi skupin podle jejich pozice ve znaku, viz obrázek 3.3 — červeně je vyznačena pozice.

Obrázek 3.3: Skupiny radikálů podle pozice ve znaku

Jako příklad nam pouslouží radikál *tehen* (ruka) ze skupiny *hen* (umístění v levé části znaku), viz obrázek 3.4. Zcela vlevo je vyzobrazen samotný radikál, dále pak následují příklady znaků, v nichž se vyskytuje.

Obrázek 3.4: Příklady použití radikálu *tehen*

Více informací včetně přehledu běžnějších radikálů lze najít např. v [1].

3.5 Kaligrafie

Významná je také umělecká stránka *kanji*. V zemích východní Asie, kde se používají čínské znaky, byl v historii vždy kladen na kaligrafii velký důraz. Kaligrafie je nejen považována za formu umění, ale inspirovala zde i malířské techniky.

Tradiční čínská kaligrafie používá štětec, tuš a zvláštní druh papíru. Tím je ovlivněn charakteristický vzhled tahů. Příklad japonské kaligrafie můžeme vidět na obrázku 3.5, autorem je japonský vládní úředník *Tachibana no Hanayari* (782–844).

Obrázek 3.5: Příklad japonské kaligrafie

Kapitola 4

Inspirace

4.1 Vektorové písmo

V počítačové grafice se pro zobrazování textu běžně používají tzv. *fonty*. Jedná se o datové soubory, které obsahují množiny tzv. *glyfů* — tak se v typografii označuje konkrétní grafická podoba grafému, tj. abstraktního znaku, kterým může být např. písmeno, číslice, interpunkční znaménko, nebo ovšem také celý znak v případě znakového písma (poněkud nešťastné je, že pojmem znak zde musíme označovat dvě odlišné věci; v rámci této podkapitoly budeme rozumět znakem grafém, nebude-li řečeno jinak). Pokusme se podívat se na existující způsoby realizace fontů a najít části, kterými bychom se mohli při tvorbě našeho systému inspirovat.

V současnosti existují tři hlavní principy, na kterých mohou být fonty založeny. *Rastrové (bitmapové) fonty* používají pro reprezentaci podoby znaku matici pixelů, neboli bitovou mapu. Jejich implementace je relativně jednoduchá a použití může být relativně snadné a rychlé. Zásadní nevýhodou ovšem je, že pro každou možnou velikost musíme mít k dispozici zvláštní bitovou mapu (případně se můžeme pokusit transformovat existující mapu na jinou velikost, to ale nemusí dávat vždy příliš dobré výsledky).

Druhou, v současné době asi nejrozšířenější realizací jsou tzv. *vektorové fonty* (přip. *obrysové*, angl. *outline*). K popisu grafické podoby znaku používají křivky (obvykle Bézierovy), vykreslovací instrukce nebo matematické rovnice. Hlavní výhodou je, že tyto fonty mohou být na základě jediného popisu zobrazeny v teoreticky libovolné velikosti bez ztráty kvality. Vykreslení těchto fontů může být ovšem pomalejší, protože se vždy musí počítat příslušné pixely. Do této kategorie patří mimo jiné dnes patrně nejběžnější systém fontů *TrueType*. Pro náš projekt bude s největší pravděpodobností podstatně vhodnější založit vykreslování japonských znaků na principu vektorovém než rastrovém.

Velkou inspiraci můžeme najít také ve třetím principu fontů. Jedná se o poměrně málo známé *fonty založené na tazích (stroke-based)*. Daly by se považovat za určité rozšíření vektorových písem v tom smyslu, že celý znak je složen z tahů, které jsou zvlášť popsány podobným způsobem jako znaky v případě vektorových fontů.

Celý znak je pak složen z několika těchto tahů. Můžeme tedy používat stejné tahy v různých znacích, aniž bychom je museli opakovaně definovat. Nebude jistě překvapením, že nemalá část z takto navržených fontů byla určena právě pro písma používaná ve východní Asii. Ačkoliv se tento způsob realizace fontů v praxi příliš neprosadil, my se jím můžeme nechat inspirovat pro náš projekt.

Více informací o jednotlivých typech fontů včetně některých konkrétních příkladů můžete najít např. v [7].

Kapitola 5

Návrh systému

V úvodní kapitole Cíle a motivace jsme se již stručně zmínili o požadavcích, které klademe na vytvářenou aplikaci. Nyní si je probereme poněkud podrobněji. Zmíníme základní problémy s nimi související a nastínime možné způsoby jejich řešení.

5.1 Požadavky na aplikaci

Vyvíjený program má sloužit v prvé řadě jako podpůrný prostředek pro výuku japonských znaků *kanji*. Bude tedy muset obsahovat databázi těchto znaků a měl by nabízet i nástroje k uprávě této databáze, především možnost přidávání nových znaků.

Minimálně by měl program nabízet dva režimy. Prvním z nich je výukový mód. V tomto režimu budou uživateli vhodným způsobem prezentovány jednotlivé znaky. Aplikace by měla uživateli pomoci získat několik základních znalostí.

Především se jedná o grafickou podobu znaku a způsob, jakým se znak píše, tedy z jakých tahů je složen a v jakém pořadí mají dané tahy následovat. Vhodné může být nejprve uživateli tuto informaci graficky zobrazit a následně jej nechat vyzkoušet si znak napsat. Přitom mu bude sděleno, zda si počíná správně, případně může aplikace poskytnout také nápovědu, jak by měl pokračovat.

Dále je třeba, aby se uživatel naučil jednotlivá čtení znaku, tedy výslovnosti a významy. Tuto informaci můžeme uživateli zobrazit textově. Mohlo by být užitečné uvést i příklady složenin, v nichž se daný znak může vyskytovat. Bude ale třeba dávat pozor, aby uživatel nebyl přehlcen informacemi.

V zkušebním režimu si uživatel bude moci své získané znalosti ověřit. Nabízí se několik možností. Program může uživateli např. zobrazit grafickou podoba znaku a žádat jej o doplnění významů nebo čtení. Složitější na řešení bude pravděpodobně opačný způsob, tj. na základě významů, resp. čtení znaku musí uživatel znak správně napsat. Zde budeme potřebovat, aby aplikace byla schopna znaky rozpoznávat.

5.2 Rozpoznávání znaků

Klíčovou součástí programu je schopnost rozpoznat znak, který uživatel napíše pomocí myši, případně tabletu nebo jiných vstupních zařízení. Nabízí se dvě obecné možnosti, jak tento problém řešit.

Můžeme nechat uživatele nejprve nakreslit celý znak a až následně analyzovat výsledný obrázek (bitmapu). V současné době existuje celá řada metod pro klasifikaci obrazu. V na-

šem případě však je problém v tom, že znaků, a tedy tříd klasifikace, je příliš velké množství na to, aby se daly tyto metody úspěšně použít. Druhou zásadní nevýhodou je, že bychom ztratili veškeré informace o procesu tvorby znaku, které také mohou být velmi důležité, jako např. pořadí tahů.

Výhodnější tedy bude analyzovat znak zadávaný uživatelem přímo v průběhu psaní, nebo přinejmenším zaznamenávat informace o procesu tvorby. Snadno můžeme mimo jiné rozdělit znak do jednotlivých tahů — v případě kreslení pomocí myši tah začíná v okamžiku, kdy je stisknuto tlačítko myši, a končí jeho uvolněním. Pohyb myši můžeme vzorkovat s vhodně zvoleným krokem.

Při volbě druhého způsobu tedy snadno dokážeme rozdělit znak na jednotlivé tahy a získat jejich pořadí. Nyní budeme muset vyřešit dva problémy — jak pro jednotlivé tahy určit, o který ze základních nebo složených tahů se konkrétně jedná, a co nejvhodněji popsat způsob, jakým je daný znak z jednotlivých tahů složen.

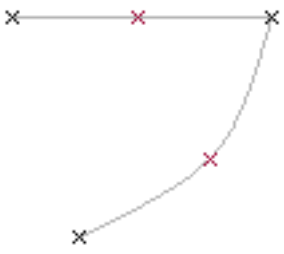
5.2.1 Rozpoznání tahu

Zde máme opět možnost volby. Buď budeme přímo rozpoznávat pouze základní tahy a tahy složené popíšeme jako jejich kombinaci (podobně jako znak budeme popisovat jako kombinaci tahů), nebo každý tah opatříme vlastním, nezávislým popisem.

Z provedených experimentů jsem shledal výhodnějším druhý způsob. Celkově všech tahů není příliš velký počet, takže je možné každý z nich popsat dostatečně podrobně. V některých složených tazích mohou základní tahy vypadat i výrazně odlišně nebo může být problém rozlišit v rámci tahu jednotlivé složky od sebe. Je tedy snazší popsat tah vždy jako celek než uvažovat všechny možné variace základních tahů.

Řesení, k němuž jsem nakonec dospěl, popisuje tah na základě jeho rozdělení na jeden nebo více (obecně neomezené množství) segmentů — ty často mohou odpovídat právě základním tahům, ale neplatí to vždy a není to podmínkou. Tah rozdělíme tak, že určíme body, v nichž dochází k ostrému zlomu — nazvěme je zlomové body.

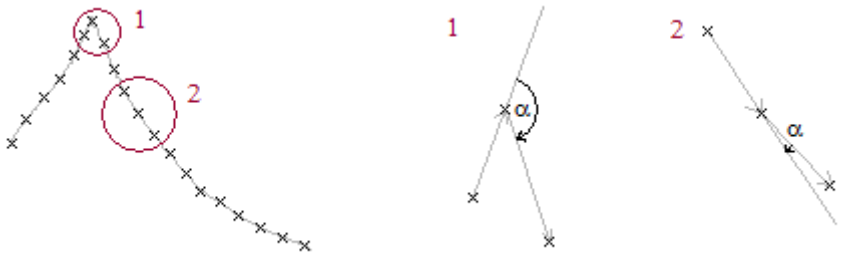
Jednotlivé segmenty pak jsou vždy určeny jednak relativní pozicí svého koncového bodu vůči bodu počátečnímu, navíc také mohou obsahovat několik kontrolních bodů. To jsou takové body, jimiž musí segment vždy procházet. Nesmí se v nich ale lámat (jinak by zde muselo dojít k rozdělení na více segmentů). Pozice kontrolních bodů jsou udávány relativně vůči počátečnímu bodu segmentu. Princip rozdělění tahu na segmenty a kontrolních bodů ilustruje obrázek 5.1. Černé body rozdělují tah na segmenty, červeně jsou vyznačeny body kontrolní.



