Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vektorizácia rastrového obrázka použitím evolučných algoritmov

bakalárska práca

2012 Tomáš Kuzma

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vektorizácia rastrového obrázka použitím evolučných algoritmov

bakalárska práca

Študijný program: Informatika
Študijný odbor: 2508 Informatika
Školiace pracovisko: Katedra informatiky
Školiteľ: RNDr. Michal Forišek, PhD.

Bratislava, 2012 Tomáš Kuzma

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Tomáš Kuzma
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský

Názov: Vektorizácia rastrového obrázka použitím evolučných algoritmov

Cieľ: Preskúmať existujúce metódy vektorizácie rastrových obrázkov, analyzovať vhodnosť ich použitia v rôznych situáciách. Implementovať a popísať nové metódy, založené na stochastických prístupoch, s dôrazom na paradigmu evolučných algoritmov. Porovnať tieto metódy s existujúcimi prístupmi.

Vedúci: RNDr. Michal Forišek, PhD.
Katedra: FMFI.KI - Katedra informatiky

Dátum zadania: 10.10.2011

Dátum schválenia: 10.10.2011 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Čestne prehlasujem, že túto bakalársku prácu som vypracoval samostatne, s použitím uvedenej literatúry.

V Bratislave, 1. júna 2012 .....

Poďakovanie

Chcel by som sa poďakovať vedúcemu tejto práce, RNDr. Michalovi Forišekovi, PhD., za usmernenie a poskytnutie zdravého nadhľadu, a taktiež RNDr. Tomášovi Kulichovi, PhD. za niekoľko plodných diskusií na témy evolúcie, evolučných algoritmov a tejto práce.

Abstrakt

Problém nájdenia vektorovej aproximácie rastrového obrazu, ktorý obsahuje prevažne plynulé prechody alebo je fotografického pôvodu, je predmetom prebiehajúceho výskumu, ktorý doteraz priniesol hneď niekoľko rôznych efektívnych techník na riešenie tohto problému; v tejto práci predstavíme stochastický prístup k jeho riešeniu inšpirovaný evolučnými algoritmami, založený na metóde zvanej *hill-climbing*. Ďalej poskytneme rozšíriteľnú, modulárnu referenčnú implementáciu založenú na platforme *Java™*, ktorá sa dá použiť na ďalší výskum v tejto oblasti, a kompletné grafické používateľské rozhranie.

Kľúčové slová:

vektorizácia obrazu, stochastický hill-climbing, evolučné algoritmy

Abstract

The problem of finding a vector approximation to a smooth-shaded or a photographic raster image is a target of ongoing research which yielded several different efficient techniques for solving it; in this thesis we introduce a stochastic, hill-climbing based approach to this problem, inspired by evolutionary algorithms. We provide an extensible, modular reference implementation based on the *Java*™ platform which could be used for further research and a complete graphical user interface.

Keywords:

image vectorization, stochastic hill-climbing, evolutionary algorithms

Obsah

Úvod

Evolučné algoritmy predstavujú alternatívny prístup k problémom, ktoré sa nedajú riešiť klasickými spôsobmi. Problémom, ktorých množstvo riešení je príliš veľké na to, aby bolo možné všetky vyskúšať a vyhodnotiť (príkladom je napríklad problém obchodného cestujúceho, ktorého počet možných riešení rastie hyperexponenciálne od počtu miest, a pre veľké inštancie preto nie je prakticky riešiteľný, alebo akýkoľvek problém, kde optimalizujeme spojitú veličinu) a funkcia, ktorá popisuje hodnotu riešenia, je príliš zložitá alebo málo známa na to, aby bolo možné optimálne riešenie odvodiť z nej.

Problém nájdenia vektorovej aproximácie k danému rastrovému obrazu spĺňa oba tieto predpoklady. Prípustným riešením je akákoľvek množina geometrických útvarov, pričom ich parametre – pozície vrcholov, veľkosť, farby, ... – sa dajú chápať ako diskrétne (obmedzené veľkosťou vstupného obrázka v pixeloch, bitovou hĺbkou jeho reprezentácie a podobne) alebo, v idealizovanom prípade, ako spojité. Je teda zrejmé, že počet prípustných riešení nie je obmedzený, a teda nie je možné ich prehľadať všetky. Navyše správanie sa chybovej metriky – podobnosti výstupnej aproximácie k vstupnému obrazu – závisí okrem riešenia aj na charakteristikách konkrétneho vstupného obrazu a teda je ho možné pre praktické účely považovať za príliš zložité pre priame skúmanie.

V prvej kapitole predstavíme klasický prístup k riešeniu tohto problému a uvedieme niekoľko efektívnych algoritmov na jeho riešenie; v druhej kapitole načrtneme alternatívny stochastický prístup založený na princípe simulovania evolúcie a objasníme základy jeho fungovania. Tretiu kapitolu venujeme referenčnej implementácii, východiskám návrhu, použitým technológiám a popisu objektových hierarchií. Zdrojový kód referenčnej implementácie je súčasťou tejto práce – nachádza sa v prílohe B.

Kapitola 1

Klasický prístup

V tejto kapitole si najprv priblížime pojem rastrového obrazu a pojmy s ním úzko súvisiace. Potom si predstavíme vektorový obraz ako protipól k rastrovému obrazu a bližšie špecifikujeme problém vyrobenia vektorovej aproximácie. Následne si predstavíme niekoľko pokročilých metód na riešenie jeho varianty, kde je vstupný obraz tvorený prevažne plynulými prechodmi, alebo fotografického pôvodu[1].

1.1 Rastrový obraz

Farebným modelom nazývame spôsob, akým označujeme farby. Ľudská sietnica má jeden druh receptorov intenzity svetla (tyčinky) a tri [2] druhy farebných receptorov (čapíky) – červený, modrý a zelený. Preto sa pri počítačovej reprezentácii farby najčastejšie používa farebný model RGB (*Red*, *Green*, *Blue*), v ktorom sa udávajú len intenzity týchto troch farebných zložiek svetla. (Existujú rôzne variácie, akým vlnovým dĺžkam a akým intenzitám tieto farebné zložky odpovedajú; nakoľko medzi nimi nie je žiadny štrukturálny ani podstatnejší sémantický rozdiel, nebudeme medzi nimi ďalej rozlišovať). Druhým bežne používaným farebným modelom je CMYK (*Cyan*, *Magenta*, *Yellow*, *blacK*), ktorý sa používa

[1] Vektorizácia obrazov pozostávajúcich prevažne z čiar, ako napríklad kresby či architektonické plány, je lepšie preskúmaná oblasť, v ktorej sa používajú radikálne odlišné prístupy

[2] Pri zriedkavých mutáciách môže počet druhov farebných receptorov klesnúť, alebo mimoriadne zriedkavo *narásť* na štyri. Týmito prípadmi sa v našej práci nebudeme zaoberať.

hlavne pri tlači (z teoretického pohľadu je čierna zložka zbytočná, význam má len pre nedokonalé zmiešavanie tlačových pigmentov).

