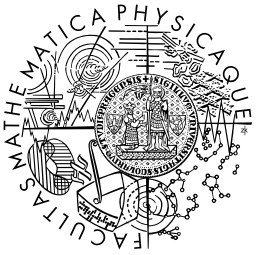


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Jan Hadrava

Vektorizace čárové grafiky

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce:	RNDr. Josef Pelikán
Studijní program:	Informatika
Studijní obor:	Obecná informatika

Praha 2016

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Rád bych poděkoval všem, kteří mě při tvorbě práce podporovali. Především svému vedoucímu, RNDr. Josefu Pelikánovi, který to se mnou úspěšně zvládl. Dále pak děkuji rodině a přátelům, speciálně Péťe Pelikánové za obrázky hrochů a Kačce Zákravské za korektury a obrovskou podporu.

Název práce: Vektorizace čárové grafiky

Autor: Jan Hadrava

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Josef Pelikán, Katedra softwaru a výuky informatiky

Abstrakt: I při tvorbě grafiky se některým tvůrcům lépe pracuje s tužkou a papírem. Je žádoucí vzniklou skicu zdigitalizovat (naskenovat, vyfotografovat) a následně upravovat v počítači. K tomu je užitečné převést obrázek do vektorového formátu – zvektorizovat.

Vektorová reprezentace obrázků poskytuje oproti rastrové mj. dobrou kvalitu i při libovolném zvětšení či snazší editaci. Každý element je reprezentován buď jako křivka, či jako vyplněná oblast definovaná svým obvodem. U čárových kreseb je pro následnou práci s obrázkem vhodnější první z uvedených.

Současné vektorizační nástroje nejčastěji hledají pouze souvislé plochy. Některé z nich jsou i volně dostupné. Programů reprezentujících výstup pomocí čar existuje podstatně méně a často jsou také velmi drahé.

Práce navrhuje vektorizační algoritmus a volně šiřitelný program, jenž vektorový obrázek reprezentuje pomocí čar. Po předzpracování rastrového obrázku je nalezena jeho (stále rastrová) morfologická kostra, která je trasována a převedena do vektorové podoby na Bézierovy křivky. Následně je obrázek vyhlazen a vyexportován do formátu zvoleného uživatelem. Kvalita výstupu je v mnohých ohledech srovnatelná s autorovi dostupnými vektorizačními nástroji.

Klíčová slova: vektorizace, čárová grafika, analýza obrazu, morfologické operace, Bézierovy křivky

Title: Vectorization of line-based images

Author: Jan Hadrava

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Josef Pelikán, Department of Software and Computer Science Education

Abstract: Some creators prefer working with pen and paper while creating graphic art. It is desirable to digitize a draft (scan it, photograph) and edit it on a computer afterwards. It is useful to convert an image to a vector format – to vectorize it.

Vector representation of images gives us good quality in any zoom level and enables easier editing compared to a raster workflow. Each element is represented as a curve, or as a filled area defined by its outline. Line-based images may be edited very conveniently and efficiently.

Contemporary vectorization tools usually search for connected areas. Some of them are also freely available. There exist significantly fewer programs which represent output graphics using lines, these programs are usually very expensive.

This work proposes a vectorization algorithm and implements freely distributable program which represents vector image using lines. After initial pre-processing of input raster image its morphological skeleton is found (still in raster). A skeleton is then traced and converted into a vector form set of Bézier curves. An image is smoothed and exported to a required vector graphics format. Quality of an outcome is in many aspects comparable with quality of vectorization tools available to the author.

Keywords: vectorization, line-based graphics, image analysis, morphological operations, Bézier curves

Obsah

Úvod

V této práci se budeme zabývat vektorizací čárových kreseb, tedy převodem černobílých kreslených rastrových obrázků (skládajících se převážně z čar) do vektorového formátu. Příklad takového převodu vidíme na obrázku 1.

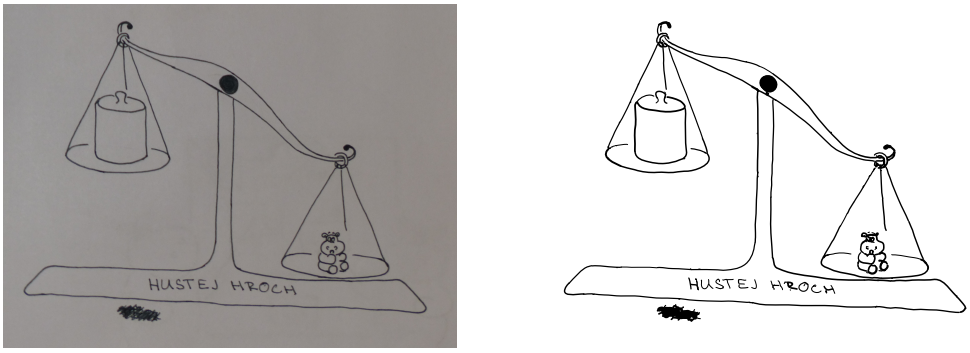
Opakem vektorizace je *rastrování*, které využíváme téměř vždy, když chceme vektorová data zobrazit. Toto ovšem činíme se znalostí rozlišovacích schopností výstupního zařízení (tiskárny, monitoru). U rastrových obrázků si pamatujeme informace o barvě každého pixelu – políčku pravidelné, nejčastěji čtvercové, mřížky. Jsou dobře známé, jelikož se s nimi běžně potkáváme třeba u fotografií. Mezi jejich formáty patří JPEG (Pennebaker a Mitchell, 1993), PNG (W3C, 2013), GIF (CompuServe Inc., 1990).

Oproti tomu vektorové obrázky tvoříme z *primitiv* – základních geometrických útvarů, jako jsou například úsečky, mnohoúhelníky, (Bézierovy) křivky a kružnice. Díky tomu je lze libovolně škálovat, aniž bychom naráželi na limit jejich kvality. Jsou proto vhodné i pro přípravu grafiky, kdy předem neznáme přesné cílové rozlišení. Podrobněji se na ně podíváme v první kapitole.

Vektorové obrázky jsou čím dál populárnější na webových stránkách, kde je lze dokonce s pomocí kaskádových stylů (CSS) animovat. Oproti rastrovým je také snazší je editovat, primitiva lze totiž přebarvovat, deformovat, otáčet, škálovat atp. V neposlední řadě může vektorová podoba sloužit i ke kompresi dat – pokud je obrázek snadno popsatelný primitivy, může být při stejné kvalitě výrazně menší než rastrový. Formátů existuje hned několik, například PostScript (PS)(Adobe Systems Inc., 1999) či Scalable Vector Graphics (SVG)(Dahlström a kol., 2011). Více si o nich povíme v podkapitole 1.3.

Při kreslení obrázků se někomu lépe kreslí tužkou na papír, než přímo do počítače (grafický tablet není tak rozšířený jako papír). Ovšem úprava obrázku už není na papíře tak jednoduchá. Využití vektorizece je tedy nasnadě. Máme fotografii či sken původního obrázku a chceme jej dále upravovat na počítači.

Lze potkat tři způsoby vektorizace – ruční, poloautomatickou a automatickou. Pro každý ze způsobů najdeme celou řadu nástrojů a služeb, které je nabízejí, od ruční vektorizace on-line službou VectorizeNow, kdy obrázek zcela překreslí zkušený grafik, po plně automatickou, zhotovenou kupříkladu nástrojem Vector Magic (Cedar Lake Ventures Inc.). Drobný přehled existujících nástrojů se nachází ve druhé kapitole.



Obrázek 1: Příklad převodu navrženým algoritmem

Hlavní motivací této práce však byla absence kvalitních volně dostupných vektorizačních nástrojů, které by se zaměřovaly na čárovou grafiku. Běžně se lze setkat s vektorizačními programy jako Potrace (Selinger, 2015) či AutoTrace (Weber, 2004), které nejsou takto specializované a převádí libovolné obrázky. V jejich výstupu je pak každá čára reprezentovaná jako vyplněný mnohoúhelník (přesněji jako oblast ohraničená různými křivkami, nikoli pouze úsečkami). I u čárové grafiky tak používají k vektorizaci metodu *trasování ploch*, která nevyužije vlastností těchto obrázků.

V této práci cílíme na návrh a implementaci (polo)automatické vektorizace na principu *trasování čar* tak, aby zde zůstala možnost výstup nadále upravovat příjemným způsobem ručně. Výstup je tedy reprezentován pomocí křivek s určitou šířkou stopy. Algoritmus je popsán v kapitole 3. Zdrojové kódy vzniklého programu Vectorix jsou volně dostupné na adrese: `https://atrey.karlin.mff.cuni.cz/~had/vectorix/`.

Ve čtvrté kapitole porovnáme výsledky vektorizování našeho programu s několika dalšími nástroji. Kvůli nedostupnosti jiných čárových vektorizátorů porovnáváme převážně s těmi, které trasují plochy. Z důvodu odlišné reprezentace dat se však kvalita výstupů srovnává obtížně.

Následující dvě kapitoly věnujeme uživatelské a vývojové dokumentaci. Poslední kapitola naváže s nápady na možná rozšíření programu.

1. Reprezentace vektorových dat

V závislosti na konkrétním formátu souboru a účelu vektorových dat se používají různá primitiva. Vektorová data totiž nemusí představovat pouze obrázky, ale také technické výkresy, či přímo instrukce pro numericky řízené obráběcí stroje (CNC, z anglického Computer Numeric Control).

Ty často využívají pouze úsečky, takže složitější útvary (například oblouk) se pro ně musí aproximovat lomenou čarou. To je však vhodné dělat se znalostí rozlišovacích schopností stroje, protože se tím zhoršuje podobnost s originálem.

Kromě úseček se také používají eliptické oblouky. Lze je například definovat počátečním a koncovým bodem, délkou poloos a jejich otočením a dvěma jednobitovými identifikátory: jeden určí, zda se jedná o větší nebo menší oblouk, a druhý pravotočivost/levotočivost oblouku. Přesně takováto reprezentace se používá ve formátu SVG (viz Dahlström a kol., 2011, sekce 8.3.8), který je také výchozím výstupním formátem programu Vectorix. (Ten však oblouky nepoužívá.)

1.1 Bézierovy křivky

Složitější útvary lze reprezentovat pomocí navazujících Bézierových křivek, tj. splinů na nich založených. Spliny Bézierových křivek jsou pro nás důležité, protože právě jimi je reprezentovaný výstup programu Vectorix. Píší o nich ve své knize pánové Piegl a Tiller (1997, kapitola 1). Ta je hlavní zdrojem informací pro celou tuto kapitolu.

Definice 1. *Bézierova křivka stupně n je parametrická křivka $C(t)$, s parametrem $t \in [0, 1]$, taková že:*

$$C(t) = \sum_{i=0}^{n} B_{i,n}(t) \cdot P_i,$$

kde P_i jsou kontrolní body a $B_{i,n}$ je i-tý Bernsteinův polynom stupně n. Všech $n+1$ kontrolních bodů dohromady tvoří kontrolní polygon. Parametr t nazýváme čas.

Definice 2. *i-tý Bernsteinův polynom $B_{i,n}$ stupně n je definován rekurentně:*

$$B_{i,0}(t) = \begin{cases} 1, & \text{pro } i = 0, \\ 0, & \text{pro } i \neq 0, \end{cases}$$

$$B_{i,n}(t) = (1 - t) \cdot B_{i,n-1}(t) + t \cdot B_{i-1,n-1}(t).$$

Speciálně tedy vychází, že Bézierova křivka prvního stupně odpovídá parametrické reprezentaci úsečky.

Pro vyšší stupně je pak možné hledat bod pro daný parametr t snadno pomocí rekurzivního algoritmu de Casteljau. Ten v každém kroku zredukuje stupeň Bézierovy křivky o jedna. Pro odvození redukce nejprve rozepíšeme hodnotu Bézierovy křivky pro parametr t podle definic 1 a 2:

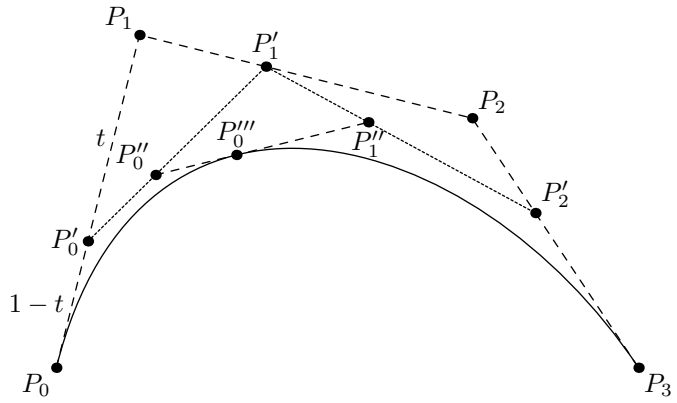
$$C(t) = \sum_{i=0}^{n} B_{i,n}(t) P_i = \sum_{i=0}^{n} (1 - t) \cdot B_{i,n-1}(t) \cdot P_i + \sum_{i=0}^{n} t \cdot B_{i-1,n-1}(t) \cdot P_i.$$

Nyní si stačí uvědomit, že Bernsteinův polynom $B_{n,n-1}$ a $B_{-1,n}$ je vždy nulový. Proto můžeme pokračovat:

$$C(t) = \sum_{i=0}^{n-1}(1-t)B_{i,n-1}(t)P_i + \sum_{i=0}^{n-1} tB_{i,n-1}(t)P_{i+1} = \sum_{i=0}^{n-1}((1-t)P_i + tP_{i+1})B_{i,n-1}(t).$$

Tím jsme zredukovali problém na vyhodnocení Bézierovy křivky stupně $n-1$ s n kontrolními body $P'_i = (1-t)\cdot P_i + t\cdot P_{i+1}$. Algoritmus tak používá jednoduché geometrické úkony, v každém kroku rozdělí úsečku mezi sousedními kontrolními body v poměru $t : 1-t$. Toto dělení je dobře vidět na obrázku 1.1.

V našem případě se setkáme nejvýše s kubickými Bézierovými křivkami. To je totiž nejvyšší stupeň, který je podporován v běžných formátch SVG a PS. Kubické křivky mají 4 kontrolní body.



Obrázek 1.1: Postupná redukce Bézierovy křivky třetího stupně

1.1.1 Vlastnosti

- Protože pro Bernsteinovy polynomy platí rovnost

$$\sum_{i=0}^{n} B_{i,n}(t) = 1, \quad (\forall t, 0 \le t \le 1)$$

  libovolný bod Bézierovy křivky leží uvnitř konvexního obalu kontrolního polygonu.

- Krajní kontrolní body P_0 a P_n jsou také krajními body Bézierovy křivky.

- Libovolnou afinní trasnformaci křivky lze provést transformací kontrolních bodů.

- Velmi důležitou, avšak nepříjemnou vlastností je, že *offsetovou křivku* nelze přesně reprezentovat Bézierovou křivkou. Offsetová křivka je taková křivka, která má od předlohy v každém bodě konstantní vzdálenost d. Důsledkem je, že pokud máme grafický objekt reprezentovaný Bézierovou křivkou s danou šířkou, nelze obvod tohoto objektu popsat Bézierovými křivkami přesně (viz Hoschek, 1988). S tímto úkolem se potýká i Vectorix, takže možné způsoby řešení jsou popsané v části 3.5.3.

- Délku Bézierovy křivky můžeme zdola odhadnout vzdáleností prvního a posledního kontrolního bodu a zhora délkou lomené čáry kontrolního polygonu.

- Bézierovu křivku stupně n je možné v libovolném bodě rozdělit a obě vzniklé části přesně reprezentovat Bézierovými křivkami stejného stupně, jako měla původní křivka. Použijeme k tomu mezivýsledky algoritmu de Casteljau. Kontrolní body původní křivky označíme $P_{0,i} = P_i$ a mezivýsledky z k-tého kroku $P_{k,i}$. První Bézierova křivka pak bude mít kontrolní body $P_{0,0}, P_{1,0}, P_{2,0}, \dots P_{n,0}$ a druhá $P_{n,0}, P_{n-1,1}, P_{n-2,2}, \dots P_{0,n}$.

  Tato vlastnost nám umožňuje křivky snadno vykreslovat. Každý segment budeme půlit do té doby, než jeho délka klesne pod námi definovanou mez. Následně každou křivku nahradíme úsečkou mezi prvním a posledním kontrolním bodem. Tím umíme získat libovolně kvalitní aproximaci lomenou čarou.