Obrázek 5.1: Rozdělení tahu na segmenty, kontrolní body

Máme-li k dispozici množinu vzorků (bodů na dráze, kterou uživatel opsal pohybem myši), potřebujeme nejprve provést rozdělení na segmenty. Jak najít ve vzorcích takové

body, v nichž dochází k ostrému zlomu? Relativně jednoduchý způsob spočívá v tom, že vypočítáme úhly mezi dvěma vektory každých dvou po sobě jdoucích bodů, viz obrázek 5.2. Vlevo je vidět navzorkovaný tah, který užvatel nakreslil, uprostřed detail výpočtu úhlu v bodě, kde zlom je, vpravo v bodě, kde zlom není.



Obrázek 5.2: Hledání zlomových bodů

Algoritmus vypadá zhruba takto:

1. Označme souřadnice tří po sobě jdoucích vzorků (x_1, y_1), (x_2, y_2), (x_3, y_3). Prostřední vzorek (x_2, y_2) je tím, pro nějž zjišťujeme, zda v něm dochází ke zlomu.

2. Příslušné vektory pak jsou $(x_2 - x_1, y_2 - y_1)$ a $(x_3 - x_2, y_3 - y_2)$ (bylo by možné zvolit i jinak, ale uvažujme nyní tuto možnost).

3. Normalizujeme vektory, tzn. podělíme souřadnice délkou vektoru. Získáme vektory (x_{n1}, y_{n1}) a (x_{n2}, y_{n2}), kde

$$x_{n1} = (x_2 - x_1)/l_1 \tag{5.1}$$
$$y_{n1} = (y_2 - y_1)/l_1 \tag{5.2}$$
$$x_{n2} = (x_3 - x_2)/l_2 \tag{5.3}$$
$$y_{n2} = (y_3 - y_2)/l_2 \tag{5.4}$$
$$l_1 = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{5.5}$$
$$l_2 = \sqrt{(x_3 - x_2)^2 + (y_3 - y_2)^2} \tag{5.6}$$

4. Nyní už můžeme vypočítat úhel α podle vzorce

$$\alpha = \arccos\left(x_{n1} \times x_{n2} + y_{n1} \times y_{n2}\right) \tag{5.7}$$

5. Získaný úhel α porovnáme s vhodně zvoleným mezním úhlem ϕ. Pokud $\alpha > \phi$, v daném bodě detekujeme zlom.

Problémem, který zbývá dořešit, je, jak vhodně zvolit zmíněný mezní úhel ϕ. Na základě implementace jednoduchého systému a experimentů s ním jsem dospěl k závěru, že nejvhodnější hodnota se pohybuje okolo čtyřiceti stupnů. Rozhodl jsem se tedy položit $\phi = 40°$.

Z provedených expermientů vyplynul ještě další závěr. Zvolená metoda měla poměrně vysokou úspěšnost, ale příliš často se stávalo, že bylo jako zlomové body označeno několik po sobě bezprostředně následujících vzorků. Částečně lze tento problém vyřešit zvětšením

vzorkovacího kroku, ale pak se naopak může stát, že některé zlomy nebudou odhaleny vůbec, protože v daném místě jednoduše není žádný vzorek. Řešení je však poměrně triviální. Vzorkovací krok můžeme ponechat relativně malý a použít navrženou metodu detekce. Pouze ji doplníme tak, že jako výsledný zlomový bod vybereme pouze jeden z případné skupiny označených sousedících vzorků. Příliš v tomto případě nezáleží na tom, který z nich se rozhodneme zvolit, může to být např. první nalezený.

Při rozpoznávání tahu nakresleného uživatelem pak nejdříve můžeme porovnat počet segmentů, pokud se ten neshoduje, je zřejmé, že nemůže jít o daný tah. Odpovídá-li počet, postupně otestujeme na základě hladkých kontrolních bodů a pozice konce vůči začátku zvlášť každý segment. Tah je rozpoznán, pokud všechny segmenty v analyzovaném tahu odpovídají (s vhodně stanovenou tolerancí) příslušným segmentům tahu referenčního. Provnání jednoho segmentu můžeme provést takto:

1. Vypočítáme vzdálenost d koncového bodu tahu rozpoznávaného a koncového bodu tahu referenčního jako

$$d = \sqrt{(x_r - x)^2 + (y_r - y)^2} \tag{5.8}$$

   kde (x, y) jsou relativní souřadnice bodu rozpoznávaného tahu, (x_r, y_r) relativní souřadnice bodu tahu referenčního.

2. Pokud je vzdálenost větší než vhodně zvolená mezní vzdálenost, segmenty si neodpovídají a algoritmus končí.

3. Jinak pro každý kontrolní bod spočítáme podle uvedeného vzorce vzdálenost od každého vzorku v daném segmentu a vybereme ten vzorek, který je nejblíže.

4. Pokud pro každý kontrolní bod nalezneme takový vzorek, který je dostatečně blízko, segmenty označíme za podobné, v opačném případě nikoliv.

Ilustrace principu rozpoznání segmentu je na obrázku 5.3. Černé body označují vzorky, modré body jsou body referenčního tahu, červené úsečky zobrazují vzdálenosti, které musíme vypočítat. Vpravo pak vidíte detail s výpočtem jedné z těchto vzdáleností.

Obrázek 5.3: Rozpoznání segmentu na základě vzdáleností

Zbývá rozhodnout, jak vhodně zvolit mezní vzdálenost. Na základě experimentů jsem dospěl k závěru, že nelze příliš dobře stanovit tuto vzdálenost obecně, pro různé tahy (resp. segmenty) se může lišit (např. pro delší segmenty může být obecně větší, ale ovlivňují to i další faktory). Rozhodl jsem se tedy mezní vzdálenost zahrnout do popisu tahu, tedy každý koncový a kontrolní bod segmentu v referenčním tahu bude kromě svých souřadnic obsahovat navíc i tuto informaci.

Celý algoritmus rozpoznání tahu pak je následující:

1. Vzorkujeme dráhu, kterou uživatel opíše pohybem myši.

2. V získaných vzorcích nalezneme zlomové body na základě mezního úhlu, viz výše.

3. Ve zlomových bodech rozdělíme tah na segmenty a v rámci jednoho segmentu přepočítáme souřadnice bodů na relativní vzhledem k pozici počátečního bodu segmentu.

4. Porovnáme zjištěný počet segmentů s počtem segmentů referenčního tahu. Neshodují-li se, algortimus končí, resp. můžeme zkusit jiný referenční tah.

5. Pro každý segment provedeme porovnání algoritmem uvedeným výše.

6. Pokud se všechny segmenty podobají, tah je rozpoznán a algoritmus končí.

7. V opačném případě tah neodpovídá referenčnímu a algoritmus může také končit, nebo můžeme přejít k dalšímu referenčnímu tahu a pokračovat bodem 4.

Dosud uváděné postupy ovšem fungují pouze v případě, že celková velikost tahu, který uživatel nakreslí, je aspoň přibližně stejná jako velikost tahu referenčního. My ovšem chceme, aby bylo možné rozpoznat tahy nakreslené v (teoreticky) libovolné velikosti. Jak toho dosáhnout?

Existuje poměrně triviální řešení. Souřadnice rozpoznávaného tahu můžeme jednoduše transformovat tak, aby celková velikost odpovídala. Problém je, že dopředu nevíme, jaký poměr velikostí bude. Můžeme ale postupně procházet různé velikosti (zvolil jsem velikosti od jedné desetiny až po desetinásobek referenční) a pro každou z nich otestovat podobnost. Pokud najdeme podobnost tahu pro více možných poměrných velikostí, vybereme takovou velikost, pro kterou je shoda nejlepší.

5.2.2 Složení znaku z tahů

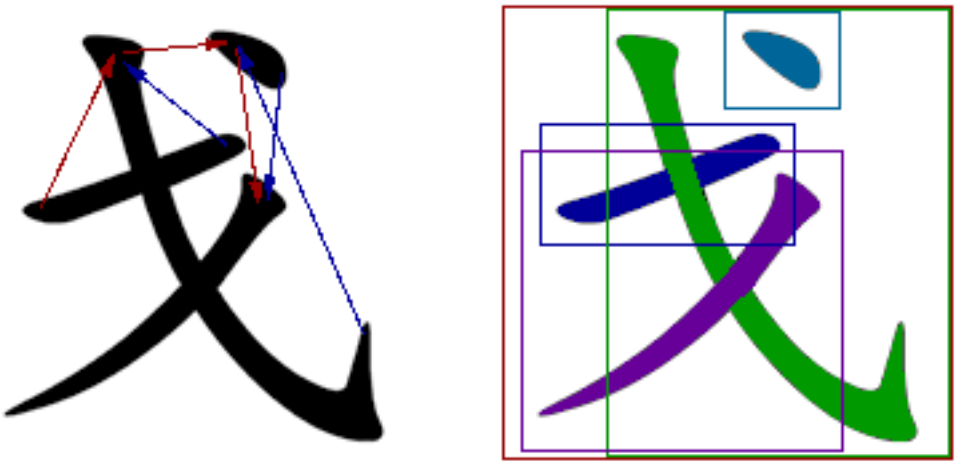
Musíme být schopni popsat a rozpoznat vzájemné vztahy mezi jednotlivými tahy ve znaku. Můžeme využít faktu, že pořadí tahů je pevně dané. Díky tomu lze stanovit pozici následujícího tahu vždy relativně k předcházejícímu. Poměrně snadno pak můžeme rozpoznávat znak už během jeho tvorby.

Kromě toho bychom měli zohlednit také absolutní pozici tahu ve znaku. Pokud znaku opíšeme myšlený čtverec, můžeme tento čtverec rozdělit do oblastí, ve kterých se musejí jednotlivé tahy vyskytovat. Tento přístup lépe odpovídá výše zmiňovanému principu radikálů. Problém může nastat, pokud chceme rozpoznávát znak již v průběhu psaní. Pak musíme buď mít předem definovanou velikost znaku (a jeho obrysového čtverce), nebo můžeme pouze odhadovat celkové rozměry na základě velikosti prvního tahu — to nemusí být příliš přesné.

Ilustrace obou postupů pro stejný znak najdeme na obrázku 5.4. Levý obrázek zachycuje jeden z možných postupů první metody, kdy vždy pozice začátku (červeně) a konce tahu (modře) určuje, kde má začínat následující tah. Na pravém obrázku vidíme druhý přístup, barevné obdélníky určují oblast, ve které se musí vyskytovat tah odpovídající barvy. Červeně je vyznačen obrysový čtverec celého znaku.

Pro vytvářený systém jsem se rozhodl zvolit čistě první variantu, tedy definování pozice tahu vždy relativně k tahu předchozímu. Přesněji, pozice tahu vždy bude určena počátečním bodem tahu. Souřadnice počátečního bodu budeme udávat relativně, vzhledem k pozici počátečního bodu tahu předchozího.