Rastrový obraz predstavuje diskretizáciu reálneho, spojitého sveta. V jeho najbezpečnejšej podobe (inými sa v tejto práci nebudeme zaoberať), ide o pravidelnú obdĺžnikovú mriežku, ktorej mrežové body sa nazývajú *pixely*, alebo tiež obrazové body. Každému obrazovému bodu je priradená určitá farba, podľa použitého farebného modelu.

1.2 Vektorizácia

Vektorový obraz je akýkoľvek obraz, ktorého obsah pomocou matematických objektov, ktoré sa nazývajú geometrické primitívy. Bežným zástupcom týchto primitív sú body, čiary, krivky a polygóny, ale je možné definovať aj zložitejšie útvary (beziérovské oblasti a difúzne krivky sú príkladmi takýchto komplikovanejších primitív; popíšeme si ich štruktúru a použitie pri vektorizácii v nasledujúcich podkapitolách).

Procesom *vektorizácie* budeme potom nazývať zostavovanie vektorovej reprezentácie daného rastrového obrazu. Tento proces môže prebiehať úplne autonómne (t.j. algoritmicky bez zásahu používateľa), môže prebiehať s asistenciou používateľa, alebo ho môže výtvarník vykonávať ručne. V tejto práci sa budeme zaoberať len plne autonómnou vektorizáciou[3].

1.3 Geometrické siete

Najbežnejším prístupom používaným pri vektorizácii rastrového obrazu je preložiť vstupný obraz určitým druhom geometrickej siete, ktorá sa potom zjednodušuje, deformuje alebo ďalej adaptívne delí, aby vystihla rysy pôvodného obrazu. V tejto podkapitole si predstavíme niekoľko takýchto prístupov.

[3]V publikáciách býva zvykom pri zavedení novej vektorizácie implementovať všetky prístupy, pozri napríklad [7].

1.3.1 Mriežkový gradient

Kolektív autorov v roku 2008 predstavil novú metódu[8] vektorizácie, inšpirovanú už existujúcim vektorovým nástrojom – mriežkovým gradientom – implementovaným v grafických programoch *Adobe Illustator* či *CorelDraw*.

Mriežkový gradient je grafická primitíva, ktorá topologicky tvorí pravidelnú obdĺžnikovú mriežku. Vrcholy mriežky sú rozmiestnené nepravidelne, a každý so sebou nesie informáciu o jeho farbe. Hranice medzi jednotlivými bodmi potom netvoria čiary, ale Beziérovské krivky. Rasterizácia tejto primitívy potom prebieha komplexným interpolovaním parametrov oblasti, v ktorej sa daný obrazový bod nachádza.

Samotný algoritmus popísaný v tomto článku potom pracuje tak, že dostane ako vstup približnú štruktúru mriežkového gradientu, a komplexnou optimalizačnou procedúrou nájde optimálnu deformáciu takto zadanej krivky. (Algoritmus je teda závislý na prvotnom, hoci jednoduchom, vstupe používateľa, ide o asistovanú vektorizáciu.) Príslušné farebné hodnoty a kontrolné parametre sa potom jednoduchým spôsobom vypočítajú z príslušných častí vzorového obrazu.

V článku[6] z roku 2009 kolektív autorov túto techniku rozšíril zovšeobecnením štruktúry mriežky tak, aby mohla vystihnúť ľubovoľnú topológiu, t.j. ľubovoľný počet dier. Týmto spôsobom je možné prirodzene reprezentovať aj objekty, ktoré sú napríklad čiastočne prekryté iným objektom, alebo naozaj takúto topológiu majú.

Druhým, veľmi podstatným vylepšením je možnosť plne autonómneho generovania týchto mriežok. Pre zmenu modelu mriežky musel tento kolektív implementovať aj pozmenenú verziu optimalizácie mriežky. Napriek tomu je tento vylepšený algoritmus zhruba desaťkrát *rýchlejší* ako pôvodný.

Posledným vylepšením je možnosť regulácie kvality výstupnej aproximácie. (Je možné zväčšiť kvalitu na úkor objemnejšej vektorovej reprezentácie a dlhšieho výpočtu, alebo naopak.)

1.3.2 Beziérovské plochy

Metóda prezentovaná v článku[9] z roku 2009 postupuje veľmi podobným spôsobom. Líši sa najmä v dvoch bodoch: vektorová reprezentácia topologicky tvorí trojuholníkovú sieť a namiesto zadávania počiatočnej mriežky používateľom je tá vytvorená analýzou vstupného obrazu pomocou tzv. *obmedzenej Delaunayovej triangulácie*. Podobne ako v predchádzajúcom prístupe, takto získaná trojuholníková mriežka potom prejde optimalizáciou, pričom je počet výrazne zmenšený spájaním trojuholníkov do oblastí, a ich hranice sú reprezentované Beziérovskými krivkami. Príslušné farebné hodnoty a kontrolné parametre sú potom takisto namerané zo vstupného rastrového obrazu.

1.4 Difúzne krivky

Radikálne iný prístup zvolil kolektív autorov z francúzskej výskumnej inštitúcie *Institut National de Recherche en Informatique et Automatique* (INRIA) v spolupráci s *Adobe Systems*. Vo svojom článku[7] predstavujú úplne nový druh vektorového obrazu – *difúzne krivky*.

Myšlienkou za difúznymi krivkami je špecificky do obrazu zaznačiť ostré hranice a ponechať zvyšnú časť obrazu, ktorá už žiadne ostré prechody neobsahuje, na plynulé farebné prechody (gradienty). Zatiaľ čo tento prístup je v oblasti vektorizačných algoritmov veľmi častý, prístup pri difúznych krivkách sa líši v jednej podstatnej vlastnosti – farebné prechody nie sú udané explicitne (technika používaná napríklad mriežkovými gradientami), ale vznikajú implicitne na základe udaných parametrov komplexného procesu – *difúzie*.