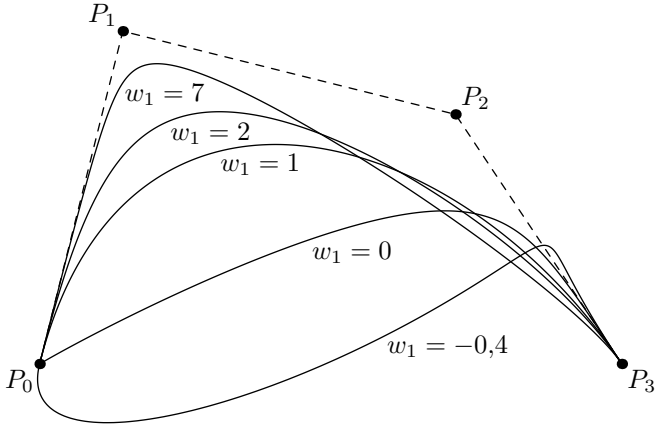
1.1.2 Racionální Bézierovy křivky

Někdy se Bézierovy křivky používají ve variantě rozšířené o váhové koeficienty. Každý kontrolní bod P_i, resp. jemu odpovídající Bernsteinův polynom $B_{i,n}$, je přenásobený váhovým koeficientem w_i. Aby se stále jednalo o afinní kombinaci, jsou výsledné koeficienty normovány:

$$C(t) = \frac{\sum_{i=0}^{n} B_{i,n}(t) \cdot w_i \cdot P_i}{\sum_{j=0}^{n} B_{j,n}(t) \cdot w_j}.$$

Intuitivně platí, že větší váha přidává bodu na „důležitosti“ a křivka se tak k danému bodu více přimyká. Dokud jsou všechny váhy kladné, leží celá křivka v konvexním obalu kontrolních bodů. Se zápornými váhami to však již neplatí. Oba tyto jevy jsou dobře pozorovatelné na obrázku 1.2.

Pokud jsou všechny váhy w_i jednotkové, dostáváme předpis z definice 1, tedy obyčejné Bézierovy křivky (občas se jim kvůli absenci vah říká *neracionální* Bézierovy křivky).



Obrázek 1.2: Racionální Bézierovy křivky se shodnými kontrolními body a vahami $w_0 = w_2 = w_3 = 1$ a proměnlivou váhou $w_1 \in \{-0{,}4; 0; 1; 2; 7\}$

.

Racionální Bézierovy křivky dokáží přesně reprezentovat libovolné kuželosečky. To neracionálními není možné. Pokud má neracionální Bézierova křivka kreslit část kružnice/elipsy, vždy se jedná pouze o aproximaci.

1.2 Cesta

Jako *cestu* označujeme posloupnost na sebe postupně navazujících primitiv, jako jsou úsečky, eliptické oblouky a Bézierovy křivky. U jednotlivych navázání definujeme třídu geometrické spojitosti řádu k, značíme ji G^k, viz Hoschek (1988).

Parametrické křivky $A(t), 0 \leq t \leq 1$ a $B(s), 0 \leq s \leq 1$ splňují podmínky spojitosti:

- třídy G^0, pokud $A(1) = B(0)$, tj. na sebe „navazují“,
- třídy G^1, pokud navíc $A'(1) = l_1 \cdot B'(0)$, tj. křivky mají stejnou směrnici, výsledný spline je hladký,
- třídy G^2, pokud navíc $A''(1) = l_1^2 \cdot B''(0) + l_2 \cdot B'(0)$, tj. křivky mají stejný poloměr křivosti,

kde l_i jsou libovolné parametry. V tomto případě čárkou značíme derivaci. Tyto parametry kompenzují skutečnost, že parametrizace křivek mohou být libovolné. Při stejné změně parametrů t a s mohou křivky A a B urazit zcela odlišnou vzdálenost.

U cesty vyžadujeme spojitost G^0, spojitosti vyšších řádů jsou již volitelné. Často se ještě setkáme se spojitostí G^1, protože ta zaručuje, že na křivce nejsou žádné ostré zlomy (rohy). Je samozřejmě možné definovat třídy i pro vyšší řády, ale nejsou pro nás příliš zajímavé, protože nejsou pouhým okem rozpoznatelné.

1.3 Používané formáty

1.3.1 Scalable Vector Graphics (SVG)

Obdobně jako je tomu u rastrových obrázků, existuje řada různých formátů i pro vektorové. Pro náš program je nejdůležitější formát Scalable Vector Graphics (SVG) (Dahlström a kol., 2011). Je totiž jeho výchozím výstupním formátem. SVG běžně používá například open-source editor vektorových obrázků Inkscape a je také podporován všemi (s výjimkou jediného) vektorizačními nástroji ukázanými v následující kapitole. Jedná se o formát založený na XML (Extensible Markup Language).

Formát umožňuje kromě všech základních primitiv popsaných výše (s výjimkou racionálních Bézierových křivek) také jednotlivé objekty slučovat do skupin, transformovat je, nastavovat jim průhlednost či na ně kreslit barevné přechody. Cesty pak lze například vykreslovat přerušovaně. Do obrázku je také možné vložit rastrový obrázek, či text, který je následně stále reprezentován jako text. Nemá smysl zde vyjmenovávat všechny možnosti formátu. My využíváme pouze cesty složené z Bézierových křivek. Pro ladící účely program také umožňuje vložit na pozadí vektorového obrázku libovolný jiný (třeba rastrový) obrázek. Můžeme si tak snadno zobrazit v jednom souboru jak předlohu, tak výsledek vektorizace.

Za povšimnutí ještě stojí, že SVG je definováno konsorciem W3C (World Wide Web Consortium), které vyvíjí standardy pro web. Formát SVG je totiž běžně podoporván ve všech (grafických) webových prohlížečích. Obrázky v SVG je dokonce možné na webu animovat pomocí kaskádových stylů, či je přímo vkládat do kódu stránky. Lze tedy očekávat, že tento formát v blízké budoucnosti jen tak nezmizí.

Pro nás je jedinou nevýhodou SVG (ale i ostatních formátů), že nepodporuje proměnlivou šířku čáry. Při vektorizaci však občas nacházíme čáry, jejichž tloušťka stopy se v průběhu mění. Před ukládáním se musíme s tímto nedostatkem vypořádat. Detaily jsou popsané v kapitole 3.5.3. Je možné, že v příští verzi formátu bude proměnlivá šířka již podporována – viz návrh (Birtles, 2014).

1.3.2 PostScript, PS

Druhý podporovaný formát je PostScript (PS) (Adobe Systems Inc., 1999), který je výrazně starší. Ve skutečnosti se jedná o zásobníkový, turingovsky úplný programovací jazyk. PostScript pomocí příkazů jako `moveto` (přemísti se na pozici) a `curveto` (nakresli kubickou Bézierovu křivku vedoucí ze současné pozice na novou pozici s použitím dvou dalších kontrolních bodů) postupně vykresluje výsledný vektorový obrázek.

Běžně se v PostScriptu popisuje vzhled jedné stránky. PS se totiž používá při tisku, kdy tiskárny (případně tiskové servery) v sobě obsahují jeho interpret.

Ve skutečnosti je výstupem programu soubor typu EPS (Encapsulated PostScript). Ten se od obyčejného PS liší jen v drobnostech. Předně definuje obdélníkovou oblast, ve které se kreslený obrázek nachází (u PostcSriptu tato oblast odpovídá jedné stránce). Na EPS jsou kladena další omezení – například musí zachovat aktuální pozici. Protože náš export využívá jen zlomek z funkcionality, na opravdové rozdíly nenarazíme.

1.3.3 Další formáty

Existuje mnoho dalších vektorových formátů. Některé z nich jsou dobře dokumentované, jiné jsou (neveřejně) definované výrobcem softwaru, který tento formát používá. Mezi ty dokumentované spadá formát DXF (Drawing Exchange Format) určený pro software CAD (Computer aided design, tj. programů na rýsování technických výkresů).

V tomto formátu (ale i v jiných) se hladké křivky reprezentují pomocí B-splinů (či dalších variant jako třeba NURBS – Non-uniform rational Basis spline). Náš nástroj však tuto reprezentaci nepoužívá, nebudeme ji proto ani definovat. Případné zájemce odkážeme na podrobnou knihu *The NURBS Book* (Piegl a Tiller, 1997).

2. Metody vektorizace a existující nástroje

Vektorizace označuje celý proces převodu z rastrového obrázku na vektorový. Skládá se zpravidla z předzpracování dat ještě v rastrové podobě, trasování linií (sledování útvarů v rastru a jejich převod na vektorovou podobu) a volitelně z dalších úprav již vektorové podoby.

Jednotlivé přístupy k trasování se liší podle toho, jak a k čemu chceme zvektorizovaná data používat. Vektorizace najde uplatnění například při automatické digitalizaci map (Lána, 2001), kdy se snažíme správně rozpoznat mapové značky. V takovém případě je vektorizátor závislý na konkrétní úloze a pro dobré výsledky by měl znát sémantiku dat.

Dále se vektorizace využívá v lékařství, např. při zpracování trojrozměrných skenů. Tím, že se automaticky podaří najít středy cév či střev, se následně usnadní práce lékařům. Blíže se tímto problémem zabývá článek Bittera a kol. (Bitter a kol., 2000).

Hledání středů čar se také využívá při práci s technickými výkresy, protože software pro CAD často umí pracovat pouze s vektorovými podklady. Jelikož běžné výkresy jsou složené z úseček a kružnic, některé vektorizační nástroje, například RasterVect, hledají pouze tato primitiva.

Jak jsme již nastínili v úvodu, je možné obrázek reprezentovat i pomocí ploch. S tím se při vektorizaci setkáváme častěji, protože je jednak snazší a mnohdy rychlejší najít hranice objektů než jejich středy, a jednak při trasování středů čar potřebujeme vstupní obrázek složený právě z čar. Trasování ploch neklade tak přísná omezení na vstup, protože pomocí ploch lze dobře reprezentovat libovolný obrázek.

2.1 Trasování ploch

Obecně se snažíme sledovat a průběžně popisovat hranice souvislé (jednobarevné) plochy. Tuto metodu využívá například open-source nástroj Potrace (Selinger, 2015) a v první fázi také komerční Vector Magic (Cedar Lake Ventures Inc.; Diebel, 2008). Detaily metody si ukážeme na příkladu programu a knihovny Potrace, protože k němu existuje velmi dobrá dokumentace (Selinger, 2003) a jsou také veřejně dostupné zdrojové kódy.

2.1.1 Potrace

Potrace je primárně knihovna, nicméně je k ní dodáván i stejnojmenný konzolový program. Jako knihovnu jej využívá třeba open-source vektorový editor Inkscape.

Vstupní obrázek je nejprve převeden na binární, v případě knihovny tento převod musí provést volající program. Z toho plyne, že Potrace sám od sebe nepodporuje vícebarevné obrázky.

Funguje v několika krocích. Prvně obrázek rozloží na uzavřené oblasti. Sleduje hranici mezi černou a bílou a jakmile se dostane zpět do počátečního bodu, nalezl

novou uzavřenou plochu. Hodnoty všech pixelů uvnitř této oblasti invertuje, díky čemuž postupně najde kontury všech objektů v obrázku.

Následně nalezenou plochu aproximuje mnohoúhelníkem a v další fázi jej vyhladí pomocí hladkých Bézierových křivek. Zde se program snaží detekovat ostré rohy, aby zůstaly zachovány. Křivky se pak volitelně ještě snaží zjednodušit tím, že sousední úseky propojuje.

Vlastnosti

Například v programu Inkscape jsou po vektorizování knihovnou Potrace bílé plochy chápany jako samostatné bílé objekty a nejsou odečteny od černé, na které leží. Tím se zhoršuje editovatelnost obrázků.

Další nevýhodou je, že samotné trasování hranic ploch je čistě lokální záležitost, tj. pracuje jen v malém okolí. Pokud je obrázek složený z relativně tenkých čar, poznáme, že každý z okrajů čáry je trasován zcela nezávisle. Vytvoří se totiž první vektorizační artefakty: úseky čáry s proměnlivou šířkou. Při trasování vznikají drobné odchylky od ideálního směru. Tím, že se objevují nezávisle, se šířka čáry ve výstupu mění, aniž by takováto změna byla na vstupu.

Další artefakty se objeví například při křížení dvou čar. Při trasování rastrového obrázku po obvodu se nám nemusí podařit najít jeden konkrétní bod, ve kterém se má okraj ostře zalomit. Místo toho může dojít ke slití (a tedy opět lokálně k rozšíření čar na obrázku). Toto se Potrace snaží eliminovat detekcí rohů, ale není to zcela spolehlivé.

2.1.2 Vector Magic

Komerční vektorizační program Vector Magic navazuje na na Diebelovu disertační práci (Diebel, 2008) zabývající se pravděpodobnostní vektorizací. V ní popisuje algoritmus, který je téměř jistě v programu Vector Magic stále používaný.

K vektorizaci přistupuje jakožto k inverznímu problému rasterizace pomocí bayesovské statistiky. Hledá nejpravděpodobnější podobu vektorové předlohy, která se vykreslením do rastru převede na vstupní obrázek. Při tomto vykreslování předpokládá zapnutý anti-aliasing. Problém vektorizace tedy převádí na optimalizační problém.

Celou úlohu řeší tak, že nejprve nalezne přibližné pozice všech útvarů a následně je upřesní metodou konjugovaných gradientů. Teprve poté jsou hranice jednotlivých objektů převedeny na kubické Bézierovy křivky.

Vlastnosti

Vector Magic podporuje plně automatickou vektorizaci, kdy sám určí všechny parametry a obrázek převede. V této konfiguraci pracuje s libovolným počtem barev. Uživatel tedy přímo nemůže ovlivnit ani to, kolik barev se objeví na výstupu.

V pokročilém režimu si uživatel může zvolit používanou paletu barev. Dalšími parametry lze ladit složitost objektů, filtrování drobných chyb a míru odstraňování artefaktů vzniklých slévání čar. Dále je možné obrázek ručně upravit ještě

v bitmapové podobě a opravit tak chyby při segmentaci – přidělení nejpravděpodobnější barvy danému pixelu. V poslední části – zřejmě odpovídá převodu na Bézierovy křivky z Diebelovy práce (Diebel, 2008) – je možné nastavit hladkost a jednoduchost čar a případně zapnout detekci rohů.

Pro snazší nastavování parametrů má Vector Magic ještě zjednodušený režim, ve kterém uživatel vybírá typ vstupu (fotografie, obrázek s ostrými hranami a antialiasovaný), jeho přibližnou kvalitu a množství barev. Barevnou paletu je možné libovolně upravit. Tím neupravujeme pouze mapování barev ve výstupu, ale rovnou vybereme, které barvy se má Vector Magic snažit hledat. Na konci má uživatel možnost obrázek drobně ručně upravit (například smazat jednotlivé objekty a odstranit tak pozadí).

2.2 Čárová vektorizace

Alternativou k trasování ploch je vektorizace obrázků sledováním středů jednotlivých objektů (centerline-tracing). Algoritmy používající tuto metodu by v ideálním případě měly být schopné se výše popsaným vektorizačním artefaktům vyhnout.

Při vektorizaci totiž nepracují jen s jednou hranicí, ale předpokládají, že celý obrázek je složen z čar. Ty se následně snaží vyhledat. Protože obrázek reprezentují pomocí čar a nikoli ploch, mohou snadno zabránit nežádoucím výkyvům v jejich šířce. Teoreticky nemusí ani docházet k problémům se sléváním čar svírajících ostré úhly, protože dvě křížící se čáry jsou vektorově reprezentované jako dvě samostatné s pevnou šířkou.

Libovolnou čáru pak ve vektorové podobě můžeme snadněji upravovat. Při jejím otáčení, ohýbání a prodlužování se rovnou zachovává její šířka.

Čárová vektorizace však nemusí dobře fungovat, pokud vstup není složený z čar. Můžeme si položit otázku, jak bychom správně měli pomocí čar reprezentovat jeden plný čtverec?

Jelikož problém čárové vektorizace není snadný a výhody se projeví jen na určitém typu vstupu, přirozeně neexistuje ani tolik nástrojů.

2.2.1 RasterVect

RasterVect je komerční nástroj, který nabízí čtyři typy reprezentace – obdélníky (Solids), obrysy (Outlines), vyplněné obrysy (Filled outlines), čáry (Centerlines). První je sice dostupná i v bezplatné verzi, ale neudělá nic víc, než převedení každého pixelu na čáru o šířce jednoho pixelu. To sice působí jako zbytečná funkce, ale může to být praktické pro uživatele CAD softwaru, který nepodporuje bitmapové formáty. Třetí reprezentace jsou již zmíněné plochy. Druhá se oproti tomu liší jen v tom, že u nalezeného obvodu plochy není ve výstupu nastavena barva výplně. Pro nás nejzajímavější je však poslední z variant, protože se našeho tématu týká nejúžeji.

Program RasterVect poskytuje také editační nástroje jak pro rastrové obrázky, tak po natrasování pro vektorové. Protože program je cílený právě na technické výkresy, používá při metodě centerline pouze úsečky a oblouky.

Výstupy porovnáme s našimi v kapitole 4.

3. Návrh algoritmu

Náš vektorizační algoritmus pracuje v několika krocích. Nejprve dojde k předzpracování v rastrové podobě. Obrázek převedeme na binární – černobílý. Aby názvy operací ve druhé fázi odpovídaly svému intuitivnímu významu, budeme pracovat s černým pozadím (hodnota 0) a objekty v popředí budou bílé (hodnota 1). Protože lze předpokládat častěji vstupní obrázky s bílým podkladem, barvy jsou v implementovaném programu na začátku invertovány. Toto nastavení lze snadno zrušit.