Při rozpoznávání znaku nejprve musíme zjistit, zda všechny nakreslené tahy odpovídají tahům referenčního znaku (zda jde o stejné tahy). Jelikož už umíme tahy jednotlivé tahy



Obrázek 5.4: Dvě možnosti popisu složení znaku

rozpoznat, tato část není problémem. Dále vypočítáme vzdálenosti počátečních bodů jednotlivých tahů od počáteních bodů tahů znaku referenčního (vše v relativních souřadnicích) podle známého vzorce

$$d = \sqrt{(x_r - x)^2 + (y_r - y)^2} \tag{5.9}$$

Pokud je některá z těchto vzdáleností větší než vhodně zvolená mezní vzdálenost, znaky si neodpovídají, v opačném případě ano. Opět na základě experimentů jsem mezní vzdálenost stanovil na 100 bodů (pro referenční znaky vytvořené v mřížce 400×400 bodů). Tato vzdálenost umožňuje rozpoznávat znaky s poměrně vysokou tolerancí, přitom však ještě musejí být z vizuálního hlediska dostatečně podobné.

Nestačí ale zjistit, zda jsou tahy správné a ve správných pozicích, ale také, zda jejich velikost je korektní. Při rozpoznávání jednotlivých tahů jsme již získali jejich poměrnou velikost, stačí ji tedy porovnat s danou velikostí tahů znaku referenčního. Rozdíl těchto velikostí musí pak být menší než vhodně zvolená mezní hodnota (experimentálně 30).

Pro rozpoznávání tedy budeme muset pro jednotlivé znaky uchovávat následující informace: kódy tahů, relativní pozice počtečních bodů tahů a poměrné velikosti tahů.

Algoritmus může vypadat takto:

1. Rozpoznáme jednotlivé tahy způsobem popsaným v předchozí podkapitole.
2. Pokud některé tahy neodpovídají tahům znaku referenčního (typ, resp. kód tahu), znaky se neshodují, algoritmus končí.
3. Jinak přepočítáme absolutní souřadnice počátečních bodů tahů na relativní (vždy vzhledem k počátečnímu bodu předcházejícího tahu; první tah znaku bude mít počátek v bodě 0, 0).
4. Určíme vzdálenosti počátečních bodů od počátečních bodů tahů referenčního znaku.
5. Pokud je některá vzdálenost větší než mezní hodnota, znaky si neodpovídají, algoritmus končí.

6. Provnáme poměrné velikosti tahů zjištěné při jejich rozpoznávání a poměrné velikosti tahů dané v referenčním znaku.

7. Jsou-li všechny rozdíly příslušných velikostí menší než mezní hodnota, znaky si odpovídají, v opačném případě nikoliv.

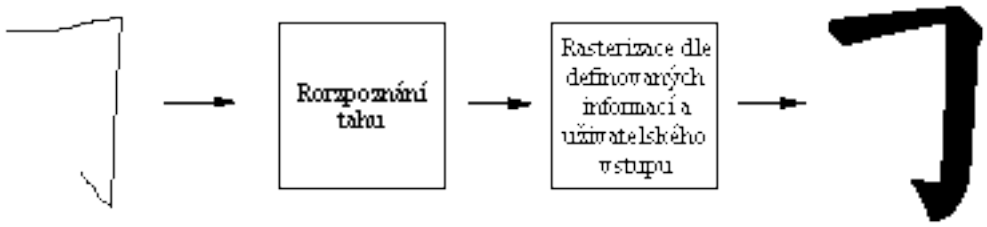
Podobně jako v případě tahů, při uvedeném řešení se setkáváme s problémem, pokud celková velikost znaku napsaného uživatelem je odlišná od velikosti znaku referenčního. Řešení může být také téměř identické (procházet různé velikosti a přepočítávat souřadnice).

5.3 Kaligrafie

V neposlední řadě je důležitá také grafická podoba aplikace. Chceme obsáhnout též kaligrafické hledisko *kanji*. Program by měl být schopen tahy, jež uživatel kreslí např. pohybem myši, schopen nejen rozpoznat, ale následně také zobrazit graficky v podobě evokující vzhled „skutečného“ psaní pomocí štetce.

Musíme tedy ke každému tahu v databázi uchovávat další informace popisující jeho grafickou podobu, jež pak při zobrazení použijeme. Přitom však při zobrazení tahu musíme zohlednit také to, jak sám uživatel tah nakreslil.

Schématické znázornění, jak může vypadat celý proces simulující kaligrafii, vidíme na obrázku 5.5. Zcela vlevo je příklad uživatelského vstupu, zcela vpravo výsledná podoba.



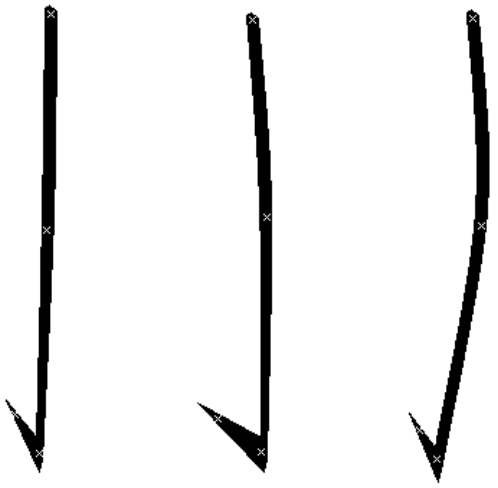
Obrázek 5.5: Proces simulující kaligrafii

Abychom mohli tah takto vykreslit, budeme potřebovat doplňující informace. Kromě dat pro rozpoznávání tedy bude třeba v referenčních tazích uchovávat ještě určitý návod pro vykreslení tahu. Jak tento návod realizovat?

Asi nejjednodušší by bylo připojit ke každému tahu např. bitmapu, kterou bychom vždy po rozpoznání tahu pouze zobrazili. Narazíme však hned na několik problémů. Bitová mapa je vhodná pouze pro konstantní velikost, ale my budeme chtít tahy vykreslovat v obecně různých velikostech. Navíc by nebylo možné zohlednit, jak uživatel tah nakreslil, výsledek by vždy vypadal stejně. Jako lepší možnost se jeví použít vektorového popisu grafiky. Nebudeme mít problém vykreslit tah v libovolné velikosti. Stále se ale nemáme možnost upravit vzhled podle uživatelského vstupu.

Způsob, který použijeme, je založen na vektorovém popisu. Nebudeme však používat pevné souřadnice, ale relativní souřadnice vůči bodům tahu, konkrétně vůči klíčovým bodům pro rozpoznávání tahu, tj. počáteční bod a koncový bod segmentu a kontrolní bodu v segmentu. Pro výsledné zobrazení pak nemusíme použít souřadnice pouze bodů z tahu referenčního, ale můžeme využít bodů z tahu rozpoznaného. Princip je znázorněn na obrázku 5.6. Vlevo můžete vidět, jak vypadá tah referenční, dále pak dvě alternativy vyobrazení

tahu na základě uživatelského vstupu. Bílými křížky jsou vyznačeny klíčové body. Je vidět, jak různé umístění techto bodů ovlivňuje výsledný vzhled tahu.



Obrázek 5.6: Varianty zobrazení tahu na základě vstupu uživatele

Vlastní popis grafiky je založen na tzv. obrysových bodech. Ke každému klíčovému bodu můžeme přiřadit několik obrysových bodů, jejichž souřadnice jsou specifikovány vzhledem k příslušnému klíčovému bodu. Při vykreslení proložíme obrysovými body křivku a případně ji vyplníme. Bude dále vhodné rozlišit dva druhy obrysových bodů, hladké a ostré. V hladkých bodech se obrysová linie neláme, prokládáme jimi tedy hladkou křivku. V ostrých naopak bude hrana. Pokud tedy narazíme při vykreslování na ostrý bod, ukončíme stávající křivku a začneme novou — když budeme mít více ostrých bodů bezprostředně za sebou, stačí je spojovat úsečkami.

Kapitola 6

Implementace

6.1 Uložení dat

Pro ukládání veškeré databáze jednotlivých tahů i znaků jsem se rozhodl používat soubory ve formátu XML. Důvodů je hned několik. V prvé řadě se jedná o standardizovaný formát, pro který existuje dobrá implementační podpora v různých prostředích, nebude tedy třeba nijak složitě řešit čtení nebo zápis těchto dokumentů. Další velkou výhodou je fakt, že informace jsou uloženy v textové podobě a tvaru, který je poměrně dobře čitelný pro člověka. Lze tedy takové soubory vytvářet a upravovat i ručně pomocí libovolného textového editoru. V neposlední řadě je pro nás výhodná také stromová struktura XML dokumentů, která dobře odpovídá tomu, jak skládáme jednotlivé tahy ze segmentů nebo znaky z tahů (viz výše).

6.1.1 Popis tahu

Celý seznam referenčních tahů je uložen v jednom XML dokumentu jako element *stroke_list*. Jednotlivé tahy jsou popsány elementy *stroke*. *Stroke* musí mít následující atributy:

- *code* – kód tahu (jednoznačně identifikuje tah)
- *segments* – počet segmentů, musí být celé číslo větší než nula

Element *stroke* obsahuje právě tolik segmentů, kolik udává jeho atribut *segments*. Segmenty zapisujeme jako elementy *segment* s atributy:

- *controls* – počet kontrolních bodů, musí být nezáporné celé číslo, může být nulový

V jednotlivých segmentech musí být vždy právě jeden element *start*, který popisuje počáteční bod, právě jeden *end* popisující koncový bod segmentu, a nejméně tolik elementů *control*, kolik je udáno v atributu *controls* příslušného segmentu (může jich být i více, ale pak jsou přebytečné zcela ignorovány). Všechny tyto elementy mají stejné atributy, s výjimkou elementu *start*, pro nějž by relativní souřadnice neměly smysl — jsou vždy 0, 0.