1.4.1 Vektorová reprezentácia

Difúzna krivka je špeciálny druh kubického Beziérovského splajnu s niekoľkými ďalšími asociovanými atribútmi. Táto krivka (v lokálnom zmysle) rozdeľuje svoje okolie na dve časti. Každej časti je, nezávisle na sebe, po dĺžke priradený lineárny farebný prechod, za pomoci postupnosti kontrolných bodov (každý

kontrolný bod reprezentuje jednu farbu[4]; farba v každom bode krivky sa získa lineárnou interpoláciou podľa vzdialenosti od najbližších dvoch kontrolných bodov po dĺžke krivky).

Naprieč difúznou krivkou je obraz (v analytickom zmysle) nespojitý. Nakoľko na reálnych obrazoch (napríklad fotografia) sú hrany len zriedkavo absolútne ostré, pridali autori tohoto prístupu difúznym krivkám ešte jeden atribút, a to neostrosť hrany (presnejšie ide o parameter σ Gaussovského rozostrenia). Rovnako ako farebná informácia, aj tento parameter je po dĺžke krivky lineárne interpolovaný.

Vektorizácia je pri tomto prístupe tvorená konečnou množinou difúznych kriviek. Nie sú kladené žiadne požiadavky na ich topológiu, krivky sa môžu navzájom pretínať, nemusia uzatvárať žiadne oblasti a podobne (to je v ostrom protiklade s ľubovoľnou metódou, ktorá používa geometrickú sieť – tá vždy tvorí planárny graf, a konkrétna metóda vektorizácie explicitne udáva jeho oblasti, prípadne vrcholy alebo hrany).

1.4.2 Rasterizácia

Rasterizácia tejto formy vektorovej reprezentácie prebieha na základe fyzikálnej analógie – procesu difúzie. Ako prvý krok sa rasterizujú obrazové body pozdĺž oboch strán difúznych kriviek[5]. V druhom kroku sa rasterizujú aj všetky ostatné obrazové body. Na exaktné vyriešenie difúzie – formulovanej pomocou Poissonovej rovnice – by bolo nutné riešiť rozsiahlu sústavu lineárnych rovníc. Aby bolo možné interaktívne vykresľovanie tejto vektorovej reprezentácie[6], autori tejto metódy implementovali jej približné riešenie pomocou výpočtov na grafickej karte. Posledná fáza spočíva v rozostrení výsledného rastrového obrazu podľa rozostrovacích parametrov určených na difúznych krivkách (špecifické

[4]Autori tejto metódy takisto pracujú s farebným modelom RGB, no myšlienka by sa dala prirodzene rozšíriť aj na ľubovoľný iný model, menovite na obraz v stupňoch šedej.

[5]V skutočnosti by pri presnej rasterizácii mali body na oboch stranách krivky splývať. To by ale neplnilo požiadavky pre ďalšiu fázu algoritmu (stratila by sa časť informácie o správnej farbe), preto sa ako kompromis vykresľuje krivka, ktorá nie je infinitezimálne tenká, ale má konečnú hrúbku, a jej telo tvorí ostrý farebný prechod medzi týmito dvomi ideálnymi "polkrivkami".

[6]Primárny zámer autorov bolo vytvorenie reprezentácie, ktorá by bola intuitívna a ľahko upravovateľná; neasistovaná vektorizácia je až ich sekundárny cieľ.

množstvo rozostrenia pre každý obrazový bod sa vypočíta analogicky ako jeho farebná hodnota).

1.4.3 Vektorizácia

Autori tejto metódy implementovali tiež metódu, ktorou sa k vstupnému rastrovému obrazu nájde aproximácia pomocou difúznych kriviek. V prvom kroku sa pomocou veľmi rozšíreného *Cannyho detektora* nájdu vo vstupnom obraze hrany (tento detektor má nastaviteľnú citlivosť, ktorou sa dá regulovať kvalita resp. zložitosť aproximácie). V ďalšom kroku sa pomocou zmenšených obrazov ku každej takto nájdenej hrane nájde najvýstižnejšia hodnota rozostrenia. Z takto nájdených hrán a hodnôt rozostrenia sa potom vytvoria difúzne krivky, ktorých farebné kontrolné body sa získavajú analýzou okolia krivky v pôvodnom rastrovom obraze.

1.4.4 Vylepšenia procesu difúzie

V roku 2010 kolektív autorov[7] publikoval [2] niekoľko vylepšení predchádzajúcej metódy, ktoré si tu stručne predstavíme:

Difúzne bariéry

V pôvodnej reprezentácii sa často stáva, že difúzna krivka tvoriaca nejakú hranu, nemala vôbec ovplyvňovať jednu svoju stranu. Aby mohol výsledok difúzia na "konštantnej strane" ostať rovnaký, bolo nutné na tejto strane navoliť presné parametre tak, aby odpovedali farebným hodnotám, ktoré by na tomto mieste difúziou vznikli nebyť tejto krivky. Rozšírený model v tejto novšej publikácii umožňuje ľubovoľnú stranu (prípadne obe) difúznej krivky prehlásiť za tzv. difúznu bariéra. Body tejto strany krivky v tom prípade neprispievajú k difúzii vlastnou farbou, ale len zabraňujú, aby sa cez ne difúzia šírila ďalej.

[7] Dvaja jeho členovia pracovali aj na pôvodnom článku.

Nerovnomerná difúzia

Autori pridávajú difúznym krivkám nové nepovinné atribúty, pomocou ktorých je možné presnejšie kontrolovať proces difúzie. Jeden atribút je preferenčný smer, ktorý spôsobí, že difúzia postupuje niektorým smerom rýchlejšie ako inými; autori ako príklad uvádzajú rozmazanú stopu za pohybujúcim sa objektom či plamene. Druhým pridaným atribútom je sila difúzia, pomocou ktorej sa dá nastaviť akási dôležitosť či sila vplyvu difúznej krivky.