Druhým krokem je hledání morfologické kostry (v tuto chvíli již vždy bílých) objektů. Kostra odpovídá v rastrové podobě středům hledaných čar. V této fázi (během hledání) také předpočítáme šířky čar. To vše stále v rastrové podobě.

Teprve poté dojde k samotné vektorizaci. Metoda se snaží trasovat (stopovat) jednotlivé čáry. Vyjdeme z jednoho bodu kostry a snažíme se pokračovat po čáře tím, že sledujeme její kostru. Použité pixely z kostry si průběžně značíme, abychom zabránili opakovanému průchodu stejnými místy. Pokud nemáme kam pokračovat, nalezli jsme konec jedné čáry. Tento postup opakujeme, dokud v kostře zbývají nějaké body. Tím dostaneme první hrubou vektorovou podobu linií.

Nyní přijde na řadu čtvrtý krok, ve kterém se pracuje již s vektorovou reprezentací. Navazující úseky můžeme spojit, pokud jsou dostatečně monotónní. Tím snížíme velikost výstupu, aniž bychom se odklonili od podoby originálu. (Vypustíme ty kontrolní body, které nepřinášejí novou informaci.) Původní křivku tedy aproximujeme s pomocí jiné s nižším počtem úseků, tedy i kontrolních bodů.

Poslední fází je samotný export dat do zvoleného formátu. Ani jeden z používaných (SVG a PS) totiž neumí přímo reprezentovat cesty s proměnlivou šířkou. Vectorix však v průběhu trasování počítá s tím, že se šířka může měnit. Algoritmus se snaží sám rozhodnout, jaká reprezentace je vhodná pro danou čáru a šířky buď zprůměruje, nebo křivku převede na její obvod.

3.1 Fáze 1: Prahování a filtrování

Pro hledání morfologické kostry potřebujeme, podobně jako knihovna Potrace, čistě bitonální (dvoubarevný) obrázek. K jeho získání se zpravidla používá prahování (thresholding) obrázku, který je v odstínech šedi. Pixely s jasem nad danou hodnotu prahu označíme za bílé a ostatní za černé. Potřebujeme k tomu jen znát vhodné nastavení prahu. Ten samozřejmě může být pro různé obrázky různě velký.

3.1.1 Otsova metoda

Velikost optimálního prahu lze pro daný obrázek vypočítat například pomocí Otsovy metody (Otsu, 1979). Ta přepokládá, že hodnota každého pixelu má odpovídat jedné ze dvou barev a jenom je navíc zatížena šumem. Hledáme práh – hodnotu jasu, která pixely rozděluje na dvě třídy.

Otsova metoda nejprve pro obrázek určí jeho histogram[1]. Následně vyzkouší

[1]Histogram obrázku udává četnosti pixelů s jednotlivými úrovněmi jasu.

všechny možné hodnoty prahu tj. rozdělení histogramu na dvě třídy. Třídy označíme čísly 0 (pixely s jasem menším nebo rovným prahu) a 1 (jas větší než práh). Protože jsme u obrázků limitování zpravidla 256 možnými jasy a celý výpočet můžeme provádět na histogramu, není toto zkoušení příliš náročné.

Otsu ve zmíněné práci ukazuje, že optimální hodnota prahu je ta, pro kterou vychází nejmenší vážený součet rozptylů v rámci každé z tříd (každý rozptyl je přenásobený vahou odpovídající velikosti dané třídy):

$$\sigma_w^2(t) = \omega_0(t) \cdot \sigma_0^2(t) + \omega_1(t) \cdot \sigma_1^2(t),$$

kde $\omega_0(t)$ a $\omega_1(t)$ jsou postupně četnosti pixelů z tříd 0 a 1; $\sigma_0^2(t)$ a $\sigma_1^2(t)$ jsou rozptyly jednotlivých tříd.

Ty se na obrázku s jasy z množiny $\{0, 1, \ldots L\}$ definují následovně:

$$\sigma_0^2(t) = \sum_{i=0}^{t} \big(i - \mu_0(t)\big)^2 \frac{p(i)}{\omega_0(t)}, \qquad \sigma_1^2(t) = \sum_{i=t+1}^{L} \big(i - \mu_1(t)\big)^2 \frac{p(i)}{\omega_1(t)},$$

kde $\mu_0(t)$ a $\mu_1(t)$ značí střední hodnotu jasu piexlů v dané třídě a $p(i)$ je četnost pixelů s jasem i. (Je-li některá z četností $\omega_k(t)$ nulová, zadefinujeme odpovídající rozptyl jako $\sigma_k^2(t) = 0$.) Příklad histogramu a jemu odpovídajících rozptylů vidíme na obrázku 3.1.

3.1.2 Adaptivní prahování

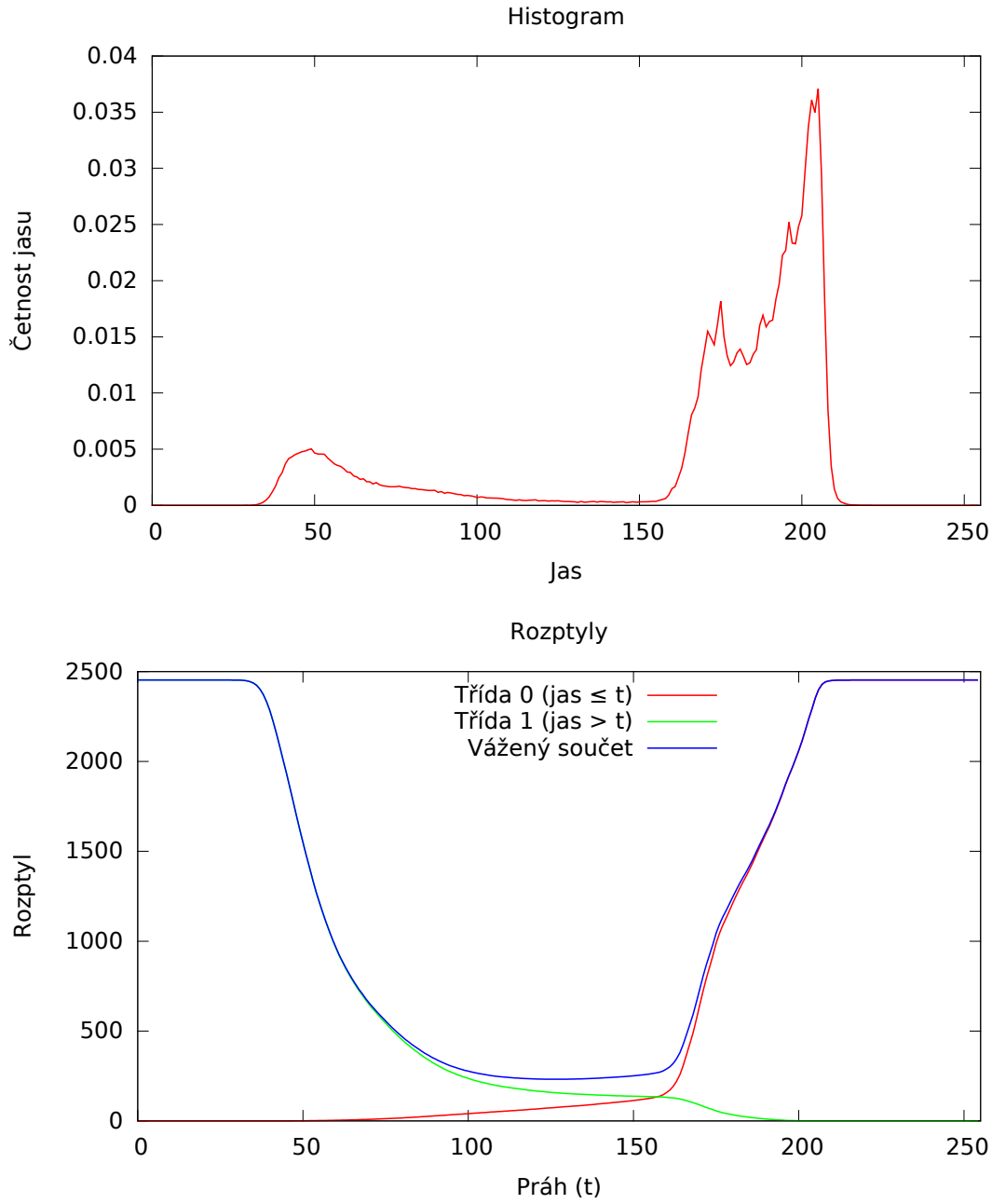
Ani Otsova metoda si však neporadí s obrázky, které jsou v různých částech osvětlené odlišně, a s jedním nastavením prahu nedostaneme nikdy dobrý výsledek. Příkladem může být nekvalitně vyfotografovaný a proměnlivě osvětlený papír. Pro takové vstupní obrázky potřebujeme zařídit, aby hodnota prahu závisela na průměrném jasu v dané oblasti. Toho lze docílit s pomocí *adaptivního prahování*.

Funguje tak, že v každém pixelu vybereme jako práh průměrnou hodnotu z okolních pixelů posunutou o určitou konstantu. Velikost okolí přitom potřebujeme dobře zvolit. Pokud je příliš malé (celé se vejde do objektu na obrázku), může se nám stát, že některé pixely z vnitřku bílého objektu označíme jako černé (případně naopak černé označíme jako objekt). Proti tomu velké okolí způsobí, že pro určení prahu se průměruje velká část obrázku a tedy se práh nezmění při vyhodnocování pixelů v tmavších a světlejších částech.

V programu si požadovaný typ prahování vybírá uživatel. Pokud neurčí jinak, použije se fixní práh spočtený Otsovou metodou. Pro méně kvalitní vstupy ale může být potřeba zvolit právě adaptivní prahování.

3.1.3 Filtrování nedokonalostí

Protože vstupní obrázek může být zašumělý, je možné, že bílé objekty v sobě obsahují malé černé tečky. V další fázi však potřebujeme, aby objekty žádné nedokonalosti neobsahovaly. Zacelíme proto všechny díry, které jsou menší než dvojnásobek zadané konstanty, která určuje jak vzdálené pixely považujeme za sousední. Toho docílíme morfologickou operací *uzavření*. Abychom se také zbavili šumu opačného typu (bílé tečky), provedeme následně ještě operaci *otevření*. Podrobněji si obě popíšeme hned v následující fázi.



Obrázek 3.1: Histogram a jemu odpovídající rozptyly tříd používané Otsovou metodou pro výpočet prahu. Jeho optimální hodnota je v tomto případě $t = 126$.

3.2 Fáze 2: Morfologická kostra

U černobílých obrázků můžeme hovořit o takzvané *morfologické kostře*. Jedná se o množinu středů vepsaných kružnic, případně jiných geometrických objektů. Protože v této fázi ještě nevíme, kterým směrem křivka vede, je pro nás vhodné vepisovat kružnice, jelikož jsou invariantní vzhledem k rotaci. Z toho důvodu také ve výstupu vykreslujeme čáry s kulatými konci. Nemusíme tak konce nijak speciálně řešit, protože jsou vždy správně orientované.

Vepisování kružnic by tedy bylo pro další zpracování ideální. Díky čtvrté fázi (post-processing) však nepotřebujeme najít kostru zcela přesně. Vystačíme si většinou s vepisováním osmiúhelníku, čtverce, či diamantu – čtverce otočeného o úhel $\pi/4$, tedy postaveného na špičku. Toto zjednodušení nám přinese hlavně zrychlení celého výpočtu.

3.2.1 Morfologické operace

Nejprve si zavedeme používané obrazové operace na binárním obrázku. První z nich je *eroze (erosion)*. Při erozi změníme všechny bílé pixely, které mají alespoň jednoho černého souseda, na černé. Tím bílé objekty obereme o jejich krajní pixely.

Opačným způsobem se chová *dilatace (dilation)*: všechny sousední pixely bílých obarvíme také na bílo. Dilatace tedy změní ty stejné pixely, které by změnila eroze na invertovaném obrázku. Příklad obou operací vidíme na obrázku 3.2.

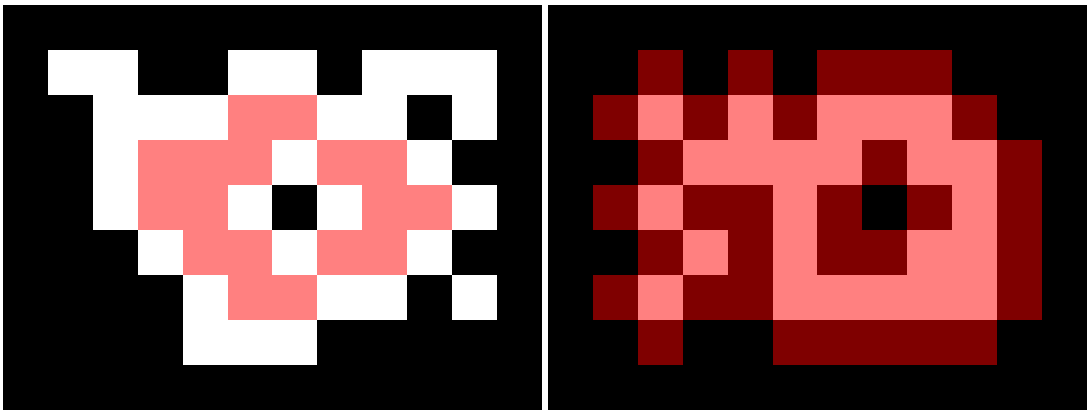
Běžně se za sousedy považují všechny pixely, které s daným sousedí hranou či rohem. Jedná se o takzvané osmiokolí – každý pixel s výjimkou krajních sousedí s osmi pixely. Druhou nejčastější variantou je čtyřokolí, kdy sousední pixely musí mít společnou hranu. Podle toho, jak přesně definujeme sousední pixely pro tyto dvě operace, dostaneme nakonec buď kostru vzniklou vepisováním čtverců (osmiokolí), nebo diamantů (čtyřokolí).

Dalšími operacemi jsou *otevření (open)* a *uzavření (close)*. Podíváme se, co se s obrázkem na vstupu stane při operaci otevření, což je ekvivalentní s provedením eroze a následně dilatace. Otevřením smažeme ty bílé pixely, které byly osamocené, tedy zmizely při erozi a dilatace je již neměla jak obnovit. Uzavření je naopak nejprve dilatace a poté eroze, tj. „zacelíme“ hranice objektů a odstraníme vnitřní díry. Pro filtrování nedokonalostí v předchozí fázi používáme pro tyto operace jako sousední pixely všechny, které jsou vzdálené maximálně zadanou konstantu.

3.2.2 Výpočet morfologické kostry – skeletonizace

Před vypočtením morfologické kostry (skeletonizací) již máme z předchozí fáze binární rastrový obraz, tedy vstup po prahování a filtrování. Výstup skeletonizace budeme uchovávat také v rastrovém obrázku, který bude na začátku prázdný (tj. černý).

V každém kroku skeletonizace provedeme otevření, které některé pixely smaže. Tyto pak tvoří kostru tenkých objektů, například v prvním kroku to budou jednopixelové čáry. Proto je přidáme do výstupního obrázku s kostrou. Nyní vstup zerodujeme, čímž všechny bílé objekty ztenčíme, a celý postup opakujeme.



(a) Eroze (b) Dilatace

Obrázek 3.2: Příklad eroze a dilatace na čtyřokolí, vstupní obrázek je zobrazen bíle, výstup operace poloprůhlednou červenou. Pro určení sousednosti pixelů zde používáme čtyřokolí

Pokud ve vstupním obrázku není žádný bílý pixel, jsme hotovi a na výstupu máme celou morfologickou kostru.

Algoritmus tedy vypadá následovně:

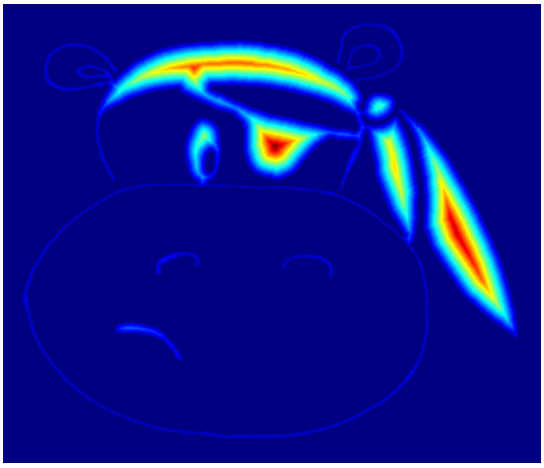
- $In \leftarrow$ Vstup `// 0 - černý pixel, 1 - bílý pixel (objekt)`
- Vynuluj Out
- Opakuj, dokud In obsahuje bílý pixel:
  - $T \leftarrow \text{dilate}(\text{erode}(In))$ `// operace open`
  - $D \leftarrow \text{and}(In, \text{not}(T))$ `// smazané pixely (`$\in In \ \wedge \notin T$`)`
  - $Out \leftarrow \text{or}(Out, D)$ `// přidání do kostry`
  - $In \leftarrow \text{erode}(In)$ `// zmenšení bílých objektů`
- Vrať Out

Zde také poznamenejme, že ve výstupním obrázku můžeme rovnou zaznamenávat informaci o tom, v kolikáté iteraci jsme daný bod do kostry přidali. Tím máme uloženou informaci o vzdálenosti daného bodu od nejbližšího okraje, a tedy i polovinu šířky čáry, která tímto bodem vede.