- x – relativní souřadnice bodu vůči počátku segmentu v ose x (vodorovná osa, hodnota se zvyšuje zleva doprava); celé číslo, může být záporné
- y – relativní souřadnice bodu vůči počátku segmentu v ose y (svislá osa, hodnota se zvyšuje shora dolů); celé číslo, může být záporné

- *range* – maximální možná odchylka bodu pro rozpoznání, nezáporné celé číslo
- *outlines* – jediný atribut, který má také element start; udává počet obrysových bodů, které přísluší k danému bodu, musí být nezáporné celé číslo

Každý element *start*, *end* nebo *control* musí obsahovat nejméně tolik elementů *outline*, jaká je hodnota jeho atributu *controls* (může i více, ale pak jsou přebytečné ignorovány, což může vést k nežádoucím výsledkům). *Outline* popisují obrysové body, které jsou významné pro vykreslení tahu. *Outline* již neobsahují žádné další elementy, a jejich atributy jsou:

- x – relativní souřadnice bodu vůči bodu, k němuž obrysový bod přísluší, v ose x (vodorovná osa, hodnota se zvyšuje zleva doprava); celé číslo, může být záporné
- y – relativní souřadnice bodu vůči bodu, k němuž obrysový bod přísluší, v ose y (svislá osa, hodnota se zvyšuje shora dolů); celé číslo, může být záporné
- *index* – celé číslo udávající pořadí bodu na obrysové linii, musí být unikátní pro každý element outline v rámci jednoho segmentu — neuškodí zdůraznit, že to je ovšem zároveň jediná podmínka, kterou na tyto hodnoty klademe. Může být výhodné číslovat body s určitým krokem (např. 10), aby pak bylo možné v případě potřeby jednoduše doplňovat další body mezi již popsané, aniž bychom je museli přečíslovat
- *type* – textový řetězec, jehož hodnota musí být buď „sharp“, nebo „smooth“, v prvním případě se jedná o bod, který bude proložen hladkou křivkou, ve druhém v něm dojde k ostrému zlomu

Následuje příklad, jak může vypadat popis jednoho tahu uvedeným způsobem. Na obrázku 6.1 pak vidíte, jak vypadá odpovídající tah vykreslený pomocí implementované aplikace. I na základě relativně jednoduchého popisu lze dosáhnout poměrně pěkných výsledků.

```
<stroke code="tpn" segments="2">
  <segment controls="0">
    <start outlines="2">
      <outline x="-2" y="-2" index="5" type="sharp"/>
      <outline x="2" y="2" index="60" type="sharp"/>
    </start>
    <end x="30" y="-30" range="10" outlines="0">
    </end>
  </segment>
  <segment controls="1">
    <start outlines="2">
      <outline x="0" y="0" index="10" type="sharp"/>
      <outline x="0" y="0" index="55" type="sharp"/>
    </start>
    <control x="25" y="25" range="20" outlines="2">
      <outline x="3" y="-3" index="20" type="smooth"/>
      <outline x="-3" y="3" index="50" type="smooth"/>
    </control>
    <end x="100" y="30" range="25" outlines="3">
      <outline x="8" y="-6" index="30" type="sharp"/>
      <outline x="-4" y="0" index="35" type="smooth"/>
```

```
      <outline x="-8" y="6" index="40" type="sharp"/>
    </end>
  </segment>
</stroke>
```

Obrázek 6.1: Vykreslení tahu na základě popisu v XML

6.1.2 Popis znaku

Jeden XML soubor obsahuje sadu znaků, označenou jako element *kanji_set*. Prvním elementem v něm musí být *header*, tzn. hlavička s informacemi o sadě. Sem spadá v prvé řadě název sady (*name*), který by měl být vždy přítomen. Dalším údajem je obtížnost znaků (*difficulty*), která může být popsána libovolným textovým řetězcem. Já jsem se ve svých sadách rozhodl používat číselné hodnocení obtížnosti, čím větší číslo, tím vyšší obtížnost). Poslední položkou je obecný popis sady (*description*), kam můžeme zapsat jakékoliv doplňující informace, např. popis, jaké typy znaků sada obsahuje, případně jméno autora apod. Příklad hlavičky následuje.

```
<header>
  <name>Numbers</name>
  <difficulty>1</difficulty>
  <description>Numbers from 1 to 10</description>
</header>
```

Za hlavičkou mohou následovat vlastní znaky. Teoreticky je množství znaků v jedné sadě zcela libovolné, prakticky však může být výhodnější nevytvářet sady příliš velké. Z mých zkušeností se jako nejlepší jeví používat sady o deseti až dvaceti znacích. Důvod bude patrný později, až si ukážeme vytvořenou aplikaci.

Každý znak popíšeme elementem *character*, který vždy musí obsahovat čtyři elementy. Prvním z nich je *on* s atributem *readings*. Zde zapíšeme jednotlivá čtení on (on'yomi), každé jako jeden element *reading*. Atribut *readings* udává počet těchto čtení. V případě elementu *kun* je situace obdobná, pouze s tím rozdílem, že popisuje čtení kun (kun'yomi). Pro zápis významů znaků nám slouží element *is* s atributem *meanings*, jež obsahuje elementy *meaning*.

Posledním prvkem je *writing*, s atributem *strokes*, který popisuje vzhled, resp. psaní znaku. Musí v něm být právě tolik elementů *stroke*, jaká je hodnota atributu *strokes*.

Stroke už žádné další elementy v sobě nemá, významné jsou jeho atributy:

- *code* – kód tahu
- *x* – relativní souřadnice počátku tahu vůči počátku tahu předchozího v ose x (vodorovná osa, hodnota se zvyšuje zleva doprava); celé číslo, může být záporné

- *y* – relativní souřadnice počátku tahu vůči počátku tahu předchozího v ose y (svislá osa, hodnota se zvyšuje shora dolů); celé číslo, může být záporné
- *ax* – absolutní souřadnice počátku tahu v ose x (vodorovná osa, hodnota se zvyšuje zleva doprava); nezáporné celé číslo
- *ay* – absolutní souřadnice počátku tahu y (svislá osa, hodnota se zvyšuje shora dolů); nezáporné celé číslo
- *size* – poměrná velikost tahu vůči tahu referenčnímu

První tah ve znaku musí mít vždy relativní souřadnice 0, 0.

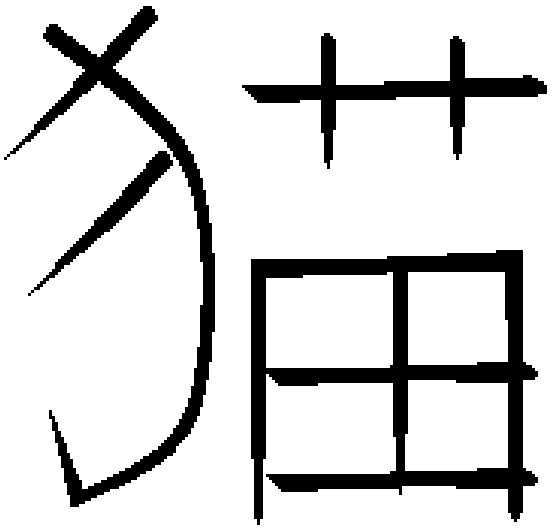
Následuje příklad zápisu znaku v XML, vygenerovaný vytvořenou aplikací. Jak takto popsaný znak vypadá po vykreslení v programu, můžete vidět na obrázku 6.2.

```
<character>
  <on readings="1">
    <reading>byou</reading>
  </on>
  <kun readings="1">
    <reading>neko</reading>
  </kun>
  <is meanings="1">
    <meaning>Cat</meaning>
  </is>
  <writing strokes="11">
    <stroke code="p" x="0" y="0" ax="100" ay="26" size="96" />
    <stroke code="wg" x="-66" y="12" ax="34" ay="38" size="315" />
    <stroke code="p" x="77" y="85" ax="111" ay="123" size="90" />
    <stroke code="h" x="55" y="-44" ax="166" ay="79" size="186" />
    <stroke code="s" x="53" y="-37" ax="219" ay="42" size="82" />
    <stroke code="s" x="86" y="2" ax="305" ay="44" size="75" />
    <stroke code="s" x="-133" y="148" ax="172" ay="192" size="169" />
    <stroke code="hz" x="0" y="2" ax="172" ay="194" size="170" />
    <stroke code="s" x="94" y="0" ax="266" ay="194" size="146" />
    <stroke code="h" x="-86" y="72" ax="180" ay="266" size="166" />
    <stroke code="h" x="2" y="71" ax="182" ay="337" size="164" />
  </writing>
</character>
```

6.2 Prostředky pro tvorbu aplikace

Pro realizaci samotného programu jsem si zvolil programovací jazyk C#, ve kterém píši spravovaný kód (managed code) pro platformu .NET Framework. Tento jazyk je založen na objektově orientovaném paradigmatu, které se snažím při implementaci co nejvíce dodržovat a využívat.

Hlavním nástrojem, který používám, je integrované vývojové prostředí Microsoft Visual Studio 2008. Mám již pomerně bohaté zkušenosti s programováním v jazyce Visual C++, ze kterého jazyk C# vychází. Proč jsem si tedy nevybral právě Visual C++? Zejména



Obrázek 6.2: Vykreslení znaku na základě popisu v XML

proto, že C# se mi jeví jako čistší jazyk, který je již přímo navržen pro platformu .NET, a programování v něm se mi zdá pohodlnější a také bezpečnější.

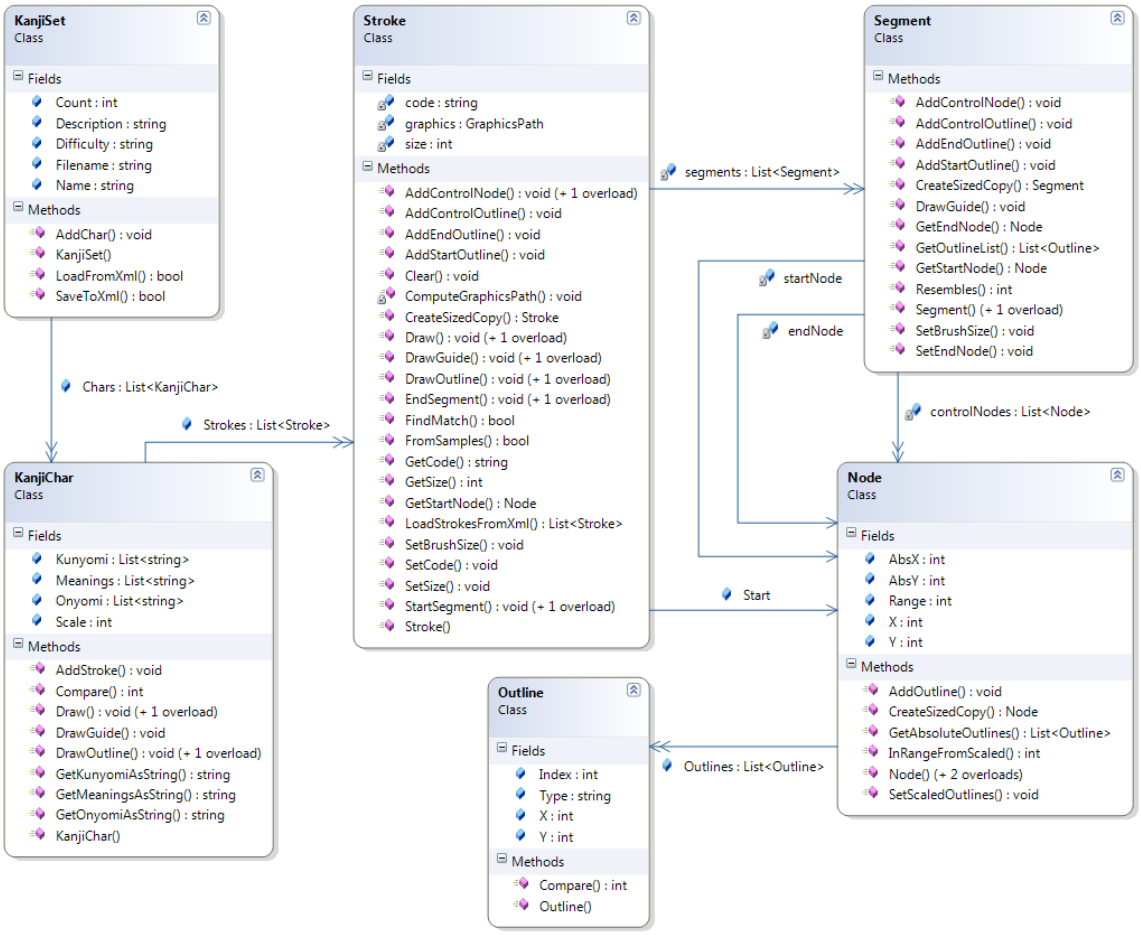
Proč ale vůbec tato platforma? Na první pohled je patrná hlavní nevýhoda. Takto napsaný program nebude přenositelný na jiné systémy. Tim však výčet nevýhod v podstatě končí. Oproti tomu výhody, alespoň z mého pohledu, převažují. Nejen pro programátora, kterému tato platforma poskytuje mnoho možností pro jednoduchou tvorbu zejména uživatelského rozhraní (ale také mnoho dalších užitečných funkcí). Myslím si, že i pro uživatele může být kladem např. fakt, že — díky tvorbě uživatelského rozhraní ze standarních komponent — mají programy v tomto prostředí jednotný vzhled (který navíc uživatel může ovlivňovat nastavením témat systému) a základní ovládání, a tak pro něj pravděpodobně nebude příliš obtížné naučit se pracovat s novou aplikací.