Tunelovanie difúzie

Táto technika spočíva v prepojení dvoch rôznych kriviek – zavedie sa bijekcia medzi bodmi týchto dvoch kriviek (manuálne sa vytvoria sa nové kontrolné body, medzi ktorými sa určí bijekcia, ktorá sa následne rozšíri na celú krivku lineárnou interpoláciou). Možným využitím tejto techniky je napríklad plynulé pokračovanie farebného prechodu ako keby v pozadí za iným objektom, či vytváranie tzv. bezšvových textúr (*seamless* textures; ide o rastrové obrazy, z ktorých je možné vytvoriť klonovaní mriežku bez viditeľných nespojitostí (*seams*)).

Kapitola 2

Evolučný prístup

Algoritmy, ktoré sme si predstavili v predchádzajúcej kapitole, analyzujú obsah vzorového rastrového obrazu a na základe informácie, ktorú získajú z rysov rozpoznaných v tomto obraze, konštruujú určitý druh vektorovej reprezentácie. My predstavíme prístup, ktorý funguje opačným spôsobom; formulujeme úlohu nájdenia vektorovej aproximácie k rastrovému obrazu ako optimalizačnú úlohu: cieľom je nájsť vektorovú reprezentáciu, ktorá sa najmenej odlišuje od vzorového rastrového obrazu.

Vektorovú reprezentáciu získame iteratívnym zlepšovaním počiatočnej konfigurácie (v tejto práci začíname s prázdnou vektorizáciou, ale bolo by možné začínať aj s určitou kostrou, získanou napr. rýchlym ale nepresným klasickým algoritmom). Tento prístup sa dá považovať za evolučný algoritmus – je prítomný prirodzený výber (riešenia, ktorá sú horšie, sa neprenesú do ďalšej generácie) aj mutácie (inkrementálne zmeny riešenia). Myšlienka aplikovať evolučné algoritmy na riešenie problému vektorizácie pochádza od Rogera Alsinga[1].

V tejto kapitole si predstavíme niekoľko druhov evolučných algoritmov, následne prezentujeme základné koncepty nášho vektorizačného postupu a formálne definujeme proces vyhodnocovania kvality vektorizácie.

2.1 Evolučné algoritmy

Myšlienka použiť prirodzený výber na riešenie numerických problémov sa prvýkrát objavila v roku 1964 na Technickej univerzite v Berlíne[3]. Svojej metóde dali názov*evolučná stratégia*.

Nezávisle na nich sa v roku 1966 kolektív pracujúci na Kalifornskej univerzite v Los Angeles venoval hľadaniu optimálneho kódu konečných automatov pre rôzne predikčné úlohy. Tento kolektív si pre svoju metódu zvolil názov *evolučné programovanie*.

Tretím zdrojom pôvodu evolučných algoritmov je američan John Holland z Michiganskej univerzity. Ten sa od začiatku šesťdesiatych rokov venuje výskumu v oblasti, ktorú nazval *genetické algoritmy*.

Tieto tri spočiatku nezávislé smery výskumu, už od začiatku silne prepojené, postupnými rozšíreniami a modifikáciami splynuli v jeden celok, ktorý sa nazýva *evolučné algoritmy*. (Niektorí autori medzi týmito smermi rozlišujú, no neexistuje zhoda, aké sú presné kritéria na zaradenie do konkrétnej oblasti. My sa týmto delením nebudeme ďalej zaoberať.) Pre všetky tieto algoritmy sú kľúčové nasledujúce pojmy:

Populácia. Množina jedincov. Každý jedinec reprezentuje jedno riešenie problému. Takmer vo všetkých variantoch evolučných algoritmov je veľkosť populácie podstatný paramater pre simuláciu evolúcie – príliš málo jedincov môže viesť k nenájdeniu dobrých riešení, príliš veľa jedincov zase predstavuje plytvanie výpočtovým výkonom.)

Mutácia. Náhodná zmena, ktorá mierne pozmení jedinca; mutácie pomáhajú udržať istú mieru variability v populácii.

Fitness funkcia. Predstavuje ohodnotenie kvality riešenia; je to základná hybná sila pre prirodzený výber, ktorý tvorí jadro každého evolučného algoritmu.

Kríženie. Objavuje sa iba v niektorých variantoch; ide o spôsob, ktorým z dvoch, prípadne viacerých jedincov vznikne nový jedinec, ktorý bude niesť niektoré znaky každého “rodiča”.

2.2 Použité konvencie

- Znakom $:=$ budeme označovať definíciu, resp. priradenie.
- Znakom $=$ budeme zapisovať rovnosť.
- Na označovanie postupností budeme používať uhlové zátvorky, e.g. $\langle a, b, c \rangle$.
- Zobrazenie f z množiny A do množiny B budeme označovať: $f : A \to B$.
- Zobrazenie prvku $a \in A$ na prvok $b \in B$ budeme zapisovať: $f : a \mapsto b$.
- Množinu prirodzených čísel, $\{0, 1, 2, \ldots\}$, budeme označovať $\mathbb{N}$.
- Množinu prirodzených čísel bez nuly, $\{1, 2, 3, \ldots\}$, budeme označovať $\mathbb{N}^+$.
- Množinu reálnych čísel budem označovať $\mathbb{R}$.
- Nezáporné reálne čísla, t.j. interval $\langle 0; +\infty)$, budeme označovať $\mathbb{R}_0^+$.
- Pre dané $n \in \mathbb{N}^+$ budeme označovať $\mathbb{Z}_n$ množinu čísel $\{ k \mid k \in \mathbb{N} \wedge k < n \}$. (Táto množina s operáciou sčítania modulo n tvorí grupu – odtiaľ pochádza toto označenie; táto vlastnosť ale pre nás nebude podstatná.)
- Nech $w, h \in \mathbb{N}^+$ a nech F je pole. Potom budeme $\mathbb{M}_{w \times h}(F)$ označovať maticu s h riadkami a w stĺpcami, ktorej prvky sú z poľa F.
  - Nech $i, j \in \mathbb{N}$, pričom $1 \leq i \leq w$ a $1 \leq j \leq h$, a nech $\mathbf{M} \in \mathbb{M}_{w \times h}(F)$. Potom prvok v i-tom stĺpci a j-tom riadku $\mathbf{M}$ budeme označovať $\mathbf{M}_{i,j}$.

2.3 Simulácia

Na exaktnú formuláciu optimalizačnej úlohy sú nutné dva objekty: množina prípustných konfigurácií, v ktorej hľadáme optimálnu konfiguráciu, a hodnotiaca funkcia, ktorá priradí každej konfigurácii zodpovedajúce ohodnotenie. Náš problém hľadania vektorovej aproximácie má zmysel formulovať ako minimalizačnú úlohu.