Aby byl algoritmus konečný, potřebujeme mít na vstupu alespoň jeden černý pixel. Ten snadno získáme tak, že kolem celého vstupu přidáme jednopixelový černý rámeček. Zároveň tím také zařídíme, že kostra čáry vedoucí podél okraje bude opravdu v jejím středu, protože tuto čáru budeme ztenčovat z obou stran stejně. Kdybychom rámeček nepřidali, tak pixely na hranici nebudou mít černé sousedy, a eroze s nimi nic neudělá.

Pokud jako sousední pixely označíme ty, které mají společnou jednu hranu (čtyřokolí), dostaneme kostru po vepisování diamatů. Alternativním pohledem tuto kostru tvoří body, které jsou od hranice nejvzdálenější (tj. středy vepsaných kružnic) v manhattanské metrice $\mathcal{L}^1$. Pokud jako sousedy označíme i pixely sousedící rohem (osmiokolí), budeme vkládat čtverce – kružnice v metrice $\mathcal{L}^\infty$.

Obrázek 3.3: Kostra po odprahovaní



Obrázek 3.4: Vzdálenostní mapa

Klasické eukleidovské metrice $\mathcal{L}^2$ se můžeme o trochu přiblížit tím, že budeme obě dvě varianty pravidelně střídat. Tím efektivně dosáhneme vkládání osmiúhelníků. Tento způsob výpočtu kostry byl inspirován algoritmem Diamond-Square (Fournier a kol., 1982) na generování náhodného terénu. Ukázka skeletonizace je na obrázku 3.3.

Složitost výpočtu

Označme rozměry obrázku, r výšku a s šířku. Po přidání rámečku proběhne maximálně $\min(r, s)/2$ iterací, protože nejpozději v i-té umažeme řádky i a $r - i$ a sloupce i a $s - i$. V každé iteraci provedeme nejvýše konstantní počet operací na jedno políčko. V případě zcela bílého vstupního obrázku (pouze s černými okraji) na tento limit narazíme. Celková časová složitost tedy je $\mathcal{O}(r \cdot s \cdot \min(r, s))$.

Teoreticky by bylo možné algoritmus zrychlit až na složitost $\Theta(r \cdot s)$, protože morfologické operace mění pouze ty pixely, které jsou na hranici černých a bílých objektů. Není proto potřeba v každém kroku procházet celý obrázek, ale stačí pracovat na bílých pixelech, které mají černého souseda. Takovéto pixely nikdy nebudeme zpracovávat ve více iteracích, protože jsou na konci aplikováním eroze smazány. Toto zrychlení můžeme implementovat pomocí fronty.

Vepisování euklidovských kružnic

Pokud bychom chtěli do obrázku vepisovat opravdové kružnice, algoritmus musíme drobně upravit. Erozi nebudeme aplikovat na zerodovaný obrázek z předchozí iterace, ale vždy vezmeme počáteční vstup a zerodujeme jej tak, že za sousední pixely považujeme všechny ve vzdálenosti rovnající se počtu již proběhlých iterací. V k-té iteraci (číslováno od 0) tedy nejprve „oloupeme“ k vrstev a poté do kostry přidáme ty pixely, které jsou osamělé – zmizí operací open na čtyřokolí.

Problém je, že tato varianta algoritmu je pomalá. Jen samotná eroze v k-té iteraci trvá $\Theta(r \cdot s \cdot k^2)$ operací, protože pro každý pixel musíme překontrolovat řádově k^2 sousedních pixelů. Všechny kroky dohromady mají proto časovou složitost $\mathcal{O}(r \cdot s \cdot \min(r, s)^3)$, protože omezení na počet iterací je opět stejné jako v předchozím případě, nejvýše $\min(r,s)$.

Vzdálenostní mapa

Podobně jako počítáme šířku čáry pro body kostry, tedy jako číslo iterace, ve které jsme bod přidali do kostry, můžeme vzdálenost od okraje počítat pro všechny body. Stačí si pro každý zapamatovat, v kolikáté iteraci jsme jej odebrali ze vstupu pomocí eroze. (V případě vepisování kružnic jde pouze o první odebrání.) Ukázka vzdálenostní mapy je na obrázku 3.4.

3.2.3 Zhangův-Suenův algoritmus

Výstup výše uvedeného algoritmu na hledání kostry nezaručuje, že je výsledná kostra souvislého úseku také souvislá. Souvislost kostry nám však pomůže v další fázi vektorizace. Ukážeme proto ještě Zhangův-Suenův algoritmus (Zhang a Suen, 1984), který souvislost zachovává.

Zhangův-Suenův algoritmus obdobně jako předchozí popsaný v jednotlivých krocích odebírá z okraje objektu pixely. Odmítne však odstranit ty pixely, které by kostru mohly rozdělit (artikulace). Jakmile jeden krok nezmění obrázek, zůstala algoritmu kostra obrázku.

Sousední pixely pixelu P_1 si pojmenujeme podle následující tabulky:

$$\begin{array}{ccc} P_9 & P_2 & P_3 \\ P_8 & P_1 & P_4 \\ P_7 & P_6 & P_5 \end{array}$$

Odstranění pixelů v každém kroku probíhá paralelně. Nejprve si tedy všechny pixely k odstranění označíme a teprve poté je smažeme. Jsou dva typy kroků, které se pravidelně střídají. Pixel P_1 označíme v lichém kroku ke smazání, pokud platí následující podmínky:

(a) P_1 je bílý, (černé pixely nemusíme mazat vícekrát)
(b) $2 \leq B(P_1)$, (pixel není součástí kostry)
(c) $B(P_1) \leq 6$, (pixel je na okraji objektu)
(d) $A(P_1) = 1$, (pixel není artikulace)
(e) alespoň jeden z pixelů $\{P_2, P_4, P_6\}$ je černý,
(f) alespoň jeden z pixelů $\{P_4, P_6, P_8\}$ je černý,

kde $B(P_1)$ je počet bílých pixelů v množině $\{P_2, P_3, \ldots P_9\}$ a $A(P_1)$ je počet vzorů `černá, bílá` v cyklické posloupnosti $P_2, P_3, \ldots P_9, P_2$. Sudý krok je stejný, jenom se změní podmínky (e) a (f) na následující:

(e') alespoň jeden z pixelů $\{P_2, P_4, P_8\}$ je černý,
(f') alespoň jeden z pixelů $\{P_2, P_6, P_8\}$ je černý.

U Zhangova-Suenova alogritmu si sice také můžeme pamatovat číslo kroku, ale neodpovídá zde jako u předchozího algoritmu poloměru nějakého vloženého objektu. Z toho důvodu jsou v programu implementovány oba algoritmy. Zhangův-Suenův se používá na nalezení kostry a pro výpočet vzdálenostní mapy se využívá vkládání osmiúhelníků střídáním diamantu a čtverce.

3.3 Fáze 3: Trasování

Třetím krokem je samotný převod z rastrové podoby morfologické kostry na vektorovou. Většina pixelů na kostře sice odpovídá středům hledaných čar, ale občas z kostry vybočují výběžky k hranicím bílého objektu. Tyto výběžky můžeme od běžného středu čáry odlišit tím, že pixely na výběžcích mají menší a postupně klesající vzdálenosti od okrajů. Vzdálenostní mapu i kostru máme již spočítanou z předchozího kroku.

Abychom nejprve zpracovali delší úseky a nezabývali se takovýmito odbočkami, začneme s trasováním v těch bodech kostry, které mají největší vzdálenost od okrajů. Při samotném trasování si pak průběžně udržujeme směr, ve kterém by čára měla pravděpodobně pokračovat. Tuto informaci získáváme z toho, kudy vedla v naposledy vektorizované části křivky. Na počátku trasování jedné čáry žádný směr nemáme a všem možným směrům dáváme stejnou pravděpodobnost. Všechny použité body kostry si označujeme, abychom stejné části netrasovali vícekrát.

3.3.1 Výběr počátečního bodu

Každou cestu začínáme trasovat v bodě kostry, který je ze všech neoznačených nejvzdálenější od okraje objektu. Souřadnice ještě upřesníme tím, že se podíváme i na okolní pixely. Zadefinujeme si k tomu *míru vhodnosti*. Čím je vyšší, tím je bod více uprostřed objektu, a tedy je vhodnější jako střed čáry. Pro konkrétní (neceločíselný) bod C určíme míru vhodnosti jako

$$f(C) := \sum_{P \in Q} \mathrm{dist}(P) \cdot e^{\frac{-||C-P||^2}{2\sigma^2}},$$

kde Q je množina pixelů v okolí bodu C, $\mathrm{dist}(P)$ je vzdálenost pixelu P od okraje objektu (vyčtená ze vzdálenostní mapy) a σ^2 je parametr funkce. Určuje rozptyl odpovídajícího normálního rozdělení. Díky vlastnostem tohoto rozdělení nám stačí pro dostatečně přesnou míru vhodnosti započítat pouze pixely do vzdálenosti 3σ od bodu C. Jako startovní bod použijeme takový, který míru vhodnosti maximalizuje.

Ze startovního bodu následně hledáme cestu. Jakmile ji najdeme, otočíme ji a pokračujeme v hledání. Tím zařídíme, že cesta startovním bodem prochází, takže čáry nemáme zbytečně rozseknuté v jejich nejširších místech.

Aby startovní bod z čáry nikterak nevybočoval, po otočení čáry jej nejprve smažeme. Zaručíme tím lepší hladkost hledané křivky a také nepotřebujeme počáteční bod hledat dokonale přesně. Při hledání maxima zkoušíme jen některé možnosti.

3.3.2 Výběr následujícího bodu

V každém okamžiku trasování máme nalezený souvislý úsek čáry (posloupnost bodů) a snažíme se najít následující nový bod. Vybíráme z několika možností, kudy by čára mohla vést. Varianty nacházejí dva různé prediktory, které si záhy popíšeme. Pro každou z variant spočteme její *fitness* a vybereme tu nejlepší, tj. maximalizující ohodnocující fitness funkci.

Tímto způsobem bychom však ohodnocení počítali pouze lokálně pro nový segment. Bylo by tak možné, že přestože vybereme lokálně nejvhodnější variantu, nebude možné na daný úsek rozumným způsobem navázat. Přitom alternativní bod mohl být pro další pokračování čáry výhodnější. Abychom tento problém omezili, vyzkoušíme pro každou variantu provést vždy několik kroků dopředu a ohodnocujeme až celé delší úseky. Tímto způsobem se program brání přednostnímu trasování popsaných výběžků.

Hladké sledování a rohy

Nejčastěji se snažíme vybrat bod tak, aby nový úsek co nejpřesněji pokrýval všechny mezilehlé pixely ležící v kostře a zároveň byla čára co nejhladší.

Začneme tak, že si vybereme stejný směr, jaký měla čára v předchozím úseku. V tomto směru nakreslíme úsečku s délkou určenou parametrem `nearby_limit`. Pro ni spočítáme její míru vhodnosti podobně, jako jsme počítali míru vhodnosti při hledání počátečního bodu. Jediný rozdíl je, že započítáváme druhou mocninu vzdálenosti pixelu od úsečky (místo od bodu). Nyní vybraný směr postupně upravujeme, abychom míru vhodnosti maximalizovali.

Tento prediktor se sám podle nastavení parametru `smoothness` (maximální povolený rozdíl mezi úhly) rozhodne, jestli nově nalezený úsek navazuje dostatečně hladce, nebo má raději vygenerovat ostré zalomení čáry (roh). Pokud je odchylka předchozího a nového směru menší než tento parametr, považuje se úsek za hladký. Pokud jsme však úpravami směru úsečky tento limit překročili, považujeme místo za roh.

Pokud je úsek hladký, obdobně najdeme také kontrolní body pro Bézierovu křivku. Hledáme ve vzdálenosti `nearby_limit - nearby_control_smooth`.

V případě detekovaného rohu jej ještě musíme přesně najít. Víme jen, že se nachází někde mezi posledním bodem a tím nově vzniklým. Stejným způsobem, jakým hledáme vhodný úhel k pokračování, určíme v nově vzniklém bodě směrnici kostry. Za roh pak považujeme průsečík této nalezené směrnice se směrnicí čáry v předchozím natrasovaném úseku.

Pokračování z rohu a prvního bodu

Pokud vycházíme z rohu či prvního bodu čáry, nemáme zatím určenou žádnou směrnici. V takovém případě zkoušíme hledat ve všech možných směrech. Úhel 2π rovnoměrně rozdělíme na `angle_steps` dílů. V každém směru pak zkoušíme pokračovat stejným způsobem, jako v předchozím případě.

Detekce konců

Prediktory výše se snaží vybrat nejlepší varianty. Pokud však čára končí, je možné, že prediktor vybere sice správný směr, ale neodhadne vzdálenost a kostra skončí dříve, než je prediktorem nalezený nový bod. V tom případě bychom jej chtěli posunout blíže ke kraji kostry a označit jako koncový. Z koncového bodu již nehledáme další pokračování.

3.3.3 Tipování a průchod do hloubky

Jak již bylo naznačeno, samotné prediktory fungují lokálně a mohlo by se nám stát, že se na křižovatce kostry vydáme nevhodným směrem a čára záhy skončí. Abychom tento případ minimalizovali, algoritmus provede až `allowed_depth` kroků dopředu, než daný směr označí za finální.

Pokud by některý z kroků nebyl dostatečně věrohodný (porovnávání s parametrem `depth_auto_choose`), je to pravděpodobně způsobeno tím, že z něj není jak pokračovat dál. Algoritmus se proto vrátí zpět a vybere další lokálně nejlepší směr, který už globálně může mít větší věrohodnost. Ze všech otestovaných možností si pak pamatujeme nejlepší možnou.

Když najdeme dostatečně věrohodnou posloupnost kroků, dále již zbylé možnosti neprohledáváme a rovnou se v daném směru posuneme o jeden krok a celý proces hledání opakujeme.

Tím se dokážeme vyhnout slepým cestám na křižovatkách v případě, že existuje lepší varianta. Na druhou stranu častější tipování dokáže vektorizaci velmi zpomalit.

Při použití Zhangova-Suenova algoritmu na hledání kostry tipování nepotřebujeme zapínat. Kostra je dostatečně jednoduchá na to, abychom si vystačili s výsledkem prvního použitého prediktoru.

3.3.4 Zajištění konečnosti

Snaha o dodržení směru sice zabraňuje tomu, abychom se v jednom bodě otočili a vyrazili zpět po stejné části kostry, po které jsme přišli, ale nezabrání cyklení. Potřebujeme zařídit, abychom jednu čáru netrasovali vícekrát.

Po natrasování čáry bychom mohli použité pixely vyřadit z kostry. Tím však nevyřešíme zacyklení v rámci trasování jedné čáry (představme si obrázek s kružnicí, tu by bylo možné neustále obcházet dokola). Proto pixely kostry označujeme za použité ve chvíli, kdy je využijeme pro nějakou část čáry. Pokud jsme pixely označili v rámci tipování, musíme je zase při návratu odoznačit. Abychom správné pixely poznali, budeme jim přiřazovat číselné popisky s aktuální hloubkou zanoření.

Tím zabráníme cyklení, ale zároveň umožníme plynulé trasování křížících se čar, které mají část kostry společnou. Nezakážeme tedy použití označených pixelů, ale pouze vynutíme, aby vždy alespoň jeden použitý pixel kostry byl neoznačený.

3.4 Fáze 4: Vyhlazování

Po natrasování máme sice již vektorovou podobu obrázku, ale každá jeho cesta je složena z velkého množství bodů. Takový obrázek pak při uložení zabírá zbytečně mnoho místa a ani se s ním v editoru nepracuje snadno. Cesty proto zjednodušíme a tím i vyhladíme drobné nedokonalosti.

V programu před tímto krokem ještě převádíme čáry s proměnlivou šířkou stopy na obvodovou reprezentaci, protože při zjednodušování šířku nejprve zprůměrujeme. Pokud bychom ji neprůměrovali, stejně by k tomu došlo při exportování. Tímto si však zjednoduššíme kód.

Algoritmus vyhlazování je popsán v článku *Approximate conversion of spline curves* (Hoschek, 1987). My jej používáme v lehce upravené podobě. Původní algoritmus nejprve každou Bézierovu křivku rozdělí v několika bodech. Vznikne tím lomená čára, kterou následně aproximuje pomocí jedné křivky. Pokud se aproximace nepodaří, resp. aproximační křivka je příliš vzdálená od těchto bodů, rozdělí ji v bodě s největší chybou a na obě části se zavolá rekurzivně.