6.3 Práce se znaky *kanji*

Celý systém lze rozdělit do dvou základních relativně nezávislých částí, kterými jsou práce se samotnými japonskými znaky a uživatelské rozhraní aplikace. Nejprve se budu zabývat první jmenovanou částí.

Veškerý kód realizující tuto část se nachází v prostoru jmen *Kanji*. Vytvořil jsem několik tříd, které hierarchicky popisují znaky a poměrně přesně odpovídají popisu v kapitole 5. Kromě toho, že tyto třídy uchovávají výše zmíněné informace o znacích, resp. jejich složkách, poskytují také řadu operací pro práci s nimi. Můžeme je rozdělit do tří základních skupin: operace pro načítání a ukládání dat ze (resp. do) XML souborů, operace týkající se rozpoznávání tahů a znaků a operace, které slouží k vykreslení znaku na obrazovku (tuto část už můžeme považovat za poněkud propojenou s uživatelským rozhraním). Podrobněji si je rozebereme zvlášť u každé třídy.

Na obrázku 6.3 můžete vidět diagram tříd, vytvořený částečně automatizovaně nástrojem Microsoft Visual Studio 2008.



Obrázek 6.3: Diagram tříd implementujících práci s japonskými znaky

6.3.1 Třída *KanjiSet*

Třídy jsou, jak již bylo zmíněno, uspořádány hierarchicky. Proberme je nyní postupně od nejvyšší úrovně. Zde se nachází třída *KanjiSet*. Jedná se o množinu znaků *kanji*, která obsahuje kromě samotného seznamu znaků také metodu *LoadFromXML()*, která načítá sadu znaků z XML dokumentu (formát viz výše), a metodu *SaveToXML()*, která naopak celou sadu do XML dokumentu uloží. Pro prácí s XML soubory využívám funkcí prostředí .NET, kde v prostoru jmen *System.XML* najdeme především třídy *XMLReader* a *XMLWriter*, pomocí nichž lze snadno dokumenty číst, resp. zapisovat. Bližší informace viz [4].

Pro seznam znaků samotný je použita generická třída *List* ze standarního prostoru jmen *System.Collections.Generic*, která se bude dále objevovat ještě poměrně často. Jedná se o kolekci objektů daného typu (v tomto případě tímto typem je třída *KanjiChar*, viz níže). Je vhodné zde zdůraznit, že jméno třídy *List* (tzn. seznam) je v tomto případě poněkud zavádějící. Jedná se totiž o typ, který bychom asi spíše označili jako dynamické pole, neboť je zde možnost rychlého náhodného přístupu k jednotlivým položkám na základě indexu. Ve zmíněném prostoru jmen existuje také mj. třída *LinkedList*, která odpovídá tomu, co běžně známe jako abstraktní datový typ seznam.

6.3.2 Třída *KanjiChar*

Následuje třída *KanjiChar*, která odpovídá jednomu japonskému znaku. Údaje o čteních a významech znaku jsou uchovávány jako *List* textových řetězců (*string*), označené jako *Onyomi* (čtení *on*), *Kunyomi* (čtení *kun*) a *Meanings* (významy). Kromě toho obsahuje tato třída ještě atribut *Strokes* typu *List(Stroke)*, který zahrnuje tahy, z nichž se znak skládá.

Třída *KanjiChar* poskytuje metodu *Compare()*, kterou lze znak porovnat s jiným (s modelem). Pokud si znaky zcela odpovídají, tj. všechny jejich příslušné tahy jsou si dostatečně podobné, vrací metoda hodnotu 0, naopak pokud se již shodovat nemohou, vrátí -1. Kromě toho ale může vrátit také nezáporné celé číslo v případě, že znak, jehož metoda *Compare()* je volána, by mohl odpovídat znaku zadanému jako parametr, ale ještě by musely být doplněny další tahy. Návratová hodnota metody pak udává počet chybějících tahů.

Pro vykreslení znaku slouží metody *Draw()*, *DrawOutline()* a *DrawGuide()*. Prvním parametrem je vždy grafický objekt (typ *Graphics*), do kterého se má kreslit, druhým pak štetec (*Brush*) nebo pero (*Pen*), pomocí nichž bude znak vykreslen. *Draw()* vykreslí znak celý včetně výplně, *DrawOutline()* pouze obrysové linie jednotlivých tahů. Základní princip spočívá v tom, že tyto metody zavolají postupně *Draw()*, resp. *DrawOutline()* každého tahu s příslušnými parametry.

Metoda *DrawGuide()* je specifická v tom, že nevykreslí celý znak, ale pouze nápovědu, jak by měl vypadat následující tah. Aby to bylo možné, potřebuje dostat navíc jako parametr model znaku — tím je opět objekt třídy *KanjiChar* — z nějž získá potřebné informace o daném tahu. Před prvním voláním této metody, resp. po každé modifikaci znaku, je nutné znak porovnat s modelem výše zmíněnou metodou *Compare()*. Pokud by *Compare()* vrátila hodnotu -1, nesmí metoda *DrawGuide()* být volána, její výsledek je pak nedefinovaný. Kromě toho se při provádění metody *Compare()* a úspěšném nalezení shody nastaví odpovídající velikost znaku, kterou musí mít metoda *DrawGuide()* k dispozici, aby dokázala nápovědu vykreslit na správné pozici a se správnými rozměry.

6.3.3 Třída *Stroke*

Třída *Stroke* reprezentuje jeden tah. Kromě attributu *code* (kód), který jednoznačně udává, o který z dostupné množiny tahů se jedná, obsahuje tato třída opět také poměrnou velikost tahu vůči referenčnímu. Dále uchovává seznam segmentů, opět s využitím třídy *List*. Posledním atributem je *graphics* obsahující informace, jak se daný tah vykreslí. Konkrétně jde o tzv. grafickou cestu, tj. objekt třídy *GraphicsPath* z prostoru jmen *System.Drawing.Drawing2D* (viz [3]). Výhodou je, že stačí tuto cestu sestavit vždy pouze jednou při rozpoznání tahu, a při samotném vykreslování už jenom využít standardní metody *DrawPath()* třídy *Graphics*.

Stroke nabízí podobně jako *KanjiChar* metody *Draw()*, *DrawOutline()* a *DrawGuide()*, které vykreslí tah, resp. jeho obrysovou linii nebo návod pro uživatele, jak tah provést.

Třídní (statická) metoda *LoadStrokesFromXML()* slouží k načtení sady tahů z XML souboru. Opět využívá služeb třídy *XMLReader* ze *System.XML*.

Pravděpodobně nejvýznamnějšími funkcemi jsou metody související s rozpoznáváním uživatelem nakreslených tahů. První je *FromSamples()*, jenž dokáže z navzorkovaného tahu myši sestavit obecný tah (zatím nemusí odpovídat žádnému tahu z referenční množiny). Vzorky tvoří uspořádanou množinu bodů, tj. *List* instancí třídy *Point* ze *System.Drawing*. V této metodě probíhají dvě hlavní činnosti. Nejprve je třeba nalézt zlomové body, které tah rozdělí do segmentů. Jedná se o relativně jednoduchý výpočet úhlu svíraného dvěma vektory, pokud je tento úhel větší než daný mezní úhel, je detekován zlom. Dále je třeba přepočítat absolutní souřadnice bodů na relativní (vždy vztažené k počátečnímu bodu segmentu, který má relativní souřadnice 0, 0).

Když takto získáme obecný tah, potřebujeme zjistit, zda tento tah odpovídá některému z referenčních tahů. K tomu slouží metoda *FindMatch()*, jejímž parametrem je právě množina referenčních tahů. Pokud metoda nalezne tah, jež s danou tolerancí vyhovuje, nastaví příslušný kód a poměrnou velikost. Nakonec pak vypočítá a sestaví grafickou cestu na základě informací z referenčního tahu upravených vzhledem k pozici kontrolních bodů tahu nakresleného uživatelem. Návratová hodnota je v tomto případě *true*. Pokud metoda nenalezne žádný vhodný referenční tah, vrací *false*.

Zjištení podobnosti tahu probíhá tak, že postupně pro každý kandidátní referenční tah nejprve ověříme, zda se shoduje počet segmentů, pokud ne, přejdeme k dalšímu tahu. Shoduje-li se počet, porovnáme jednotlivé segmenty. K tomu je použita metoda *Resembles()* třídy *Segment*. Pokud se kterýkoliv segment neshoduje, přejdeme k následujícímu tahu.

Možnost různých velikostí tahů řeším poměrně jednoduchým způsobem. Metoda *Resembles()* má jako jeden z parametrů poměrnou velikost v procentech, pro kterou se má porovnání provést. V cyklu projdu velikosti od jedné desetiny až po destinásobek referenční (provádí se vždy tisíc iterací). Ačkoliv toto řešení není zjevně příliš efektivní, vzhledem k tomu, že rozpoznávání se provádí na základě uživatelského vstupu, který je obecně poměrně pomalý, není zde časová náročnost kritická. Z provedených experimentů vyplynulo, že řešení dává dobré výsledky a na všech testovaných systémech pracovalo bez uživatelsky zaznamenatelného zpoždění.