Množinu prípustných konfigurácií budeme nazývať priestor riešení (budeme ho označovať symbolom $\mathbb{S}$), a konkrétne konfigurácie budeme nazývať riešenia. Štruktúrou riešení sa po formálnej stránke nebudeme zaoberať (budeme riešenia ďalej pokladať za atomické symboly).

Úlohu hodnotiacej funkcie bude u nás plniť pojem *chybovej metriky*.

Definícia 1. Triedou chybových metrík budeme nazývať:

$$\mathbb{E} := \left\{ \mathbf{E} \;\middle|\; \mathbf{E} : \mathbb{S} \to \mathbb{R}_0^+ \right\}$$

Jednotlivé prvky tejto množiny budeme nazývať chybové metriky.

Úlohou našej simulácie bude nájsť čo najlepšie riešenie, t.j. riešenie s čo najmenšou odchýlkou (hodnotením chybovou metrikou). Požadované riešenie nezískame priamo, ale vždy iteratívnym zlepšovaním predchádzajúceho riešenia (začiatočné riešenie je prázdna vektorizácia).

Definícia 2. Zlepšujúcou sa postupnosťou aproximácií dĺžky n vzhľadom na chybovú metriku $\mathbf{E}$ budeme nazývať postupnosť $\langle \mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_n \rangle$, pre ktorú platí:

$$\forall i, j \in \mathbb{N},\ 1 \leq i, j \leq n : \ (i < j) \implies (\mathbf{E}(\mathbf{S}_i) > \mathbf{E}(\mathbf{S}_j))$$

2.4 Rasterizácia

Aby sme mohli porovnať pôvodný obraz s našou aproximáciou, musíme najprv zaviesť pojem rastrového obrazu. Základným stavebným kameňom pre neho bude *kanálový raster* – reprezentácia jedného kanálu (červeného, zeleného alebo modrého) rastrových obrazových dát. Pre potreby tejto sekcie predpokladáme, že vždy platí: $w, h \in \mathbb{N}^+$.

Definícia 3. Triedou kanálových rastrov veľkosti $w \times h$ nazývame:

$$\mathbb{K}_{w \times h} := \mathbb{M}_{w \times h}(\mathbb{Z}_{256})$$

Definícia 4. Triedou všetkých kanálových rastrov nazývame zjednotenie:

$$\mathbb{K} := \bigcup_{w \in \mathbb{N}^+} \bigcup_{h \in \mathbb{N}^+} \mathbb{K}_{w \times h}$$

Jednotlivé prvky týchto množín potom budeme nazývať *kanálové rastre*. (Knižnica *Java2D*, ktorú v tejto práci používame, vie natívne pracovať s rastrovými

obrazmi len niektorých farebných modelov a bitových hĺbok. Naša implementácia preto používa najrozšírenejší farbený model – 24-bitový RGB. Každý pixel je reprezentovaný tromi farebnými zložkami – červenou, zelenou a modrou – a každá z týchto 8-bitových zložiek má prípustný rozsah $0 \ldots 255 = \mathbb{Z}_{256}$.)

Ďalej definujeme *rastrový obraz* (vo farebnom modeli RGB) ako trojicu kanálových rastrov rovnakej veľkosti:

Definícia 5. Triedou rastrových obrazov veľkosti $w \times h$ nazývame:

$$\mathbb{I}_{w \times h} \coloneqq \mathbb{K}_{w \times h} \times \mathbb{K}_{w \times h} \times \mathbb{K}_{w \times h}$$

Definícia 6. Triedou všetkých rastrových obrazov nazývame:

$$\mathbb{I} \coloneqq \bigcup_{w \in \mathbb{N}^+} \bigcup_{h \in \mathbb{N}^+} \mathbb{I}_{w \times h}$$

Pomocou pojmu rastrového obrazu môžeme definovať *rasterizáciu* riešenia. Nakoľko vektorový obraz nie je obmedzený na konkrétne rozlíšenie (na rozdiel od rastrového obrazu), je možné ho rasterizovať na ľubovoľnú cieľovú veľkosť (odhliadnuc od deformácie, ak si vyberieme rozlíšenie s iným pomerom strán, ako mal pôvodný obraz).

Definícia 7. Rasterizáciou nazývame akékoľvek zobrazenie $\mathbf{R} : \mathbb{S} \times \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{I}$, pre ktoré platí[1]:

$$\left(\mathbf{R} : \langle S, w, h \rangle \mapsto \mathbf{I}\right) \implies \left(\mathbf{I} \in \mathbb{I}_{w \times h}\right)$$

V implementácii používame na rasterizáciu geometrické primitívy knižnice *Java2D*; náš výstupný formát SVG neurčuje, ako presne má rasterizácia prebiehať. Nezistili sme však medzi rôznymi nástrojmi na prezeranie a rasterizáciu súborov SVG iný rozdiel, než prítomnosť alebo absenciu vyhladzovania hrán.

2.5 Rastrová metrika

Pomocou pojmu rasterizácie už teraz môžeme zaviesť pojem *rastrovej chybovej metriky*, ktorá predstavuje spôsob, ako ohodnotiť kvalitu riešenia na zákla-

[1] Teda že rasterizáciou obraz na rozlíšenie $w \times h$ vznikne obraz veľkosť $w \times h$.

de odchýlky rasterizácie našej vektorovej aproximácie od pôvodného rastrového obrazu.

Aby sme oddelili jednotlivé časti tohoto konceptu, najprv si zadefinujeme *rastrovú metriku*, ktorá bude porovnávať dva rastrové obrazy.