V našem programu postupujeme obráceně. Začneme se dvěma segmenty a pokusíme se je aproximovat jedním. Pokud se to podaří, přidáme další segment a zkusíme aproximovat všechny dohromady. Jakmile jednou selžeme, vrátíme se o jeden krok zpět.

Tento způsob sice není tak efektivní, ale je mnohem snazší na implementaci. Výstupem původního algoritmu je totiž hladká cesta (spojitost třídy G^1). My jsme však při trasování mohli nalézt rohy, které vyhladit nechceme. Náš způsob je tak může snadno rozpoznat a v přidávání dalších segmentů se zastaví.

Zbývá popsat, jak se hledá aproximační křivka, máme-li již množinu bodů. Krajní body by měly být shodné s krajními body lomené čáry, nemusíme je tedy hledat. Stejně tak pro zachování spojitosti máme v těchto krajních bodech určené směrnice. Zbývají nám dva volné parametry – vzdálenosti kontrolních bodů Bézierovy křivky (podle značení z kapitoly 1.1 to jsou vzdálenosti $|P_0 - P_1|$ a $|P_2 - P_3|$.)

Ty můžeme určit metodou nejmenších čtverců, pokud známe správné časy bodů, které jsou na hledané křivce nejbližší k bodům lomené čáry. Na začátku však časy neznáme. Odhadneme je proto ze vzdálenosti jim odpovídajících bodů na lomené čáře (tj. součtem délek všech úseků před bodem vydělený délkou celé lomené čáry). Pro tyto časy spočítáme nové vzdálenosti kontrolních bodů.

Nyní můžeme definovat chybové vektory:

$$\delta_i = A(t_i) - P_i,$$

kde A je nalezená Bézierova křivka, P_i body na lomené čáře a t_i časy odpovídající jednotlivým bodům. Správnou parametrizaci máme pouze tehdy, kdy jsou chybové vektory kolmé na tečny ke křivce.

Časy proto posuneme ve správném směru a tím parametrizaci vylepšíme. Poté znovu metodou nejmenších čtverců nalezneme vzdálenosti kontrolních bodů a celé to několikrát zopakujeme.

Algoritmus se zastaví ve chvíli, kdy je velikost největšího chybového vektoru menší než konstanta daná parametrem `approximation_error`. (Případně se zastavi s neúspěchem, pokud je překročen maximální povolený počet iterací).

3.5 Fáze 5: Export

Po celou dobu vektorizační algoritmus počítá s tím, že čára může mít proměnlivou šířku. To ovšem není možné přímo zapsat do výstupních formátů. Proměnlivá šířka Bézierovy křivky není podporována ani v SVG, ani v PS.

Máme několik možností, jak se s problémem vypořádat. Každá z následujících metod je vhodná v jiné situaci. Nejúčinnější tedy je výstupní metody zkombinovat a každou čáru vypisovat pro ni nejlepším způsobem. Není však snadné strojově poznat, která z variant je pro danou čáru nejvhodnější. Program se toto snaží

odhadnout spočítáním rozptylu šířky každé čáry. Pokud překročí mez určenou parametrem `auto_contour_variance`, použije se raději obvodová reprezentace. Nastavením extrémně velké / záporné hodnoty můžeme vynutit danou reprezentaci u všech čar.

3.5.1 Průměrná šířka

První možností je šířku jednotlivých segmentů čar zprůměrovat a poté už exportovat celou čáru s jednotnou šířkou. Tím v některých případech ztratíme důležitou část informace. Zabráníme ale prvním vektorizační artefaktům – bezdůvodně se měnící šířce. Tento druh exportu je však vhodný pro obrázky, které obsahují pouze čáry s konstantní šírkou.

3.5.2 Rozdělení cest na úseky

Druhou variantou je čáry rozdělit po jednotlivých segmentech na samostatné objekty. U formátu SVG si můžeme pomoci tím, že objekty vzniklé z jedné čáry vložíme do společné skupiny. Rozdělením se však následná práce s výstupem stane náročnější. Stále tímto způsobem nedokážeme dokonale reprezentovat libovolná data, například jednu obyčejnou rozšiřující se úsečku. I proto se tento druh exportu nepoužívá, přestože je v programu implementovaný. Můžeme jej vynutit pro všechny čáry nastavením parametru `export_type` na `1`.

3.5.3 Obvodová reprezentace

Třetí variantou je převedení čar na plochy. Editovatelnost dané čáry pak bude obdobná jako u algoritmů hledajících plochy. Při správném nastavení parametru však tímto způsobem budou reprezentovány pouze ty čáry, u kterých je to nutné.

Samotný převod ovšem není zcela jednoduchý problém. Obvod Bézierovy křivky (dvě offsetové[2] křivky a zakončení na obou koncích) nelze přesně reprezentovat Bézierovými křivkami. Je tedy nutné najít vhodnou aproximaci.

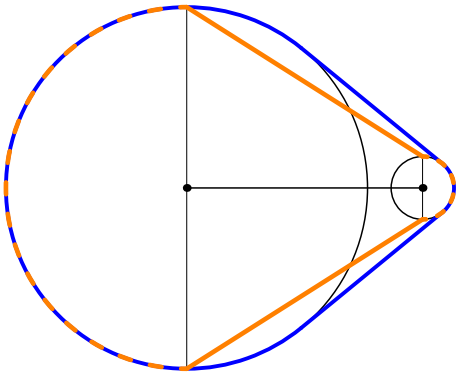
V našem případě je problém ještě komplikovanější, protože se šířka křivky mění. Musíme si dát také pozor na přesný význam šířky v daném bodě. Běžná interpretace je, že pokud má křivka v daném bodě X šířku w, potom existuje obvodový bod, který leží na normále procházející bodem X a zároveň je od něj vzdálený $w/2$.

V naší interpretaci však šířka odpovídá průměru vložené kružnice (osmiúhelníku) se středem v daném bodě. Při měnící se šířce se tento průměr kružnice nerovná kolmé vzdálenosti k okraji. Rozdíly mezi oběma významy šířky jsou patrné z obrázku 3.5.

Offsetování Bézierovy křivky

Jednu z možných metod offsetování popisuje článek *Spline Approximation of Offset Curves* (Hoschek, 1988). Ta je v mnohém podobná aproximování, které používáme v předchozí fázi. Liší se tím, že před samotnou aproximací body, ve kterých počítáme chyby, posuneme ve směru kolmém k offsetované křivce. Tuto

[2]Offsetovou křivkou rozumíme křivku, která je od původní v každém bodě stejně vzdálená.



Obrázek 3.5: Úsečka s proměnlivou šířkou stopy: modře je zakreslen obvod úsečky v případě, že šířka určuje průměr kružnice; oranžový obvod odpovídá častější kolmé vzdálenosti

úlohu však nepotřebujeme řešit, protože čáry s konstantní šířkou nemusíme převádět na obvodovou reprezentaci.

Zajímá nás však převod čar s proměnlivou šířkou. Začátek algoritmu k tomu drobně upravíme. Nemůžeme pouze posunovat každý bod X po normálovém vektoru o vzdálenost odpovídající příslušné šířce – vzhledem k naší interpretaci šířky musíme normálu nejprve správně natočit.

Natočení (a tedy i správnou polohu posunutého bodu X') můžeme spočítat přesně. Při (nekonečně) malé změně času δ_t se bod $X = C(t)$ po křivce C lehce posune a drobně se změní jemu odpovídající šířka $w(t)$. Hledaný offsetový bod X' pak leží na společné tečně kružnic o průměrech $w(t)$ a $w(t + \delta_t)$ se středy v $C(t)$ a $C(t + \delta_t)$.

Aby byl výpočet numericky stabilnější, řešení si zjednodušíme. Do každého z aproximovaných bodů na původní (neoffsetové) křivce si umístíme kružnici se správným průměrem a následně najdeme společné tečny pro každé dvě sousední kružnice. Z každé tečny použijeme bod, který leží na větší z kružnic.

4. Srovnání výsledků

Zhodnocení navrženého algoritmu a porovnání jeho kvality s jinými není u vektorizace snadná úloha, obzvlášť pokud bychom toto chtěli vyjádřit číselně.

4.1 Měřítka kvality

Můžeme třeba pro každý testovací obrázek určit počet pixelů, v nichž se vektorová podoba liší od originálu. S takovýmto měřítkem kvality by ale vyhrával vektorizér, který každý pixel na vstupu vektorově reprezentuje jako čtverec stejné barvy o rozměrech 1×1. Celý obrázek by pak byl složen právě z takovýchto čtverců.

Výstup můžeme také hodnotit podle toho, z kolika primitiv se skládá. Počet objektů přímo souvisí s velikostí výstupního souboru, takže pro nás toto kritérium může být důležité. V nástrojích však zjednodušování a vyhlazování křivek často bývá jako jedna z posledních fází vektorizace, navíc ovlivnitelná parametrem. Například u programu Potrace se odpovídající parametr nazývá `opttolerance`. V našem programu máme tomu podobný `approximation_error`. Nelze tedy jednoduše říct, že program `A` vytváří meší výstup než program `B` a je proto lepší.

Dvě popsané metriky bychom mohli spojit do jedné: určíme odchylku od předlohy pro takové nastavení parametrů, kdy oba porovnávané programy produkují stejné množství bodů. Potom si musíme položit otázku, zda tímto způsobem opravdu hodnotíme celý program, nebo pouze část starající se o vyhlazování.

4.1.1 Typ reprezentace

Pro nás je mnohem důležitější ještě jeden faktor. Tím je zcela odlišná reprezentace čárové a plošné grafiky. Čárové vektorizéry nejsou příliš obvyklé. Mnohem častější je reprezentace objektů plochami.

4.2 Porovnání s existujícími nástroji

Porovnávat budeme výsledek naimplementovaného programu Vectorix s existujícím volně dostupným programem Potrace (Selinger, 2015) a komerčními nástroji Vector magic (Cedar Lake Ventures Inc.) a RasterVect. U posledních dvou se ještě musíme vypořádat s tím, že k dispozici máme pouze zkušební verze programů, které oproti plným neumožňují výsledek ukládat.

Nebudeme porovnávat výsledky s dalším volně dostupným nástrojem Autotrace (Weber, 2004). Protože má podle nás znatelně horší výsledky než Potrace.

Nesmíme zapomenout, že celou dobu pracujeme s obrzovými daty a že i výsledný vektorový obrázek je určen k tomu, aby se na něj dívali lidé. Kvalita vektorizace nemusí být ani nijak matematicky popsatelná. Můžeme se rozhodovat subjektivně podle toho, který obrázek je „hezčí“. (Případně který obrázek se „snáze“ upravuje.)

Nástroje tedy necháme zvektorizovat stejný obrázek a následně popíšeme, čím se výsledky kterého od sebe liší.

Na následujících stránkách vidíme výsledky jednotlivých vektorizátorů. Program Vectorix si neporadí dobře s očima hrocha, protože se již jedná o plochu. Oproti výstupu z Potrace se však celý obrázek skládá z výrazně méně uzlů. Ve Vectorixu totiž bylo nastaveno poměrně silné zjednodušování, i proto jsou čáry kostrbatější.

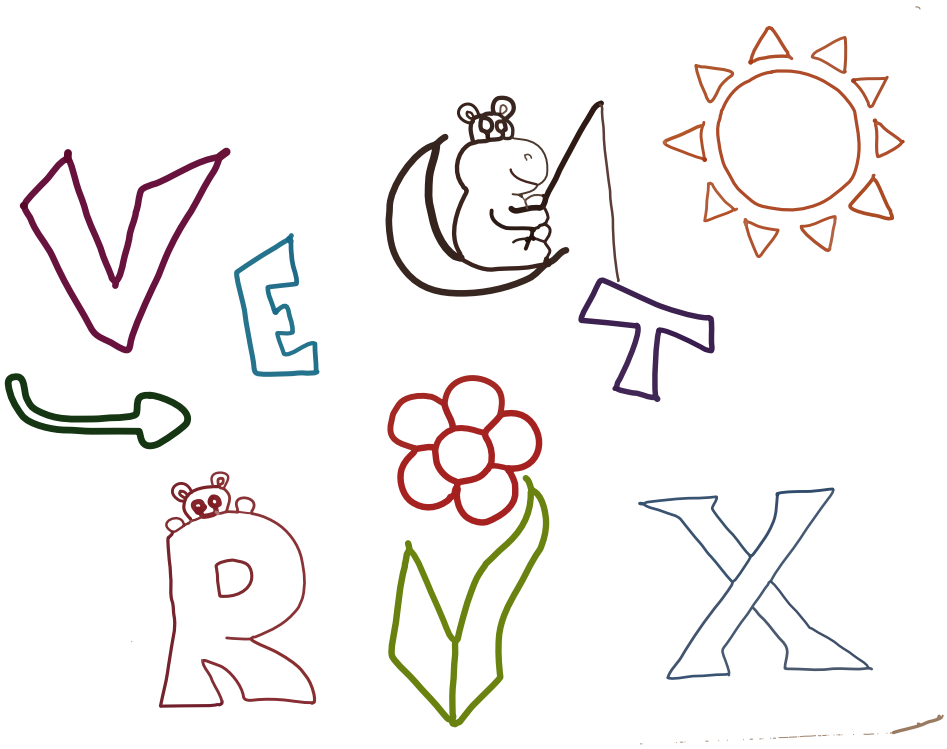
Výstup z Vector Magicu na první pohled vypadá velmi podobně originálu. Při bližším pohledu si všimneme nedostatků jako překrývající se čar (šedá a černá) a jejich proměnlivé šířky.

Potrace zvládá oproti Vector Magicu lépe detaily očí, ale naopak má kostrbatější čáry.

Metodou vektorizace jsou však Potrace a Vector Magic od zbylých dvou programů odlišné. Pouze Vectorix a RasterVect reprezentují objekty pomocí čar. Barevné obrázky RasterVect nezvládá čárově trasovat vůbec. Pokud z něj nejprve uděláme černobílý, objekty již kreslí lépe. Skládají se však pouze z mnoha malých úseček. Počty úseček/uzlů u RasterVectu a Vector Magicu jsou bohužel neznámé, protože tuto hodnotu programy neuvádí.

Rychlost jednotlivých vektorizátorů je srovnatelná. Probíhá řádově do minuty.

Obrázek 4.1: Vyfotografovaný vstupní obrázek

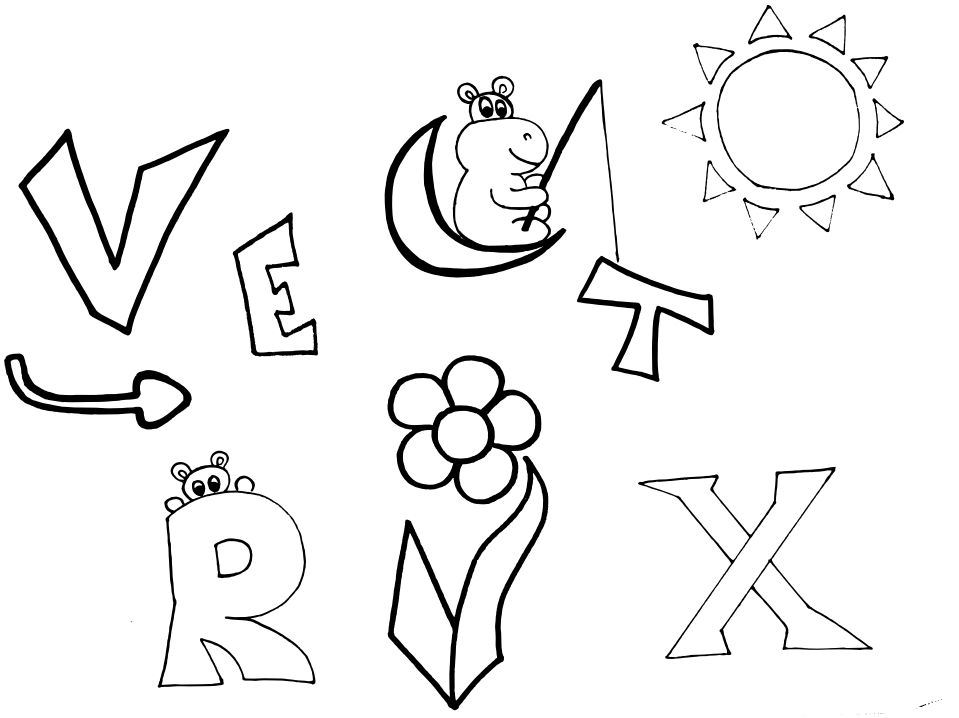


Obrázek 4.2: Výstup z programu Vectorix: Je složený ze 173 objektů a dohromady obsahuje 1 007 uzlů

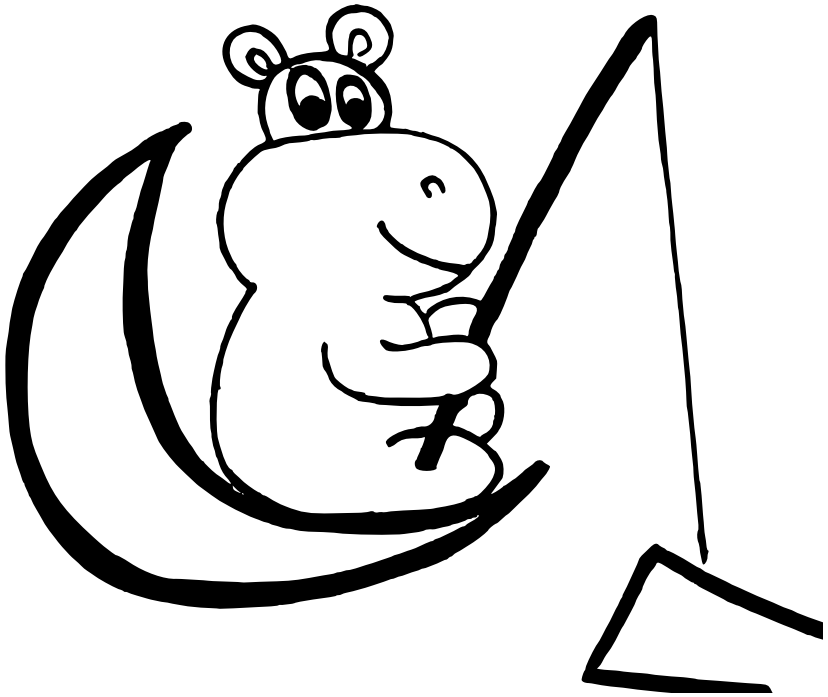
Obrázek 4.3: Detail výstupu z programu Vectorix: Z obrázku jsou patrné nedostatky na ostrých rozích čar a u vyplněných oblastí

Obrázek 4.4: Vektorizace nástrojem Vector Magic: Obrázek byl rozpoznán jako fotografie

Obrázek 4.5: Detail výstupu z programu Vector Magic



Obrázek 4.6: Vektorizace programem Potrace: Obrázek obsahuje 6590 uzlů



Obrázek 4.7: Detail výstupu z programu Potrace

Obrázek 4.8: Barevná vektorizace nástrojem RasterVect: S barevnými obrázky si program při metodě centerline příliš nerozumí

Obrázek 4.9: Vektorizace nástrojem RasterVect: Úspěšná vektorizace černobílého obrázku, modré čáry jsou nalezená vektorová reprezentace

5. Uživatelská dokumentace

V této kapitole se s jednotlivými částmi programu seznámíme v pořadí, které odpovídá předpokládanému běžnému používání.