Původně jsem používal určitou optimalizaci uvedeného řešení, která spočívala v tom, že jakmile byla pro některou poměrnou velikost nalezena shoda všech segmentů, algoritmus mohl skončit. Ukázalo se však, že takové řešení není příliš vhodné. První takto nalezená velikost totiž obvykle nebude optimální, jiank řečeno, pro odlišnou velikost může být shoda ještě lepší. Ve finální verzi tedy vždy procházím všechny velikosti a pro takové, pro něž byla

shoda nalezena, si uchovávám velikost odchylky. Nakonec pak vyberu velikost, pro kterou je odchylka minimální.

Pokud je rozpoznání tahu úspěšné, tato metoda na závěr sestaví grafickou cestu (aby nebylo nutné počítat ji při každém vykreslování znovu). K tomu účelu zavolá metodu *ComputeGraphicsPath()*, která sestavení provede. Metoda *ComputeGraphicsPath()* nejprve získá pozice obrysových bodů a uspořádá tyto body podle jejich indexů. Z výsledné uspořádané množiny pak vytvoří cestu pomocí standardních metod třídy *GraphicsPath AddLine()* a *AddCurve()*.

6.3.4 Třída *Segment*

Atributy tříde *Segment* zahrnují počáteční bod, koncový bod, a množinu kontrolních bodů (opět třída *List*). Původně jsem uvažoval pro tyto body použít standardní třídu *Point* z prostoru jmen *System.Drawing*. Záhy jsem však zjistil, že bude výhodnější nadefinovat třídu vlastní, kterou jsem nazval *Node* (uzel).

Nejvýznamnější metodou třídy *Segment* je již zmíněná funkce *Resembles()*. Tato funkce porovná segment tahu s jiným, kandidátním segmentem. Pokud si segmenty odpovídají s danou tolerancí, vrátí velikost odchylky, pokud ne, je návratovou hodnotou -1. Velikost odchylky se v tomto případě určuje jako největší vzdálenost mezi bodem segmentu a odpovídajícím bodem kandidátního segmentu.

Porovnání využívá metody třídy *Node*, kterou jsem nazval *InRangeFromScaled()*. Tato metoda vrací -1, pokud jsou body příliš daleko od sebe (tj. jejich vzdálenost je větší než daná mezní hodnota), jinak vrátí zjištěnou vzdálenost. Jedním z parametrů je i poměrná velikost (scale), *Resembles()* pouze tuto hodnotu předá. Nejprve jsou porovnány relativní pozice koncových bodů. Pokud souhlasí, je třeba nalézt ještě odpovídající kontrolní body. Původní řešení vždy jako nalezený kontrolní bod považovalo první, který byl dostatečně blízko referenčnímu. Opět se ukázalo, že tento způsob není vhodný, protože často by bylo možné najít jiný bod, který by odpovídal lépe. Je tedy třeba vždy projít všechny body a vybrat ten, který je nejbližší. Pokud se podaří nalézt dostatečně blízké body ke všem referenčním kontrolním bodům, segmenty se podobají, v opačném případě nikoliv.

Pro rozpoznání bychom již zde mohli skončit, ale kvůli vykreslení je ještě třeba upravit pozice obrysových bodů vzhledem k nalezením kontrolním bodům (jinak by vykreslení nezohledňovalo, jak uživatel tah nakreslil).

6.3.5 Třída *Node*

Třída reprezentující bod, či přesněji řečeno uzel. Může uchovávat informace o relativní a absolutní pozici bodu v rámci tahu — relativní je vždy vztažena k začátku segmentu, absolutní udává pozici na kreslicí ploše. Dalším atributem je *range*, tj. maximální povolená vzdálenost pro rozpoznání bodu. Nakonec *Node* obsahuje ještě seznam obrysových bodů (*List* objektů typu *Outline*), který je důležitý pro vykreslení tahu na obrazovku.

Metoda *InRangeFromScaled()* přepočítá souřadnice bodu podle daného měřítka a pak porovná s referenčním bodem. Pokud je vzdálenost těchto bodů větší než hodnota atributu *range*, vrací -1, jinak vrátí danou vzdálenost.

6.3.6 Třída *Outline*

Poslední třída *Outline* odpovídá obrysovému bodu. Kromě souřadnic a indexu bodu obsahuje také metodu *Compare()* pro porovnání dvou instancí této třídy na základě hodnot

jejich indexů. Této metody se využívá pro seřazení obrysových bodů tahu.

6.4 Vlastní aplikace, uživatelské rozhraní

Třídy realizující samotnou aplikaci jsou implementovány v prostoru jmen *KanjiGakuen* (název aplikace). Jedná se především o standardní formuláře pro platformu .NET. Uživatelské rozhraní je vytvořeno z prefabrikovaných komponent pomocí WYSIWYG editoru, který je součástí nástroje Visual Studio 2008, tvorba zdrojového kódu je tedy z velké části automatizovaná. Je třeba pouze doplnit obsluhu událostí (tj. např. jaké akce se mají provést po stisku určitého tlačítka formuláře).

Celkem program obsahuje sedm formulářů, které odpovídají oknům aplikace. Jde o následující třídy:

- *Form1* – popisuje hlavní okno aplikace, které se objeví po spuštění programu. Na základě událostí hlavního okna pak mohou být otevírána okna ostatní
- *FrmLearn* – slouží pro výuku znaků, zobrazuje informace o znaku a umožňuje kreslení tahů
- *FrmPractice* – testování získaných znalostí, toto okno zobrazuje otázky (jejich součástí může být i žádost o napsání znaku)
- *FrmPractRes* – výsledky testu, hodnocení
- *FrmEditor* – formulář pro tvorbu nebo editaci jednoho znaku, umožňuje kreslení
- *FrmSetEdit* – editace sady japonských znaků, úprava informací o sadě, přidávání nebo odebírání znaků
- *FrmCreateSet* – slouží k vytvoření nové sady znaků

Pro práci s japonskými znaky — jejich načítání a ukládání, rozpoznávání i vykreslování — aplikace využívá tříd definovaných v předchozí podkapitole. Bylo třeba přidat ještě třídu, která umožňuje vygenerování testu pro formulář *FrmPractice*. Jde o třídu *PracticeList*, jež definuje seznam otázek a obsahuje metody pro pseudonáhodné vytvoření otázek z dané sady znaků.

Kapitola 7

Závěr

7.1 Dosažené výsledky

V rámci semestrálního projektu jsem prostudoval problematiku japonského písma a pokusil se v tomto textu obsáhnout nejdůležitější informace. Seznámil jsem se s některými existujícími aplikacemi pro výuku japonských znaků. Na základě získaných zkušeností jsem se poté snažil formulovat požadavky na takový systém. Rozebral jsem některé problémy, kterými se budu muset v souvislosti s danou problematikou dále zabývat, a navrhl možná řešení.

Vybraná řešení jsem blíže rozebral a posléze implementoval. V průběhu implementace došlo i k některým úpravám oproti teoretickému návrhu, když se ukázalo, že původně navržené řešení nemusí být nejvhodnější. Obvykle ale nebyly takové úpravy příliš výrazné.

Některé ukázky z výsledné aplikace můžete najít v uživatelské příručce (příloha A).

7.2 Možnosti dalšího vývoje

Možnosti dalšího vývoje systému jsou poměrně široké. V prvé jde o vytváření dalších sad japonských znaků. Dosud byla dosud popsána pouze poměrně malá část všech běžne používaných znaků *kanji*.

Doplnění a úpravy by bylo možné provést i v databázi tahů. Současný popis se zatím zdá být pro rozpoznávání povětšinou dostatečný, ale v budoucnu se může stát, že pro některé znaky budeme potřebovat definovat tahy nové. V případě stávajících tahů bychom mohli dále pracovat především na jejich grafické podobě. Lepší vizuální kvality bychom dosáhli zejména použitím více obrysových bodů.

Aplikace samotná lze dále rozvíjet přidáváním nových funkcí. Užitečné by mohly být např. pokročilejší možnosti editace znaků a sad. Tématem k zamyšlení je také např. zařazení uživatelských profilů, aby program mohl uchovávat informace o tom, které znaky nebo sady už se uživatel naučil, jak si vedl v předchozích testech apod.

Z hlediska uživatelského rozhraní můžeme dále zkoumat, jak program udělat uživatelsky co nejpřívětivějším — důraz klademe nejen na jednoduchost a srozumitelnost ovládání, ale také na prezentaci informací při výuce. Je třeba, aby se uživatel pomocí programu mohl učit znaky co nejefektivněji.

Literatura

[1] Abe, N.: All About Radicals - Japanese Language [online].
http://japanese.about.com/library/weekly/aa070101a.htm, 2006-11-08 [cit. 2009-01-04].

[2] Bishop, T.; Cook, R.: Character Description Language (CDL): The Set of Basic CJK Unified Stroke Types [online].
http://www.wenlin.com/cdl/cdl_strokes_2004_05_23.pdf, 2004-05-23 [cit. 2009-01-03].

[3] Microsoft: System.Drawing Namespace () [online].
http://msdn.microsoft.com/en-us/library/system.drawing.aspx, [cit. 2009-05-23].

[4] Microsoft: XML Documents and Data [online].
http://msdn.microsoft.com/en-us/library/2bcctyt8.aspx, [cit. 2009-05-23].

[5] Miller, A. R.: Kanji Stroke Order [online].
http://infohost.nmt.edu/~armiller/japanese/strokeorder.htm, [cit. 2009-01-03].

[6] WWW stránky: Stroke (CJK character) [online].
http://en.wikipedia.org/wiki/Stroke_(CJK_character), [cit. 2009-01-03].

[7] WWW stránky: Computer font [online].
http://en.wikipedia.org/wiki/Computer_font, [cit. 2009-05-22].

[8] WWW stránky: Yokozuna! 1.1 [online].
http://www.brothersoft.com/yokozuna-39387.html, [cit. 2009-05-22].

Příloha A

Uživatelská příručka aplikace

A.1 Instalace a spuštění

Aplikaci není nutné nijak instalovat, lze spustit přímo z CD. Je ale doporučeno zkopírovat program nejprve na pevný disk (případně jiné zařízení, kde máte možnost zápisu), jinak nebude možné přidávat nové sady znaků nebo je upravovat.

Aplikace je určena pro operační systém Microsoft Windows (testováno na XP a Vista). K jejímu spuštění je třeba mít nainstalované prostředí .NET Framework (verze nejméně 2.0, nejlépe však nejnovější), které majitelé Windows mohou bezplatně získat z oficiálních stránek společnosti Microsoft.