Definícia 8. Triedou kompatibilných dvojíc rastrových obrazov nazývame:

$$\mathbb{P} \coloneqq \bigcup_{w \in \mathbb{N}^+} \bigcup_{h \in \mathbb{N}^+} \mathbb{I}_{w \times h} \times \mathbb{I}_{w \times h}$$

Definícia 9. Rastrovou metrikou nazývame zobrazenie $\mathbf{D} : \mathbb{P} \to \mathbb{R}_0^+$, ktoré spĺňa nasledujúce podmienky:

$$(1) \qquad \forall \langle \mathbf{I_1}, \mathbf{I_2} \rangle \in \mathbb{P} : \ \Big(\mathbf{D} : \ \langle \mathbf{I_1}, \mathbf{I_2} \rangle \mapsto 0 \Big) \iff \Big(\mathbf{I_1} = \mathbf{I_2} \Big)$$

$$(2) \qquad \forall \langle \mathbf{I_1}, \mathbf{I_2} \rangle \in \mathbb{P} : \ \Big(\mathbf{D} : \ \langle \mathbf{I_1}, \mathbf{I_2} \rangle \mapsto c \Big) \implies \Big(\mathbf{D} : \ \langle \mathbf{I_2}, \mathbf{I_1} \rangle \mapsto c \Big)$$

Aby boli definície konkrétnych rastrových metrík čitateľnejšie, definujeme si dva pomocné pojmy. Prvým z nich budú *kanálová projekcia* – zobrazenie, ktoré dostane ako argument rastrový obraz a jeho výsledkom je niektorá zložka tohto obrazu – kanálový raster.

Definícia 10. Kanálovými projekciami nazývame zobrazenia k_R, k_G, k_B:

$$\begin{aligned} k_R &: \ \mathbb{I} \to \mathbb{K} \\ k_R &: \ \langle \mathbf{K}_R, \mathbf{K}_G, \mathbf{K}_B \rangle \mapsto \mathbf{K}_R \end{aligned}$$

$$\begin{aligned} k_G &: \ \mathbb{I} \to \mathbb{K} \\ k_G &: \ \langle \mathbf{K}_R, \mathbf{K}_G, \mathbf{K}_B \rangle \mapsto \mathbf{K}_G \end{aligned}$$

$$\begin{aligned} k_B &: \ \mathbb{I} \to \mathbb{K} \\ k_B &: \ \langle \mathbf{K}_R, \mathbf{K}_G, \mathbf{K}_B \rangle \mapsto \mathbf{K}_B \end{aligned}$$

Druhým užitočným pojmom bude *indexová množina*. Obsahuje všetky prípustné kombinácie horizontálnej pozície i, vertikálnej pozície j a kanálu (reprezentovaného ako kanálová projekcia). Účelom indexovej množiny je teda odka-

zovať sa na všetky najmenšie adresovateľné jednotky rastrovej obrazovej informácie – farebné zložky jednotlivých obrazových bodov (prvky $\mathbb{Z}_{256}$).

Definícia 11. Indexovou množinou rastrového obrazu veľkosti $w \times h$ nazývame nasledujúcu množinu:

$$P_{w\times h} \coloneqq \left\{ \langle i,j,k\rangle \;\middle|\; i,j \in \mathbb{N}^{+} \wedge 1 \leq i \leq w \wedge 1 \leq j \leq h \wedge k \in \{k_R, k_G, k_B\} \right\}$$

Pomocou týchto dvoch pomocných pojmov už môžeme zadefinovať konkrétne rastrové metriky. Všetky rastrové metriky, ktoré sme v tejto práci použili, reprezentujú nejaký druh priemeru všetkých odchýlok farebných zložiek jednotlivých obrazových bodov.

Definícia 12. Lineárnou rastrovou metrikou nazývame metriku:

$$\mathbf{D}_1 : \mathbb{P} \to \mathbb{R}_0^+$$
$$\mathbf{D}_1 : \langle \mathbf{A}, \mathbf{B}\rangle \mapsto \left(\frac{1}{|P_{w\times h}|} \sum_{\langle i,j,k\rangle \in P_{w\times h}} \left| \big(k(\mathbf{A})\big)_{i,j} - \big(k(\mathbf{B})\big)_{i,j} \right| \right)$$

Lineárna rastrová metrika odpovedá aritmetickému priemeru odchýlok[2].

Definícia 13. Kvadratickou rastrovou metrikou nazývame metriku:

$$\mathbf{D}_2 : \mathbb{P} \to \mathbb{R}_0^+$$
$$\mathbf{D}_2 : \langle \mathbf{A}, \mathbf{B}\rangle \mapsto \left(\frac{1}{|P_{w\times h}|} \sum_{\langle i,j,k\rangle \in P_{w\times h}} \left| \big(k(\mathbf{A})\big)_{i,j} - \big(k(\mathbf{B})\big)_{i,j} \right|^2 \right)^{\frac{1}{2}}$$

Kvadratická rastrová metrika odpovedá kvadratickému priemeru odchýlok. (Táto metrika je najčastejšie používaná v publikáciách, preto je najužitočnejšia, ak by sme chceli porovnávať výsledky našej metódy s výsledkami iných metód.)

[2] Odchýlkou sa myslí veľkosť rozdielu, teda ide o nezáporné reálne číslo ($\mathbb{R}_0^+$)

Definícia 14. Kubickou rastrovou metrikou nazývame metriku:

$$\mathbf{D}_3 : \mathbb{P} \to \mathbb{R}_0^+$$
$$\mathbf{D}_3 : \langle \mathbf{A}, \mathbf{B} \rangle \mapsto \left(\frac{1}{|P_{w \times h}|} \sum_{\langle i,j,k \rangle \in P_{w \times h}} \left| \left(k(\mathbf{A})\right)_{i,j} - \left(k(\mathbf{B})\right)_{i,j} \right|^3 \right)^{\frac{1}{3}}$$

2.6 Rastrová chybová metrika

Použitím rastrovej metriky, ktorej výsledkom je miera odlišnosti medzi dvomi rastrovými obrazmi, už môžeme definovať sľubovaný pojem rastrovej chybovej metriky. Konceptuálne ide o nasledujúci proces:

1. Máme pôvodný rastrový obraz **O** a nejakú jeho vektorovú aproximáciu
2. Rasterizujeme pomocou rasterizačnej funkcie **R** vektorovú aproximáciu **S**; dostávame rastrový obraz **I**.
3. Vyhodnotíme pomocou rastrovej metriky **D** veľkosť odchýlky medzi vzorovým obrazom **O** a rasterizáciou aproximácie **I**.
4. Získanú odchýlku prehlásime za výsledok chybovej metriky riešenia **S**.