5.1 Instalace

Program Vectorix je psaný v jazyce C++11, je tedy nezbytné mít překladač, který tento dialekt podporuje. Nejsou však využívány všechny nové funkce, takže by měl posloužit kterýkoli dnes běžný. Vývoj probíhal s překladačem GCC (Stallman a kol., 2014) ve verzi 4.9.2.

Dále program vyžaduje knihovnu na zpracování obrazu OpenCV (2015, Open Source Computer Vision). Je možné využít jak řadu 2.4.x, tak 3.0.x. Pokud knihovna není nainstalována v systému, je možné si ji stáhnout a nainstalovat lokálně do podadresáře programu Vectorix. Zdrojový balíček verze 3.0.0 pro Linux/Mac je rovnou přiložen.

Volitelně je možné použít také vektorizaci pomocí knihovny Potrace. V takovém případě je nutná i její instalace. Při běžném používání programu není potřeba a ani se v současnosti nevyužívají všechny její možnosti. Její přítomnost je spíše příprava do budoucna na možné rozšíření (viz kapitola 7.2).

Zdrojový balíček nejprve rozbalíme:

```
> tar -xzf vectorix.tar.gz
```

Před instalací knihovny OpenCV je vhodné se přesvědčit, že máme vše potřebné pro její kompilaci. Zde je nejvhodnější odkázat na podrobné návody na webových stránkách knihovny[1]. Na Linuxu (Debian a Ubuntu) by měly být potřeba následující balíčky:

```
build-essential cmake git libgtk2.0-dev pkg-config libavcodec-dev
libavformat-dev libswscale-dev libjpeg-dev libpng-dev libtiff-dev
libjasper-dev
```

Nyní již knihovnu nainstalujeme. Abychom ji nemuseli stahovat, pouze její zdrojový balíček překopírujeme k programu. Tento krok (první řádek) můžeme přeskočit, knihovna se pak stáhne v následujícím kroku sama.

```
> cp opencv-3.0.0.zip vectorix/opencv/
> cd vectorix/opencv
> make opencv-3.0.0.zip include-3.0.0 lib-3.0.0
    (případně)
> make opencv-2.4.11.zip include-2.4.11 lib-2.4.11
```

Používáme-li OpenCV ve verzi 3.0.0 (tj. přiloženou), můžeme rovnou zkompilovat program:

```
> cd ..
> make
```

[1] http://opencv.org/quickstart.html

Ostatní verze OpenCV

Chceme-li použít jinou verzi OpenCV, stačí upravit dva řádky v souboru `vectorix/Makefile`. Druhá varianta předpokládá, že OpenCV je nainstalována v systému.

```
# select OpenCV library version
L_OPENCV=${L_OPENCV_2.4.11}
C_OPENCV=${C_OPENCV_2.4.11}
    (nebo)
L_OPENCV=${L_OPENCV_SYSTEM}
C_OPENCV=${C_OPENCV_SYSTEM}
```

Použití knihovny Potrace

Pro kompilaci knihovny Potrace stačí spustit v adresáři `vectorix/potrace` příkaz `make`. Knihovna se sama stáhne.

```
> cd vectorix/potrace
> make
```

Aby se knihovna Potrace v programu použila, je po jejím nainstalování potřeba odkomentovat následující dva řádky v souboru `vectorix/Makefile`:

```
L_FLAGS+=-L potrace/lib/ -lpotrace
C_FLAGS+=-D VECTORIX_USE_POTRACE
```

5.2 Ovládání programu

Vektorizaci je možné přizpůsobovat parametry, které lze specifikovat v konfiguračním souboru a některé i v grafickém rozhraní. Pokud program spustíme bez parametrů, zeptá se nás na název konfiguračního souboru a uloží do něj výchozí hodnoty parametrů.

```
> ./vectorix
No config file given, new will be created, please enter name:
```

5.2.1 Konfigurační soubor

Konfigurační soubor má velmi jednoduchou strukturu. Každý řádek je buď komentář (začíná znakem `#`), nebo obsahuje dvojici `<klíč> <hodnota>` oddělenou mezerou. Hodnota je buď číslo (celé / desetinné s tečkou), nebo řetězec. Pokud se některý klíč vyskytuje v konfiguračním souboru vícekrát, vždy se použije jeho poslední výskyt.

Při ukončení programu je konfigurační soubor uložen do souboru určeného parametrem `file_parameters`, pokud je specifikovaný. Ukládají se pouze parametry relevantní pro daný běh programu. To například znamená, že pokud vektorizujeme pomocí knihovny Potrace, neuloží se parametry týkající se vektorizace

naším algoritmem. Toto lze obejít tak, že parametry necháme připisovat na konec souboru (`parameters_append 1`).

Pro začátek však potřebujeme nastavit jen jméno souboru se vstupním rastrovým obrázkem a souboru s výstupním vektorovým. Vstupní obrázek je určen parametrem `file_input` a může být v libovolném formátu podporovaném OpenCV (z nejznámějších BMP, PNM, JPEG, TIFF a PNG). Výstupní obrázek je uložen do souboru dle parametru `file_vector_output`. Standardně se výstup ukládá ve formátu SVG. Pro PostScriptový výstup je třeba nastavit `output_engine 1`.

Čtení obrázku pomocí knihovny OpenCV je možné obejít. Náš program implementuje načítání obrázků ve formátu PNM (Henderson, 2013). K tomu se používá parametr `file_pnm_input`. Tento parametr je důležitý pro experimentální vektorizační metody (např. knihovnou Potrace), protože ty knihovnu OpenCV nepoužívají.

5.2.2 Grafické rozhraní

Máme-li v konfiguračním souboru specifikovaný vstupní soubor, můžeme program spustit znovu. Tentokrát mu jako parametr dáme připravený konfigurační soubor.

```
> ./vectorix konfigurace.conf
```

Program se spustí v interaktivním grafickém režimu. K dispozici máme 4 okna. První je nazvané *Original*. V něm vidíme (zmenšený) původní obrázek s červeným rámečkem označujícím viditelnou oblast obrázku ve všech ostatních oknech. Pomocí posuvníku *Zoom* pak můžeme obrázek přiblížit, či oddálit, myší vybíráme pozici zobrazeného výřezu.

Pří největším přiblížení (100) odpovídá přiblížení původní velikosti obrázku – jeden pixel na obrazovce je tedy jeden pixel obrázku. Z toho plyne, že posuvník *Zoom* nemá žádný vliv při zobrazení obrázku menšího, než je velikost oken.

Prahování

Jako druhé nás bude zajímat okno *Grayscale*. V něm je možné invertovat vstupní barvy. V další fázi potřebujeme, aby čáry byly bílé a pozadí černé. Protože očekávaný běžný obrázek má bílé pozadí a tmavé linie, je inverze ve výchozím nastavení zapnutá.

Třetí okno *Threshold* umožňuje ladit parametry prahování. Při nastavení typu prahu (*Threshold type*) na 0 se používá Otsova metoda (kap. 3.1.1) pro určení efektivního prahu a zbylé dva posuvníky tak nemají žádný vliv na výsledek. Chceme-li zvolit fixní práh (posuvníkem *Threshold*), musíme nastavit *Threshold type* na 1.

Zbylé dva typy prahování využijeme při nerovnoměrném rozložení jasu v obrázku. Typicky to potřebujeme u fotografií (nikoli však skenů) obrázků nakreslených na papír. Většinou chceme použít typ 3, který odpovídá adaptivnímu prahování, kde hodnota prahu je pro každý pixel spočtena z průměrné hodnoty na okolí o poloměru *Adaptive threshold* pixelů (s přičteným offsetem podle posuvníku *Threshold*).

Zbývající typ 2 funguje obdobně, jenom okolní pixely započítává s vahou podle normálního rozdělení. Pozor na to, že tato varianta je pro větší okolí pomalá. Pro obrázky s rozměry kolem tisíců pixelů je vhodné udržet posuvník *Adaptive threshold* pod hodnotou 100. Tato varianta však neposkytuje výrazně kvalitnější výsledky než typ 3, takže ji většinou nevyužijeme.

Obecně při prahování chceme vybrat takové parametry, aby všechny linie byly co nejzřetelnější. Měly by pokud možno obsahovat co nejméně „děr“. Zároveň je vhodné minimalizovat množství šumu, „bílých teček“, v obrázku.

Filtrování šumu

V posledním ze čtyř oken (*Filled*) můžeme čáry zacelit posuvníkem *Filling size*. Pozor na to, že při vyšších hodnotách se mohou začít slévat blízké čáry. Míru odstranění bílých teček lze ovlivnit pomocí *Dust removal size*. Zde si naopak musíme dát pozor, abychom neodstranili i nějakou čáru.

Jakmile jsme s výsledkem spokojeni, můžeme stisknutím klávesy Enter přejít do další fáze vektorizace.

Kontrola skeletonizace

Objeví se nám dvě nová okna, jedno s kostrou (*Skeleton*) a jedno se vzdálenostní mapou (*Distance*). Jediným posuvníkem *Skeletonization* si můžeme vybrat z pěti variant algoritmů hledání kostry. Typ 0 vkládá střídavě diamant a čtverec, typ 1 čtverec, typ 2 diamant. Následující typ číslo 3 je nejpomalejší, protože dochází k vkládání kruhů. V praxi se od typu 0 kvalitou výstupu téměř neliší, ale je výrazně pomalejší. Této variantě je dobré se vyvarovat, pokud obrázek obsahuje širší čáry (či vyplněné objekty).

Ve většině případů používáme výchozí hodnotu (4), která odpovídá Zhangovu-Suenovu algoritmu a vzdálenostní mapě počítané vkládáním střídavě diamantu a čtverce.

Každou změnu parametrů je potřeba potvrdit stiskem klávesy Enter, jinak se kostra nepřepočítá. Dalším stisknutím přejdeme do fáze trasování. Parametry této fáze jsou většinou neceločíselné. Grafické rozhraní tvořené knihovnou OpenCV bohužel neumožňuje jejich nastavování. Naštěstí tyto parametry není pro většinu obrázků potřeba nijak přenastavovat.

5.2.3 Další důležité parametry

Přehled všech existujících parametrů s jejich stručným vysvětlením se nachází v příloze 1. Zde následuje popis šesti parametrů, u kterých je nejpravděpodobnější, že je potřebuje uživatel měnit.

- `interactive`: Nastavením hodnoty 0 se vypne interaktivní režim a nebudou zobrazena žádná okna. Program je následně použitelný pro spouštění ze skriptů a vektorizuje zcela bez dalších zásahů uživatele.

- `max_window_size`: Nastavuje maximální rozměry oken (počítáno bez posuvníků). Výchozí hodnotu (640 pixelů) může být vhodné na menších monitorech snížit. Ze všech obrázků je následně zobrazen výřez nebo zmenšenina.

- `force_black`: Pokud je tento parametr nenulový, všechny čáry jsou obarveny na černo. Ve výchozím nastavení je tento parametr zapnut, protože předpokládáme černobílé obrázky.
- `auto_contour_variance`: Parametr upravuje typ exportování. S nižší hodnotou parametru je více čar ve výstupu reprezentováno pomocí jejich obvodu.
- `approximation_error`: Hodnota tohoto parametru určuje maximální povolenou chybu při zjednodušování čar, větší číslo znamená větší míru zjednodušování, a tedy čáry složené z menšího počtu úseků.
- `approximation_preserve_corners`: Pokud je parametr nenulový, jsou při vyhlazování všechny rohové body zachovány.

6. Vývojová dokumentace

Program Vectorix, jehož zdrojový kód se nachází v elektronické příloze této práce, je psán v jazyce C++ a využívá některé novinky z verze C++11. Pro práci s obrazovými daty je použita knihovna OpenCV (2015). Vše z knihovny OpenCV se nachází v namespace `cv`. Kromě standardní knihovny jazyka již program není závislý na žádných dalších. (S výjimkou volitelného rozšíření o Potrace.)

OpenCV poskytuje velké množství různých obrazových operací a algoritmů, takže s její pomocí si lze ušetřit spoustu práce při zpracování rastrových obrázků a zejména při implementování a zkoušení nových algoritmů. Občas je však výhodnější nepoužít knihovní funkci a naimplementovat si vlastní, protože můžeme těžit z dalších předpokladů, které v našem případě platí. Příkladem je skeletonizace (kap. 3.2.2), kde přímá implementace algoritmu může využívat operace dilatace a eroze z knihovny. Nebo můžeme celý proces zrychlit, omezíme-li operace jen na některé pixely (optimalizace pomocí fronty).

6.1 Dělení na funkční bloky

Algoritmický návrh je popsán v kapitole 3, nebudeme jej tedy zbytečně opakovat. Zaměříme se místo toho na zajímavé „implementační detaily“ a celkové členění kódu na jednotlivé funkční bloky. Ty zároveň korespondují s rozdělením kódu do zdrojových souborů.

Parametry (`parameters.cpp`)

Jednotlivé části jsou často parametrizovatelné uživatelem. Ten parametry zadává do jednoho společného konfiguračního souboru. O jeho správu se stará třída `parameters`, kterou si objekty mezi sebou předávají. (Je tedy možné mít více různých instancí, nicméně v praxi se to neděje.) Tato třída přitom sama žádné konkrétní parametry nedefinuje; ostatní části programu si je u ní postupně registrují za běhu. Pokud si řekneme (například z dvou různých míst) o stejně pojmenovaný parametr, dostaneme vždy ukazatel na stejnou proměnnou.

Díky tomu je velmi snadné přidat kdekoli v kódu nový parametr. Stačí zavolat funkci `bind_param(ukazatel, název, defaultní hodnota)`, která do ukazatele uloží odkaz na proměnnou s hodnotou parametru odpovídajícího názvu. Pokud jsme první, kdo si řekl o proměnnou s tím to názvem a tato není v konfiguračním souboru specifikována, je do proměnné uložena defaultní hodnota.

Konfiguraci můžeme kdykoli načíst ze souboru nebo do něj naopak uložit (funkce `load_params` a `save_params`). Při načtení se zaktualizují všechny již zaregistrované parametry. Doposud neregistrované se pak samy načtou až s první registrací. Ukládání parametrů probíhá v tom pořadí, v jakém byly (poprvé) registrovány. Díky tomu je možné přidávat do konfiguračního souboru k jednotlivým parametrům také komentáře (v souboru začínají znakem `#`). Přidáme je funkcí `add_comment(komentář)`.