K dispozici jsou rovněž zdrojové soubory v jazyce C#. Jedná se o projekt vývojového prostředí Microsoft Visual Studio 2008, k překladu je tedy třeba použít tento nástroj (popř. jeho novější verzi).

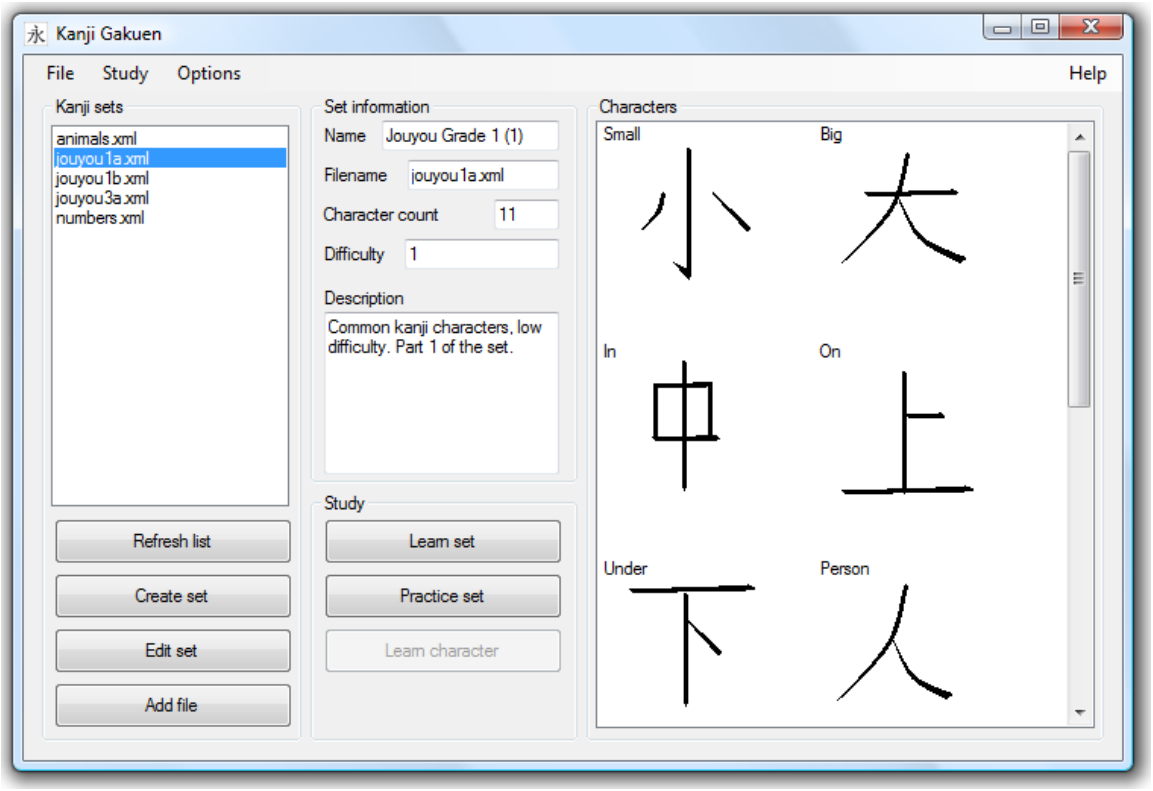
A.2 Použití programu

Po spuštení programu uvidíte obrazovku zachycenou na obrázku A.1. Nejprve v seznamu v levé části vyberte sadu znaků. Ve střední části se vám zobrazí informace o vybrané sadě, vpravo pak seznam znaků, které sada obsahuje (konkrétně bude vyobrazena grafická podoba znaku a první z jeho významů).

A.3 Výuka

Pokud se chcete začít učit postupně znaky ve vybrané sadě, stačí nyní stisknout tlačítko *Learn set* (také můžete použít menu *Study* v horní části okna). Poté se vám zobrazí okno viz obrázek A.2. Zde si přečtěte pozorně jednotlivá čtení a významy znaku, a poté překreslete znak do okna v pravé části. Aplikace bude kontrolovat, zda děláte tahy správně, pokud ne, tah nebude přijat a musíte jej zkusit znovu.

Znak můžete kreslit libovolně velký a začít na libovolné pozici, ale nedoporučuje se kreslit znak příliš malý nebo příliš velký; také si dejte pozor, aby se vám celý znak vešel na plátno (okno můžete v případě potřeby zvětšit). Tlačítko *Hint* vám ukáže, jak byste měli provést další tah. Pomocí *Undo* a *Clear* vrátíte zpět poslední nakreslený tah, resp. smažete všechny tahy a můžete začít znovu. Tlačítky *Next* a *Previous* se můžete přesunou na následující (resp. předchozí) znak v sadě. Výuku kdykoliv ukončíte pomocí *Close*. Parametr *Brush size*



Obrázek A.1: Úvodní obrazovka aplikace

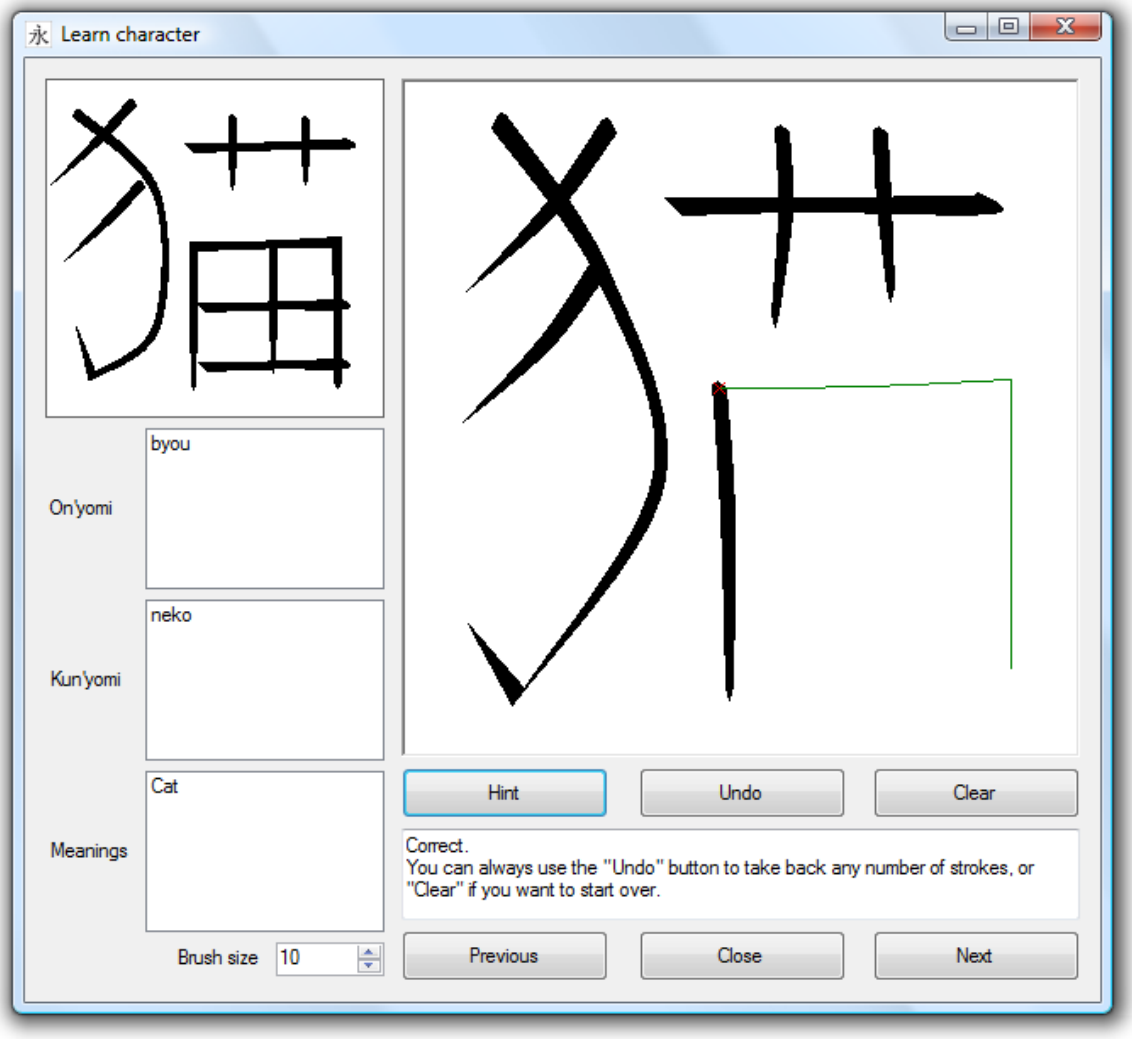
udává velikost štětce — jedná se čistě o estetickou záležitost. Pokud chcete psát znak velký, může být vhodné použít větší štětec, a naopak.

Chcete-li se učit konkrétní znak z dané sady, vyberte jej v seznam vpravo a stiskněte *Learn character*. Opět lze využít také menu *Study*, také stačí poklepání (dvojklik, doubleclick) na zvolený znak. K dispozici máte i kontextové menu, které vyvoláte stiskem pravého tlačítka.

A.4 Ověření znalostí

Pokud si přejete otestovat své znalosti, vyberte sadu znaků a stiskněte tlačítko *Practice set* (nebo použijte menu *Study*). Objeví se jedna z obrazovek zachycených na obrázcích A.3 a A.4, případně ještě jiná, podobná první z nich. Celkem aplikace nabízí tři možnosti otázek. V prvním případě (obrázek A.3) vám zobrazí všechna čtení znaku (on'yomi i kun'yomi). Vaším úkolem je znak správně napsat a z pěti možností vybrat jeho správné významy. Podobná je otázka, kdy vám jsou řečeny významy, a vy máte vybrat správná čtení a znak také nakreslit. V třetím typu otázek (obrázek A.4) dostanete napsaný znak a opět z pěti možností musíte vybrat jeho správná čtení i významy.