Definícia 15. Nech **O** je vzorový rastrový obraz veľkosť $w \times h$, **R** je rasterizácia a **D** je rastrová metrika, potom definujeme rastrovú *chybovú* metriku $\mathbf{E}_{\mathbf{D},\mathbf{R},\mathbf{O}}$ ako:

$$\mathbf{E}_{\mathbf{D},\mathbf{R},\mathbf{O}} : \mathbb{S} \to \mathbb{R}_0^+$$
$$\mathbf{E}_{\mathbf{D},\mathbf{R},\mathbf{O}} : \mathbf{S} \mapsto \mathbf{D}\Big(\mathbf{R}\big(\mathbf{S}, w, h\big), \mathbf{O}\Big)$$

Kapitola 3

Implementácia

V tejto kapitole si predstavíme základné východiská nášho objektového návrhu a použité technológie. Následne charakterizujeme hierarchie objektov a ich interakcie. Kapitolu uzavrieme popisom grafického používateľského rozhrania.

3.1 Ciele návrhu

Pri návrhu a výbere technických prostriedkov na implementáciu tejto práve sme za zamerali na štyri kľúčové požiadavky:

Právna stránka.
Keďže táto práca neslúži na komerčné účely, toto kritérium nebolo príliš obmedzujúce. Jedinou použitou technológiou, ktorú nie je možné na komerčné účely použiť bez zakúpenia licencie, je knižnica *JTattoo*. Keďže ovplyvňuje iba *vzhľad* grafického užívateľského rozhrania, nie je pre samotný proces vektorizácie nutná.

Výkon.
Evolučné algoritmy sú vo všeobecnosti značne náročné na výpočtová výkon, preto aj nami vybraná platforma musí odrážať tieto požiadavky. Významné postavenie má v tomto prípade pracovať s viacerými procesormi či jadrami.

Prenositeľnosť.

Budeme sa usilovať, aby naša implementácia nebola naviazaná na konkrétny *hardware*, inštrukčnú sadu ani operačný systém. Navyše si kladieme požiadavku, aby jadro našej simulácie nezáviselo na prítomnosti grafického systému na počítači, na ktorom simulácia beží.

Rozšíriteľnosť.

V tejto práci predstavujeme základnú kostru, ktorá má slúžiť na riešenie problému vektorizácie pomocou evolučných algoritmov. Implementujeme základné metódy tohto prístupu a špecializovanejšie algoritmy, chybové metriky či modely reprezentácie vektorového obrazu ponechávame otvorené pre ďalší výskum.

Medzi sekundárne kritériá patrila tiež rozšírenosť technológií, prehľadnosť a v neposlednom rade tiež jednoduchosť použitia (či už z pohľadu vývojára alebo užívateľa). Aké prostriedky sme zvolil (a argumenty pre tieto rozhodnutia) si predstavíme v ďalšej podkapitole.

3.1.1 Organizácia kódu

Jednotkou organizácie zdrojového kódu v jazyku Java je tzv. balíček (`package`). Všetok kód napísaný v tomto jazyku by mal byť zaradený do globálnej hierarchie. Aby nedošlo ku konfliktu, *Java Language Specification* odporúča konvenciu[5] podľa ktorej je možné tvoriť unikátne mená balíčkov.

Môj osobný webový priestor je `http://www.ksp.sk/~tommy/`, balíček pre všetok môj osobný kód preto podľa konvencie dostane názov `sk.ksp.tommy`; táto práca dostala identifikátor `apxvec`, hlavný balíček má teda názov `sk.ksp.tommy.apxvec`.

Kód tejto práce je ďalej rozdelený na (pod)balíčky:

`core`	všetky abstraktné triedy týkajúce sa simulácie
`elements`	geometrické útvary
`elements.actions`	prípustné zmeny na útvaroch
`mutators`	objekty mutácií
`metrics`	chybové metriky

`strategy`	implementácie stratégii
`gui`	grafické používateľské rozhranie
`util`	nezaradené

3.2 Použité technológie

Java

Základným stavebným kameňom našej referenčnej implementácie je platforma *Java*. Mimoriadne dobre spĺňa požiadavku prenositeľnosti, jej použitie nie je právne obmedzené, a je možné ju vďaka dynamickému načítavaniu tried a reflexii rozširovať dokonca aj za behu a bez modifikácie zdrojového výkonu.

Jedinou požiadavkou, kde táto platforma nie je optimálnym riešením, je oblasť výkonu. Programy v jazyku *Java* sa kompilujú na tzv. *bytecode*, ktorý sa následne interpretuje vo *virtuálnom stroji*. Tento proces prirodzene nemôže byť rovnako rýchly, ako vykonávanie natívneho kódu procesorom[1]. V jazyku *Java* je ale možné jednoduchým spôsobom písať programy, ktoré vykonávajú niektoré výpočty paralelne na viacerých procesoroch (jadrách).

VisualVM

Tento nástroj umožňuje zisťovať detailné informácie o behu akejkoľvek aplikácie napísanej v jazyku *Java*. Je obzvlášť užitočný na *profilovanie* – analýzu, ktoré časti kódu vyžadujú najviac pamäte a najviac výpočtového výkonu. (Táto technológia nie je nutná ani k spusteniu, ani ku kompilácii našej aplikácie, no ukázala sa byť pri vývoji užitočnou.)

Java2D

Táto knižnica je súčasťou platformy *Java*, a obsahuje kompletnú implementáciu rastrových obrazov, rastrových operácii na nich a vykresľovanie jednoduchších vektorových primitív. Možnosť urýchľovania týchto operácií pomo-

[1]Väčšina bežne používaných virtuálnych strojov ale kompiluje často vykonávané časti programu do natívneho strojového kódu, teda výkonový rozdiel pravdepodobne nebude príliš veľký.

cou grafickej karty je v súčasnosti experimentálna, a je nutné ju špecifikovať pri spúšťaní akéhokoľvek programu. (Naše obmedzené skúsenosti nepreukazujú, že by k takémuto urýchľovaniu aj pri explicitnom zapnutí akcelerácie dochádzalo; predpokladá sa ale, že v budúcnosti bude mať toto urýchľovanie úplnú podporu a bude štandardne zapnuté.)

PNG

Portable Network Grahpics je formát určený na ukladanie rastrového obrazu, so silnou podporou pre *bezstratovú* kompresiu obrazu. Je mimoriadne rozšírený v kontexte webových stránok a platforma *Java*, ako aj najrozšírenejšie operačné systémy (*Windows*, *Linux*, *OS X*), má plne podporuje bez nutnosti dodatočnej inštalácie alebo konfigurácie.