Grafické rozhraní (`zoom_window.cpp`)

Program využívá pouze funkcionalitu knihovny OpenCV, která v tomto směru nenabízí příliš. Funkcí `imshow(jméno okna, obrázek)` vytvoříme okno a zobrazíme v něm daný obrázek. Do oken lze dále přidávat posuvníky. Okna však nelze rozumným způsobem zmenšovat. Větší obrázky se tak vůbec nemusí vejít na monitor, obzvlášť, pokud chceme mít otevřených několik oken současně.

Nedostatek řešíme funkcí `zoom_imshow(jméno okna, obrázek, přehled = false)`, která zobrazí jeho výřez, jenž následně dle potřeby zmenší. Aby se dalo v rámci obrázku navigovat, potřebujeme jedno speciální okno s přehledem celého obrázku a se zakresleným umístěním výřezu. To také obsahuje posuvník nastavující úroveň přiblížení. Přehledové okno lze vytvořit zavoláním `zoom_imshow` s třetím parametrem nastaveným na true.

O vše potřebné se stará třída `zoom_window`, která je singletonem a její instanci zbylé části programu nikdy nezískají. Protože velikost okna lze určit parametrem, je potřeba před prvním vytvořením okna předat třídu s parametry (zavoláním funkce `zoom_set_params`).

Textové hlášky (`logger.h`)

S výpisem různých typů hlášek pomáhá třída `logger` s metodou `log<typ hlášky>` fungující ve stylu klasického `printf`. Vypisují se však pouze hlášky, které mají dostatečnou závažnost. Před vypsáním se překontroluje, že hláška je vzhledem k aktuálnímu nastavení upovídanosti dostatečně důležitá. Aby nás při normálním běhu programu nezdržovaly ladící hlášky (a ani jejich kontrola důležitosti), je v souboru `config.h` definována konstanta `VECTORIX_MAX_VERBOSITY` udávající nejvyšší povolenou třídu hlášek. Pokud ji nastavíme například na úroveň `warning`, může kompilátor rovnou všechny výpisy typu `info` a `debug` zahodit při překladu.

6.1.1 Datové struktury

Program parcuje jak s rastrovými, tak vektorovými daty. Rastrové obrázky se vyskytují ve dvou základních podobách. Buď jsou uložené ve třídě `cv::Mat`, nebo `pnm_image`. První je třída z knihovny OpenCV, která umí reprezentovat libovolná maticová data. V tomto formátu jsou ukládána data uvnitř vektorizéru.

Rastrové obrázky (`pnm_image.cpp`)

Druhá třída `pnm_image` slouží pro práci s obrázky typu Netpbm (Henderson, 2013). Jedná se již o vlastní implementaci. Protože obrazový formát je velmi jednoduchý, není ani tato implementace složitá. Obrázky se rozlišují na barevné, v odstínech šedi a černobílé bitmapy. Každá z těchto varinat může být uložena binárně, či textově.

Zajímavostí je, že v černobílé podobě reprezentuje hodnota `0` bílé pixely a `1` černé. Při binárním uložení černobílých obrázků se jeden byte skládá z osmi po sobě jdoucích pixelů.

Obrázky ve třídě `pnm_image` lze načítat ze souboru (funkce `read`), zapisovat (`write`) či mezi sebou převádět (`convert`). Není implementován pouze převod

z binární černobílé bitmapy. Tento formát však stejně není příliš častý. Většinou máme obrázky barevné.

Vektorové obrázky (`v_image.cpp`)

Vektorové obrázky vyžadují neceločíselné hodnoty. Je pro ně zadefinovaný datový typ `p`, který ve skutečnosti odpovídá typu `double`. Teoreticky je tak možné přesnost reprezentace snadno změnit. (V praxi to však asi nijak nevyužijeme.)

Samotné vektorové obrázky se v programu reprezentují pouze pomocí kubických Bézierových křivek. Jednomu obrázku odpovídá třída `v_image`. Každý obrázek má svou šířku a výšku (`width` a `height`) a seznam (`std::list`) cest (`line`). Pro ladící účely si u obrázku ještě pamatujeme cestu k souboru, který může být zobrazený na pozadí (`underlay_path`) a seznam cest, které byly do obrázku přidány při ladění (`debug_line`). Tento seznam lze připodobnit ke grafickým ladícím výpisům.

Každá cesta (datový typ `v_line`) pak obsahuje seznam jednotlivých úseků, resp. kontrolních bodů (`segment`), typ (`type_`) určující, zda se jedná o čáru či vyplněnou plochu a typ skupiny (`group_`). Nastavením typu skupiny lze několik po sobě jdoucích čar v SVG výstupu sloučit do jedné skupiny. U obvodové reprezentace mají skupiny speciální význam. První cesta odpovídá tmavé oblasti a všechny následující dírám v ní.

Kontrolní body (třída `v_point`) odpovídají jednotlivým předělům mezi segmenty Bézierových křivek. Ty se setkávají v bodě `main`, který je zároveň pro první křivku čtvrtým, tj. posledním kontrolním bodem a pro druhou křivku prvním kontrolním bodem. Druhý, resp. třetí kontrolní bod každého segmentu je uložený v `control_next`, resp. `control_prev`. Pokud se na data podíváme z pohledu jednoho segmentu, jsou jeho první dva kontrolní body uloženy v jednom objektu třídy `v_point` (jako `main` a `control_next`) a další dva ve druhém objektu (`control_prev` a `main`). Třída `v_point` v sobě dále uchovává informaci o barvě, šířce čáry a průhlednosti (`color`, `width` a `opacity`).

Jednotlivé body v prostoru pak jsou reprezentovány typem `v_pt`, který má souřadnice `x` a `y`. Barvy si pamatujeme třísložkově ve třídě `v_co`.

Manipulace s vektorovými daty (`geom.cpp`)

Některé geometrické operace tematicky nespadají pod žádnou konkrétní třídu. (Například protože pracují s jedním segmentem Bézierovy křivky, který je reprezentován dvěma objekty `v_point`.) Tyto operace jsou proto ve vlastním namespace `geom`. Mezi nejdůležitější patří funkce:

- `bezier_chop_in_t`: Tato funkce rozpůlí Bézierovu křivku v bodě určeném parameterem `t`.
- `bezier_maximal_length`, `bezier_minimal_length`: Funkce odhadují délku Bézierovy křivky délkou kontrolního polygonu, resp. vzdáleností prvního a posledního kontrolního bodu.
- `bezier_intersection`: Jako vstup bere dvě Bézierovy křivky a vrátí `true`, pokud mezi nimi existuje průsečík. Ten se nachází na pozici určitelné podle nalezených parametrů `t`.

- `auto_smooth`: Nastaví všechny body `control_prev` a `control_next` na cestě tak, aby výsledná křivka byla hladká.

6.1.2 Vektorizace

V práci je implementována vektorizace pomocí popsaného vektorizačního algoritmu. Mimo to je také možné použít knihovnu Potrace (pokud ji máme nainstalovanou a povolíme její používání při překladu). V současnosti se však nedají nijak ladit její parametry, takže možnosti jsou značně omezené. Přítomnost knihovny je příprava na jedno z možných rozšíření programu, kde by se mohla použít na vektorizaci některých objektů.

Aby se jednotlivé vektorizéry příliš nelišily a byly použitelné ve stejných případech, byl zde (a na dalších podobných místech) použit návrhový vzor Template method. Existuje proto abstraktní třída `vectorizer` (`vectorizer.cpp`) definující rozhraní vektorizérů (metoda `vectorize`).

V práci navržený postup je použit ve vektorizátoru `vectorizer_vectorix`. Ten postupně provádí jednotlivé fáze algoritmu. Pokud pracuje v interaktivním režimu, stará se také o návrat na správné místo v případě změny parametru některé dřívější fáze.

Jednotlivé fáze pak mají velmi podobná rozhraní. Pro samotný výpočet slouží funkce `run` s argumenty odpovídajícími potřebám daného kroku. Používá-li daná fáze ladění parametrů v grafickém rozhraní, má také metodu `interactive`. Ta přijímá jako parametr funkci, která je zavolána při změně libovolného parametru. Jedná se o stejný způsob, jaký používá knihovna OpenCV pro oznámení o změně hodnoty posuvníku. To není náhoda, v současnosti se předávaná funkce volá právě pouze přes posuvníky OpenCV.

Prahování (`thresholder.cpp`)

Prahování je velmi jednoduché, protože na všechny operace tohoto kroku se přímo používají funkce z OpenCV.

Skeletonizace (`skeletonizer.cpp` a `zhang_suen.cpp`)

Pro hledání morfologické kostry je potřeba nejprve obrázek doplnit o černý rámeček, následně provést vybraný algoritmus a na závěr můžeme rámeček odebrat. Algoritmus vkládání čtverce a/nebo diamantu je implementovaný s pomocí fronty pixelů ležících na hranici. Ve frontě jsou ty pixely, které jsou bílé a mají černého souseda. Protože se střídáním kroků definice sousednosti mění, jsou ve frontě ty pixely, které mají černého souseda podle osmiokolí (čtverec).

V případě vkládání diamantu tedy ještě nejprve znovu kontrolujeme barvu sousedů. Pixel se tak ve frontě nemusí objevit pouze jednou, ale maximálně dvakrát.

U Zhangova-Suenova algoritmu se používá obdobná fronta. Její význam se však maličko liší. Pixel je zde ve frontě, pokud je bílý a v jedné ze dvou předchozích iterací se mu změnil soused (v prvním kroku: alespoň jeden z jeho sousedů je černý). Abychom jeden pixel nepřidávali do fronty vícekrát, značíme si navíc pixely v pomocné matici `inq` (in queue).

Značení pixelů při trasování (`tracer_helper.cpp`)

V průběhu trasování si potřebujeme jednotlivé pixely kostry značit. Aby se nám se značkami a kostrou lépe pracovalo, používáme třídu `labeled_Mat` ze souboru `tracer_helper.cpp`. Té na začátku funkcí `init` přiřadíme obrázek s kostrou.

Na čtení pixelů pak používáme funkce `safeat` a `apxat`. Obě funkce mohou dostat libovolné souřadnice a vrací hodnotu odpovídajícího pixelu. Pokud čteme z oblasti mimo obrázek, dostaneme nuly. Dále také přijímají parametr `unlabeled`, pokud je nastaven na `true`, jsou u označených pixelů místo jejich původních hodnot vráceny nuly. Funkce se liší v tom, že `safeat` pracuje výhradně s celočíselnými souřadnicemi, ale `apxat` přijímá i neceločíselné. Neceločíselné souřadnice jsou pak počítány z okolních celočíselných bilineární interpolací.

Pixely lze označovat funkcemi (`label_near_pixels` a `label_pix`) a případně podle potřeby můžeme značky zase zahazovat (`drop_smaller_or_equal_labels`, `drop_smaller_labels_equal_or_higher_make_permanent`). Druhá funkce přenastaví všechny značky větší nebo rovné zadané hodnotě na 255. Touto hodnotou značíme ty pixely, které jsme se už při trasování rozhodli definitivně použít. Interně si za účelem značení třída udržuje obdélník, ve kterém se nacházejí všechny označené pixely. Při zahazování značek pak používáme funkce OpenCV určené k prahování.

Poslední důležitou vlastností je, že třída umožňuje efektivně získat pozici neoznačeného pixelu s nejvyšší hodnotou (`get_max_unlabeled`).

Trasování (`tracer.cpp`)

Na trasování není mnoho zvláštností. K provedení jednoho kroku se používá funkce `do_prediction`. Ta vyzkouší podle povoleného počtu zanoření několik variant a vrátí tu nejlepší z nich. Funkce je volaná z `trace_part`, která je zodpovědná za natrasování jedné cesty, otočení, smzání prvního bodu a dotrasování zbytku. První bod značíme v `labeled_Mat` číslem 254.

Vyhlazování a zjednodušování (`approximation.cpp`)

Zde je nejdůležitější funkce `approximate_with_one_segment`, která zadaný `std::list` segmentů cesty zkusí nahradit jednou Bézierovou křivkou.

Pro použití stejného kódu při hledání obvodové reprezentace se používá funkce `optimize_control_point_lengths`. Ta dostane kontrolní body Bézierovy křivky a seznam bodů, které má touto křivkou aproximovat. Pokud se jí to povede, upraví podle toho vzdálenosti kontrolních bodů.

Metoda nejmenších čtverců (`least_squares_{simple,opencv}.cpp`)

V programu je možné použít jednak vlastní implementaci metody nejmenších čtverců a jednak implementaci využívající knihovnu OpenCV. Protože metodu nejmenších čtverců používáme jen pro dvě neznáme, na rozdíly většinou nenarážíme. Vlastní implementace však není tolik odladěná jako knihovní a může zde častěji docházet k zaokrouhlovacím chybám.

Vykreslování Bézierových křivek ({opencv_,}render.cpp)

Volitelně je možné natrasovaný obrázek opět vykreslit do rastrového. Verze nevyužívající OpenCV používá fixní šířku čáry a je také poněkud nepřesná. Druhá varianta nejprve každý segment rozseká na dostatečně krátké části, které pak vykreslí jako úsečky.

Export (exporter{,_ps,_svg}.cpp)

V současnosti je export možný do dvou formátů. Každý je implementovaný zvlášť, přitom však používají jednotné rozhraní `exporter`. To předpokládá, že se výstup skládá z hlavičky, následně jednotlivých cest a na závěr patičky. Konkrétní třída tedy nemusí procházet přes všechny cesty sama. Toto procházení je však možné v případě potřeby nového exportéru možno nevyužít a celý export provést třeba už ve funkci vypisující hlavičku.

Přestože je SVG formát založený na XML, nevyužíváme k jeho ukládání žádnou speciální knihovnu a konstruujeme si celý výstup sami. Knihovnou pro práci s XML bychom si totiž ani příliš nepomohli, protože nejvíce práce máme s konstrukcí řetězce reprezentujícího jednu cestu a nepotřebujeme načítat žádný vektorový vstup.

7. Rozšíření algoritmu

V následující kapitole rozebereme několik možných doporučených rozšíření, která však v práci nebyla implementována.

7.1 Barevné obrázky

Ačkoli byl vektorizační algoritmus navržený na nebarevné obrázky, dokáže si do určité míry poradit i s barevnými. Problém však nastává již při prahování a hledání morfologické kostry. Morfologickou kostru lze hledat jen pokud o každém pixelu umíme určit binární informaci, zda je součástí objektu, či nikoli. To je nejsnazší s binárním obrázkem.

Náš program v této části předpokládá, že obrázek je skutečně černobílý a vektorizované čáry jsou vůči pozadí dostatečně kontrastní i po převedení obrázku do odstínů šedi, jinak o ně přijdeme již při prahování.

Na barvy se dostává až při trasování. Každému kontrolnímu bodu je přiřazena barva odpovídající barvě pixelů původního obrázku, které jsou na stejném místě jako odpovídající část morfologické kostry. Tento způsob získávání barvy má výhodu, že barvu vzorkuje v místech, která nebývají příliš rušena okolím – středy objektů vs. okraje objektů.

Další problém je, že výstupní formáty nepodporují přímo proměnlivou barvu (obdobně jako u šířky). Proto nedokážeme dokonale podchytit jakékoli změny barvy uvnitř objektu a barva celé cesty je tedy v současnosti před exportem průměrována. Pokud jsou barevné čáry odděleny, dokážeme obrázek vektorizovat věrně.

Alternativní přístup, který však nebyl implementován kvůli netriviálnosti prvního kroku, by prahování nahradil segmentací na omezený počet barev. Hledání kostry by potom bylo možné zobecnit tím, že eroze by nahrazovala všechny pixely sousedící s pixely s odlišnou barvou za černé a dilatace by barvu rozšiřovala pouze na dříve černé pixely. Při nekvalitním osvětlení je však občas i u jinak černobílých obrázků problém se správným prahováním.

Případně je možné inspirovat se tím, jak funguje barevné trasování pomocí Potrace v programu Inkscape. Barevný obrázek je nejprve rozdělen na b binárních. Následně je každý z nich vektorizován samostatně. Výsledek pak vznikne složením jednotlivých rozdílně obarvených vrstev přes sebe. Stačilo by tedy vždy vybrat jednu konkrétní barvu a všechny ostatní ztotožnit s pozadím. Stále však potřebujeme spolehlivě vyřešit problém segmentace na barevné plochy.

Nelze obecně určit, který z přístupů je lepší. V případě, že je ve vstupním obrázku malý počet dostatečně odlišitelných barev, vyplatí se pro každou z nich počítat kostru zvlášť. Pokud však barvy přecházejí plynuleji, vznikly by nám rozdělením falešné hranice mezi kostrami.

Nicméně problém barevných přechodů je opět samostatná úloha, která se týká především barevných ploch, a tedy u čárové grafiky nemá takové využití.

7.2 Nečárová grafika

Pokud obrázek obsahuje i nečárové objekty, jsou stále převáděny na svou morfologickou kostru a následně reprezentovány pomocí širokých čar. Ty navíc velmi často mění svoji šířku, protože u krajů objektu nelze vepsat větší kružnice se středem v kostře. Příkladem nečárového objektu je obyčejný vyplněný čtverec. Jeho kostru tvoří dvě úhlopříčky s průběžně se měnící šířkou stopy. Na jejich krajích je tato šířka nulová, rovnoměrně se rozšiřuje až do prostřed, kde odpovídá velikosti hrany čtverce. Následně se opět symetricky zužuje.