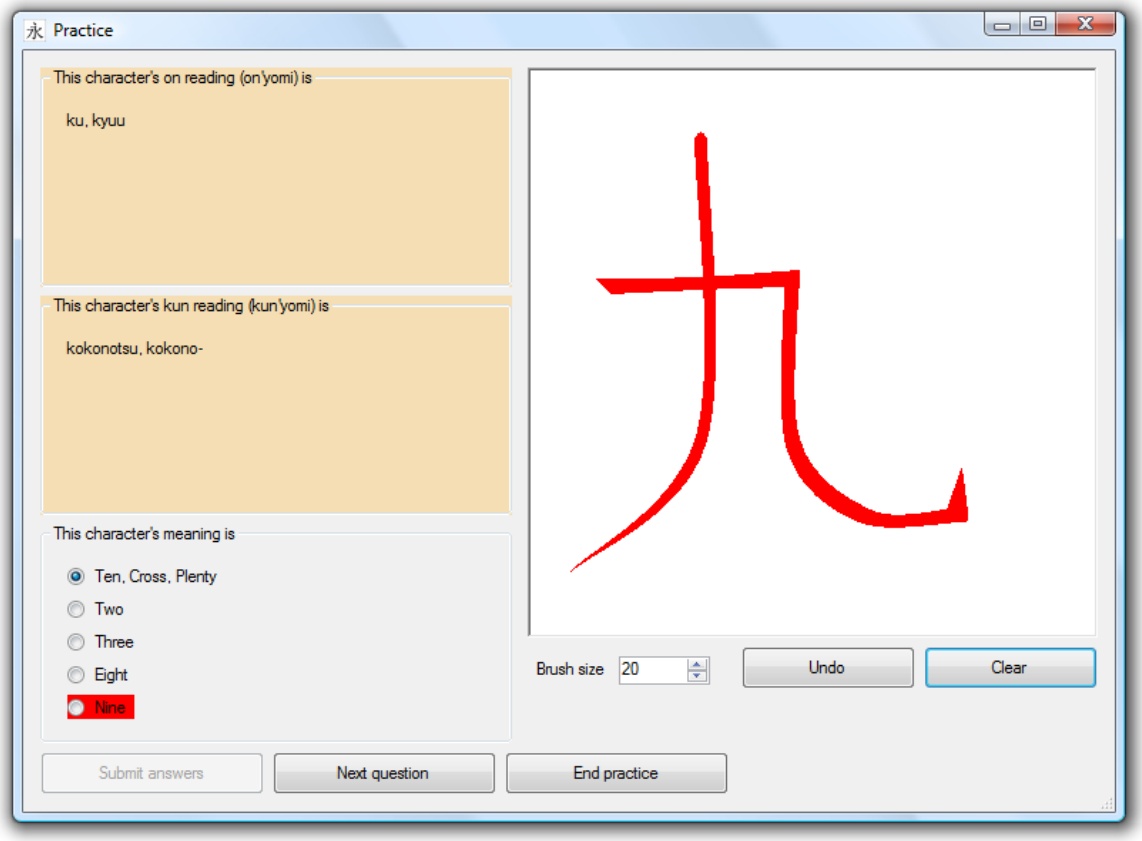
Jakmile máte vybrány všechny odpovědi, případně napsán celý znak, stiskněte *Submit answers*. Aplikace vaše odpovědi vyhodnotí, zeleně označí správné, v případě špatných vám červeně vyznačí správnou odpověď. Pokud jste znak napsali správně, také jej označí zeleně, jinak vám červeně ukáže správnou podobu znaku. Poté můžete přejít k další otázce tlačítkem *Next question*. Přejete-li si testování ukončit, stiskněte *End practice*. Tato tlačítka je ovšem



Obrázek A.2: Okno výuky znaků

možné použít vždy pouze po odeslání odpovědí. Testování také skončí automaticky, jakmile vyčerpáte všechny otázky.

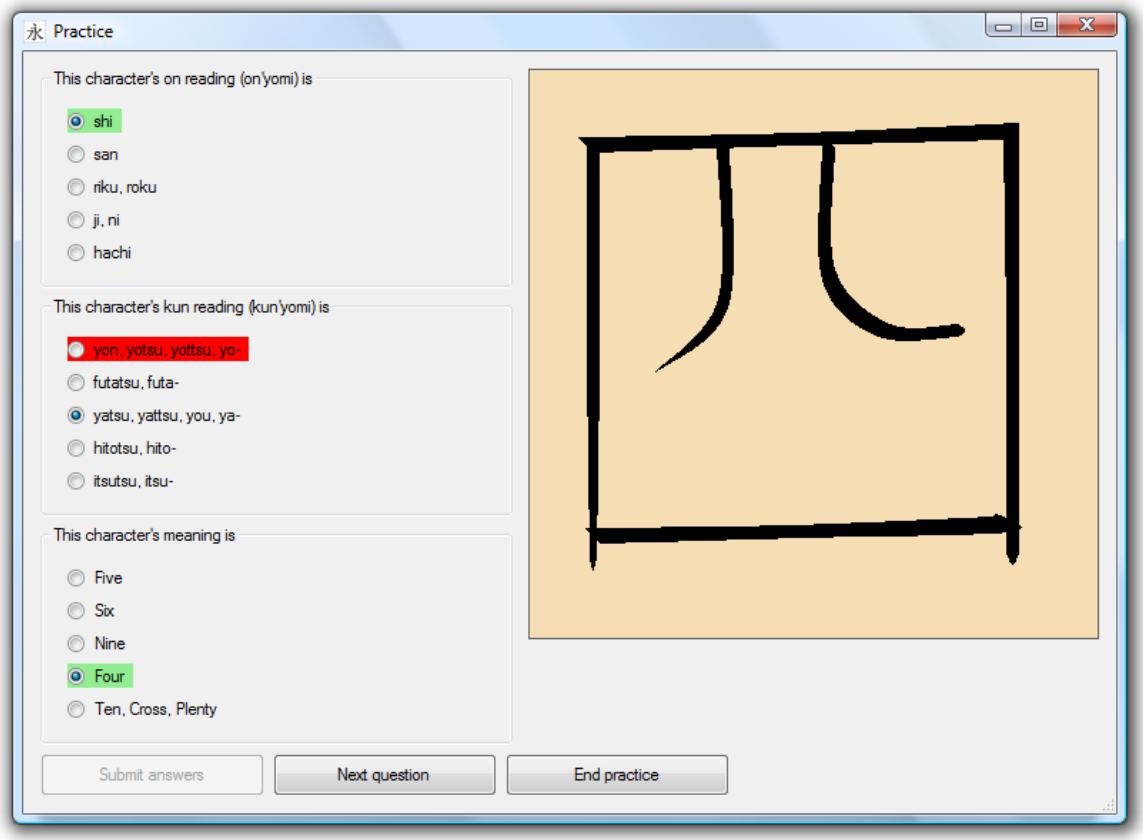
Nakonec uvidíte statistiku, jak jste si vedli. První položka je celkový počet otázek (resp. podotázek, každé okno má dvě nebo tři), druhá počet správných odpovědí, třetí pak vaše procentuální úspešnost. Na základě té obdržíte hodnocení A až F, odstupňované po deseti procentech. Pokud jste neudělali vůbec žádnou chybu, můžete získat i zvláštní známku.



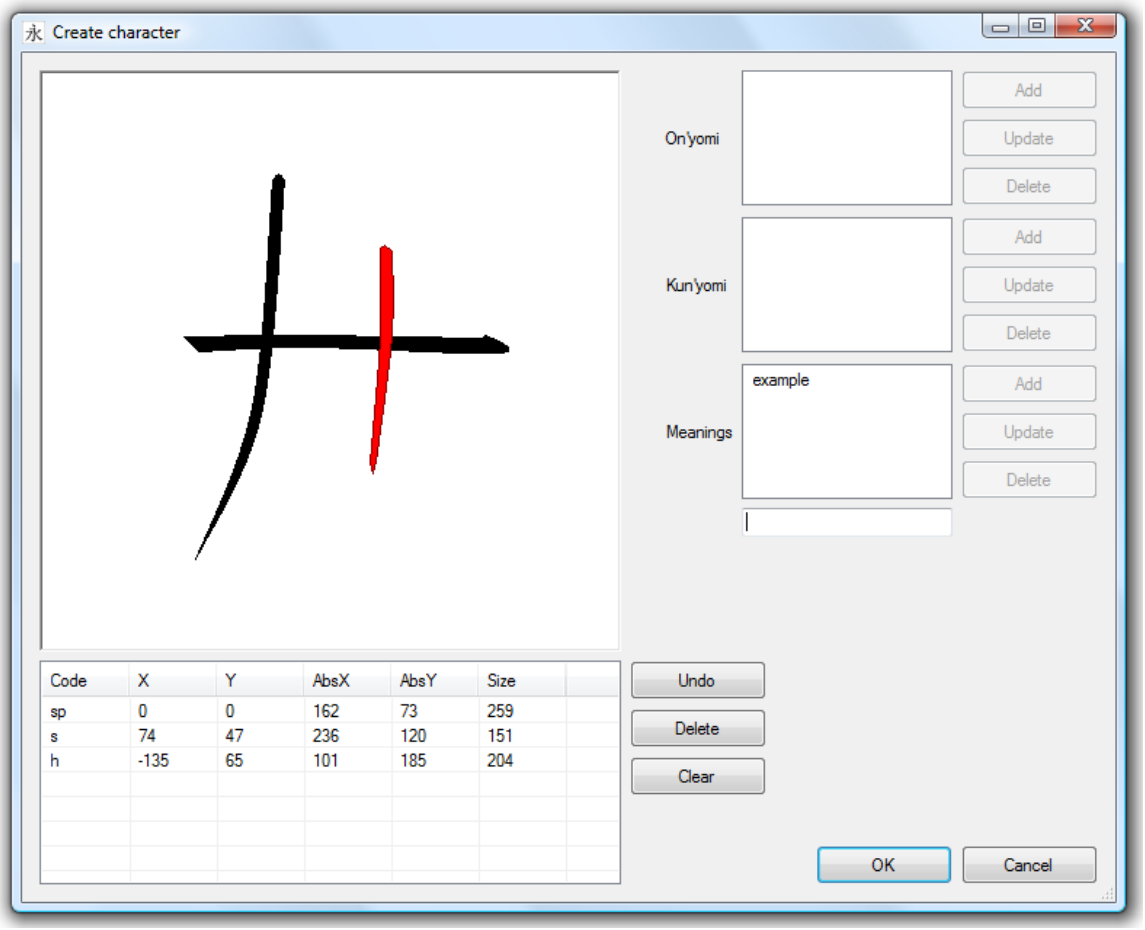
Obrázek A.3: Ověřování znalostí

A.5 Editace sad znaků

V hlavním okně najdete dále zatím nezmiňovaná tlačítka *Edit set* a *Create set*, pomocí kterých můžete otevřít okno pro úpravu, resp. tvorbu sad znaků. Zde můžete editovat informace sad a přidáváa nebo odebírat znaky (*Create character*, *Delete character*). Tlačítko *Create character* zobrazí okno zachycené na obrázku A.5. Můžete pak upravovat a dpolňovat čtení a významy, a také nakreslit znak. Tlačítkem *OK* znak uložíte.



Obrázek A.4: Ověřování znalostí - další varianta



Obrázek A.5: Obrazovka tvorby znaku