Výsledný obraz tak sice bude v mnohém odpovídat našim původním požadavkům (věrná podoba vektorové verze s originálem, středová reprezentace čar, dostatečně malý výstup), ale velmi se bude lišit od reprezentace, kterou bychom očekávali na první pohled či vytvořili ručně. Nutno podoktnout, že kostra čtverce je složena z úhlopříček pouze za předpokladu, že používáme skeletonizaci vkládající diamanty, osmiúhelníky, nebo kružnice. Zhangův-Suenův algoritmus i vkládání čtverců označí za kostru pouze jeden prostřední pixel.

Vhodným rozšířením by tedy bylo naimplementovat také trasování ploch a automaticky mezi oběma způsoby přepínat dle potřeb objektů. Chystané vylepšení může využívat knihovnu Potrace.

7.3 Uživatelské rozhraní

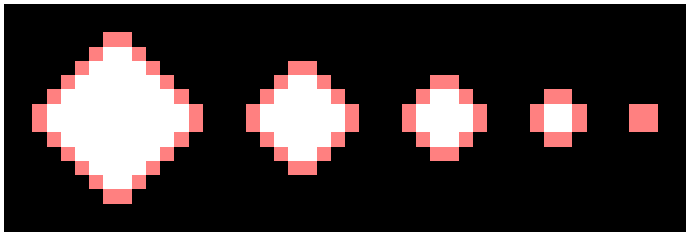
Chování jednotlivých kroků vektorizátoru lze ovlivňovat pomocí parametrů. Ty nejdůležitější lze ladit z grafického rozhraní, avšak ne všechny. Toto je dáno omezením použité knihovny OpenCV, která na uživatelské rozhraní není zaměřená. Bylo proto zvoleno jednodušší uživatelské rozhraní, díky kterému ale program není závislý na dalších knihovnách. Na lepší návrh uživatelského rozhraní je vhodné nejprve nasbírat zkušenosti s používáním čárového vektorizátoru.

7.4 Plugin do Inkscape

Pokud se program Vectorix v praxi osvědčí, bylo by praktické, aby jej mohlo snadno používat více lidí. Protože cílem bylo vytvořit open-source vektorizační nástroj, chtěli bychom o něj rozšířit funkcionalitu open-source vektorového editoru Inksapce. Inkscape již používá vektorizaci pomoci knihovny Potrace, ale čárovou dosud nenabízí.

7.5 Vylepšení skeletonizace

Většina v současnosti používaných metod skeletonizace je poměrně rychlá. Pouze Zhangův-Suenův algoritmus produkuje souvislou a jeden pixel širokou kostru. Pokud kostra není široká jeden pixel, klademe tím větší nároky na trasování, které si musí umět z více pixelů správně vybrat a všechny je označit jako použité. V opačném případě by totiž hrozilo, že se dva pixely široký úsek natrasuje dvakrát. Pokud kostra není souvislá, může se snadno stát, že si ji trasování nespojí v jednu a čára je potom ve vektorovém výstupu zbytečně rozdělena na dvě (či více).



Obrázek 7.1: Nedostatek Zhangova-Suenova algoritmu: na tomto obrázku algoritmus najde pouze prázdnou kostru, červené pixely algoritmus označil ke smazání v dalším průchodu

Zhangův-Suenův algoritmus však má jinou nevýhodu. U objektů ve tvaru diamantu se sudým rozměrem žádnou kostru nenajde (viz obrázek 7.1). V každé iteraci totiž odebere všechny pixely, které sousedí s některým černým hranou. Libovolně velký objekt tak může při vektorizaci zcela zmizet. V praxi tato situace nenastává, protože nemáme takto pravidelné obrázky. Při změně libovolného pixelu již dostáváme neprázdnou kostru.

7.6 Grafová interpretace kostry

Pokud by kostra byla kvalitnější, bylo by možné na ni nahlížet jako na kombinatorický graf. Jednotlivé pixely by byly vrcholy a hrana by vedla mezi těmi sousedními. Tím bychom mohli trasování zjednodušit na procházení grafu.

Pokud by to podporoval výstupní formát, mohli bychom do něj přidat informace o křižovatkách čar. Tyto informace by pak mohl využívat vektorový editor a mohl tak umožnit pohybování s celou křižovatkou naráz.

Závěr

Jako součást práce vznikl program Vectorix, ve kterém jsou v práci popsané metody implementovány a demonstruje tím jejich použitelnost. Prokazujeme tím, že současný nedostatek volně dostupných čárových vektorizátorů není způsoben neřešitelností úlohy.

Kvalita výstupu našeho programu je srovnatelná s běžně dostupnými vektorizátory. Náš vektorizátor si poradí i s trasováním čárového obrázku, který byl pořízen nekvalitně (např. fotoaparátem).

Protože naším cílem byla vektorizace čárové grafiky, není překvapivé, že program nezvládá tak dobře části s většími vyplněnými objekty. Jelikož předpokládáme, že obrázek bude nadále upravován ručně v počítači, nepovažujeme za zásadní nedostatek, že některé detaily obrázku bude nutné po vektorizaci opravit.

Seznam použité literatury

Adobe Systems Inc. (1999). *PostScript Language Reference (3rd Edition)*. Addison-Wesley. ISBN 0-201-37922-8. URL http://www.adobe.com/products/postscript/pdfs/PLRM.pdf.

Birtles, B. (2014). SVG Proposals: Variable width stroke. URL https://www.w3.org/Graphics/SVG/WG/wiki/Proposals/Variable_width_stroke.

Bitter, I., Sato, M., Bender, M., McDonnell, K. T., Kaufman, A. a Wan, M. (2000). Ceasar: A smooth, accurate and robust centerline extraction algorithm. pages 45–52, Salt Lake City, 2000.

Cedar Lake Ventures Inc. Vector magic. URL http://www.vectormagic.com/. Software.

CompuServe Inc. (1990). *Graphics Interchange Format, Programming Reference*. URL https://www.w3.org/Graphics/GIF/spec-gif89a.txt.

Dahlström, E. a kol. (2011). *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. World Wide Web Consortium. URL https://www.w3.org/TR/SVG/.

Diebel, J. R. (2008). *Bayesian Image Vectorization: The Probabilistic Inversion of Vector Image Rasterization*. Disertační práce, Stanford, CA, USA. AAI3332816.

Fournier, A., Fussell, D. a Carpenter, L. (1982). Computer rendering of stochastic models. *Commun. ACM*, **25**(6), 371–384. ISSN 0001-0782. doi: 10.1145/358523.358553. URL http://doi.acm.org/10.1145/358523.358553.

Henderson, B. (2013). *Portable Any Map*. San Jose. URL http://netpbm.sourceforge.net/doc/pnm.html.

Hoschek, J. (1987). Approximate conversion of spline curves. *Computer Aided Geometric Design*, **4**(1-2), 59–66. ISSN 0167-8396. doi: 10.1016/0167-8396(87)90024-0. URL http://www.sciencedirect.com/science/article/pii/0167839687900240.

Hoschek, J. (1988). Spline approximation of offset curves. *Computer Aided Geometric Design*, **5**(1), 33–40. ISSN 0167-8396. doi: 10.1016/0167-8396(88)90018-0. URL http://www.sciencedirect.com/science/article/pii/0167839688900180.

Inkscape. URL https://inkscape.org/. Software.

Lána, J. (2001). *Digitalizace mapy*. Diplomová práce, Univerzita Karlova v Praze, Matematicko-fyzikální fakulta.

OpenCV (2015). *The OpenCV Reference Manual*. Itseez. URL http://opencv.org/.

Otsu, N. (1979). A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man and Cybernetics*, **9**(1), 62–66.

Pennebaker, W. B. a Mitchell, J. L. (1993). *JPEG: Still Image Data Compression Standard*. Springer-Verlag, US. ISBN 0-442-01272-4.

Piegl, L. a Tiller, W. (1997). *The NURBS Book*. Second Edition. Springer-Verlag, Berlin. ISBN 3-540-61545-3.

RasterVect. URL `http://www.rastervect.com/`. Software.

Selinger, P. (2003). Potrace: a polygon-based tracing algorithm. URL `http://potrace.sourceforge.net/potrace.pdf`.

Selinger, P. (2015). Potrace. URL `http://potrace.sourceforge.net/`. Software.

Stallman, R. M. a kol. (2014). *Using the GNU Compiler Collection*. Boston. URL `https://gcc.gnu.org/`.

VectorizeNow. URL `http://www.vectorizenow.com/`. On-line služba.

W3C (2013). *Portable Network Graphics (PNG) Specification (Second Edition)*. World Wide Web Consortium. URL `https://www.w3.org/TR/PNG/`.

Weber, M. (2004). Autotrace. URL `http://autotrace.sourceforge.net/`. Software.

Zhang, T. Y. a Suen, C. Y. (1984). A fast parallel algorithm for thinning digital patterns. *Commun. ACM*, **27**(3), 236–239. ISSN 0001-0782. doi: 10.1145/357994.358023. URL `http://doi.acm.org/10.1145/357994.358023`.

Seznam obrázků

Přílohy

Příloha 1 – Přehled parametrů

Základní nastavení

- `vectorization_method`: Použitá vektorizační metoda, `0`: navržený algoritmus, `1`: Potrace, `2`: ladící vektorový obrázek.
- `parameters_append`: Pokud je nenulový, parametry se při ukládání přidávají na konec souboru.
- `file_parameters`: Název souboru, do kterého se budou ukládat parametry.
- `file_pnm_input`: Specifikuje název vstupního PNM souboru.
- `file_input`: Vstupní soubor v libovolném formátu podporovaném OpenCV (lze použít pouze pro `vectorization_method 0`).
- `show_rendered_window`: Pokud je parametr nenulový, zobrazí se okno s vyrenderovaným vektorovým výstupem.
- `output_engine`: Formát vektorového výstupu, `0`: SVG, `1`: PostScript.
- `file_vector_output`: Název výstupního souboru s vektorovým obrázkem.
- `file_pnm_output`: PNM soubor, do kterého bude výstupní vektorový obrázek vyrenderován (pomalou a nepřesnou metodou).
- `file_opencv_output`: Libovolný rastrový soubor, do kterého bude výstupní vektorový obrázek vyrenderován (rychlejší a přesnější metoda).

Grafické rozhraní

- `interactive`: Pokud je parametr nenulový, bude zapnut grafický interaktivní režim.
- `max_window_size`: Maximální rozměry obrázku v okně, větší obrázky budou oříznuty/zmenšeny.
- `zoom_level`: Úroveň přiblížení obrázků v okně, `0`: zmenšené obrázky, `100`: skutečná velikost, obrázky jsou před zobrazením oříznuty dle limitů velikosti okna.

Prahování

- `invert_colors`: Invertování barev na vstupu (shodné s posuvníkem *Invert input*).
- `threshold_type`: Typ prahování (posuvník *Threshold type*), `0`: Otsova metoda, `1`: binární práh se zadanou hodnotou, `2`: adaptivní práh s váhami dle normálního rozdělení, `3`: adaptivní práh s průměrem okolí.

- `threshold`: (Posuvník *Threshold*) hodnota prahu. Pro adaptivní prahování určuje posun prahu o `threshold - 128`.
- `adaptive_threshold_size`: Velikost okolí pro adaptivní prahování (posuvník *Adaptive threshold*).
- `file_threshold_output`: Soubor pro uložení mezivýsledku po prahování.
- `fill_holes`: Míra zaplnění děr (posuvník *Filled*).
- `dust_size`: Míra odstranění bílých teček (posuvník *Filling size*).
- `file_filled_output`: Soubor pro uložení mezivýsledku po filtrování odprahovaného obrázku.

Skeletonizace

- `skeletonization_type`: Parametr udává typ skeletonizace (stejně jako posuvník *Skeletonization*), `0`: diamant-čtverec, `1`: čtverec, `2`: diamant, `3`: vkládání kružnic, `4`: Zhangův-Suenův algoritmus.
- `files_steps_output`: Soubory, do kterých budou uloženy jednotlivé kroky skeletonizace. Znak `#` bude nahrazen pořadovým číslem iterace.
- `file_skeleton`: Soubor pro uložení kostry.
- `file_distance`: Soubor pro uložení vzdálenostní mapy (hodnota pixelu přímo odpovídá vzdálenosti od okraje objektu).
- `file_skeleton_norm`: Soubor s kostrou normalizovanou pro zobrazení.
- `file_distance_norm`: Soubor se vzdálenostní mapou normalizovanou pro zobrazení.

Trasování

- `max_dfs_depth`: Počet trasovacích kroků, které program zkouší před vybráním finální varianty.
- `depth_auto_choose`: Maximální počet nepovedených trasovacích kroků, při kterých je varianta vybrána bez zkoušení dalších možností.

- `distance_coef`: Parametr $2\sigma^2$ definující velikost okolí při hledání středu čar.
- `nearby_limit_gauss`: Vzdálenost v pixelech, ve které se započítávají pixely při hledání středu čar.
- `gauss_precision`: Přesnost v pixelech, se kterou se hledá bod určující střed čáry.

- `angle_steps`: Počet směrů, ve kterých se hledá další pokračování segmentu z rohového bodu.
- `param_min_nearby_straight`: Body ve vzdálenosti do této konstanty se nezapočítávají do ohodnocení rovného úseku.

- `nearby_limit`: Výchozí vzdálenost v pixelech, ve které je hledán následující trasovací bod.
- `angular_precision`: Přesnost se kterou se dourčí úhel k následujícímu segmentu.
- `size_nearby_smooth`: Výchozí vzdálenost, ve které se započítávají body při hledání hladkých úseků. Je vhodné, aby tento parametr odpovídal přibližně třetině parametru `nearby_limit`.
- `max_angle_search_smooth`: Úhel v radiánech určující maximální odchylku, ve které se hledá hladce navazující čára.
- `nearby_control_smooth`: Výchozí vzdálenost, ve které je hledán první kontrolní bod hladce navazující čáry. Vhodná nastavení se pohybují okolo třetiny až poloviny hodnoty parametru `nearby_limit`.
- `smoothness`: Maximální úhel v radiánech, o který může křivka během jednoho segmentu změnit směr, aniž by byla označena za roh.

Závěrečné vektorové úpravy

- `false_colors`: Obarvení jednotlivých čar v obrázku posloupností barev pro snazší rozpoznání nespojitostí čar. Parametr udává rozdíl tónu barev ve stupních mezi následujícími barvami (dle barevného modelu HSV).
- `force_black`: Pokud je nenulový, všechny čáry jsou obarveny na černo.
- `force_width`: Všechny čáry budou exportovány s touto šířkou (pokud je nenulová).
- `force_opacity`: Všechny čáry budou exportovány s touto průhledností (pokud je nenulová).
- `underlay_image`: Cesta k obrázku, který bude vložen na pozadí výstupu, vhodné pro ladění. Tato volba má význam pouze při vykreslování do formátu SVG.
- `debug_lines`: Přidání ladících (červených) čar do vektorového výstupu. V současnosti tyto čáry znázorňují chybové vektory při převodu na obvodovou reprezentaci.

Převody reprezentací a aproximace

- `offset_error`: Maximální povolená chyba aproximace pro obvodovou reprezentaci.
- `offset_iterations`: Maximální počet iterací pro fitování segmentu Bézierovou křivkou.
- `render_max_distance`: Přesnost renderování v pixelech.
- `export_type`: Způsob reprezentace dat ve výstupu, `0`: průměrování šířky, `1`: rozdělení na krátké segmenty, `2`: obvodová reprezentace, `3`: automatická detekce dle rozptylu.
- `auto_contour_variance`: Čáry s rozptylem šířky větším než tento parametr budou převedeny na obvodovou reprezentaci.
- `lsq_method`: Výběr implementace metody nejmenších čtverců, `0`: OpenCV, `1`: vlastní.
- `approximation_error`: Maximální chyba povolená při zjednodušování čar, větší číslo znamená větší míru zjednodušování.
- `approximation_iterations`: Maximální počet iterací algoritmu na aproximaci úseku.
- `approximation_preserve_corners`: Vynucení zachování rohových bodů. Aproximace tyto body nikdy nezahodí.

Textové výpisy

Program produkuje na standardní chybový výstup informační hlášky čtyř kategorií: chyby (`0`), upozornění (`1`), informace (`2`), ladící hlášky (`3`). Následujícími parametry se určuje upovídanost jednotlivých částí programu. Jsou vypisovány všechny hlášky ze stejné a nižší kategorie.

- `pnm_verbosity`: načítání, ukládání a konverze obrázků ve formátu PNM
- `vectorizer_verbosity`: vektorizér
- `approximation_verbosity`: aproximace čar
- `lsq_verbosity`: metoda nejmenších čtverců
- `offset_verbosity`: převod čar na obvodovou reprezentaci