XML

Extensible Markup Language je formát na ukladanie všeobecných dokumentov, a to spôsobom, ktorý ho umožňuje ľahko čítať a zapisovať ako počítaču, tak aj človeku. Formát *XML* v súčasnosti nadobúda značnú popularitu, v oblasti webových technológií, konfiguračných súborov ale aj všeobecných dokumentov a dokumentovo-orientovaných databáz. V našej práci tento formát používame na ukladanie a načítavanie tzv. simulačných profilov – kompaktnej reprezentácie všetkých parametrov simulácie – a pri ukladaní vektorového obrazu vo formáte *SVG*.

SVG

Scalable Vector Graphics je formát určený na reprezentáciu (najmä) vektorovej grafiky; na reprezentáciu dát sa používa formát *XML*. Bol vybraný spomedzi iných možností (`.cdr` je rozšírený formát používaný programom *CorelDraw*, `.ai` je formát používaný populárnym programom *Adobe Illustrator*, atď.) pre hneď niekoľko výhod: je najrozšírenejší, nie je naviazaný na konkrétny produkt, ide o štandard W3C (*World Wide Web Consortium*) a existuje pre neho na platforme *Java* adekvátna podpora (tú v našom prípade reprezentuje knižnica *Apa-*

che Batik[2]). Vektorový výstup našej simulácie budeme ukladať do súboru práve v tomto formáte.

Swing

Knižnica *Swing* je štandardným nástrojom pre tvorbu grafických používateľských rozhraní (od verzie 1.2; v istom zmysle nahradila staršiu, ale stále podporovanú knižnicu *AWT*, na ktorej základoch knižnica *Swing* stavia). Všetky ovládacie prvky sú reprezentované priamo v kóde[3] (na rozdiel od väčšiny iných systémov na tvorbu používateľských rozhraní, kde sú dáta pre vizuálne atribúty ovládacích prvkov uložené v samostatnom súbore).

MigLayout

Táto knižnica obsahuje rovnomennú triedu `MigLayout`, ktorá ma na starosti rozloženie ovládacích prvkov v okne. (Takáto trieda sa tiež nazýva *layout manager*. Presunutím zodpovednosti za presné rozmiestnenie ovládacích prvkov z dizajnéra na algoritmus sa dajú vytvárať rozhrania, ktoré vyzerajú prirodzene na aj pri zmene veľkosti okna.) Táto konkrétna knižnica umožňuje jednoduché ale estetické rozmiestňovanie prvkov, bez potreby pridávania neviditeľných panelov[4].

JTattoo

Táto knižnica poskytuje množstvo alternatívnych vzhľadov pre ovládacie prvky grafického používateľského rozhrania. Pre našu práca nie je nevyhnutná, použili sme ju, aby naše používateľské rozhranie bolo vizuálne konzistentné, bez ohľadu na operačný systém systému, kde je spustené.

[2] K tejto knižnici *de facto* neexistuje žiadna alternatíva.

[3] Na návrh sme používali grafický nástroj *WindowBuilder*, súčasť platformy *Eclipse*, no niektoré zmeny sme museli realizovať priamo v kóde textovým editorom.

[4] To je bežný problém pri použití štandardných tried, poskytovaných knižnicou *Swing*.

3.3 Základné koncepty

Nakoľko cieľom tejto práve bola aj jednoduchá rozšíriteľnosť, celá referenčná implementácia je navrhnutá objektovým spôsobom s využitím abstrakcie na každom mieste, kde to bolo prirodzene možné. Vždy ak bolo rozumné predpokladať, že budú triedy tvoriť jednoduchú hierarchiu, vybrali sme sa cestou abstraktných tried. Rozhrania (*interfaces*) sme používali len vtedy, ak bolo *nutné*, aby trieda dedila naraz od viacerých nadradených objektov[5].

Z pohľadu implementácie vyzerá "objektový ekosystém" prebiehajúcej simulácie nasledovne:

- Celú simuláciu, vrátane zodpovednosti za jej inicializáciu, beh aj finalizáciu, vlastní inštancia triedy `Strategy`.
- Riešenia problému reprezentujú inštancie triedy `Solution`.
- Kvalitu riešenia hodnotí inštancia triedy `ErrorMetric`.
- Mutácie reprezentujú inštancie trieda `Mutator`, ktoré sa aplikujú na riešenia. Stratégia vyrába mutácie pomocou inštancie trieda `MutatorFactory`.

Pri implementácii práce sme využili niekoľko návrhových vzorov[4]. Vždy keď sme tak vedome spravili, je táto skutočnosť zaznačená v popise relevantnej triedy.

3.4 Riešenie

Abstraktná trieda reprezentujúca riešenie problému je `Solution`; pre náš problém vektorizácie reprezentuje konkrétne riešenie trieda `ElementSet` – množina geometrických útvarov (`Element`).

[5]Jazyk *Java* nepodporuje viacnásobnú dedičnosť a podporuje rozhrania. Pre porovnanie, rozšírený jazyk *C++* podporuje viacnásobnú dedičnosť, no nemá prirodzenú prostriedok na vyjadrenie rozhrania (je ale možné mať abstraktnú triedu, ktorá neimplementuje žiadne metódy).

3.4.1 Trieda `Solution`

Centrálna abstraktná trieda tejto práce, ktorá reprezentuje riešenie. Nekladie si žiadne ďalšie požiadavky na *štruktúru* riešenia (ak by sme sa napríklad rozhodli evolvovať mriežkové gradienty, mohli by sme znovu použiť časti tejto práce). Od implementujúcich tried sa vyžadujú nasledovné metódy:

`Object clone()`
: Vytvorí novú kópiu riešenia. Často potrebujeme vyrobiť viacero nových pozmenených verzií z jedného riešenia, preto existujúce riešenie naklonujeme pomocou tejto metódy a na klonoch vykonáme zmeny.

`void draw(Graphics2D canvas)`
: Vykreslí riešenie (trieda `Graphics2D` reprezentuje plátno v *Java2D*). Výsledkom môže napríklad byť rasterizácia (ktorá sa používa na ohodnotenie riešenia) alebo sa môžu kresliace inštrukcie zaznamenávať (napríklad ako súbor vo formáte `SVG`).

`boolean mutate(Mutator M)`
: Pokúsi sa na toto riešenie aplikovať mutáciu (obsiahnutú v objekte typu `Mutator`; pozri ďalej). V prípade, že sa to nepodarí (a teda nedôjde k žiadnej zmene), vráti `false`.

3.4.2 Trieda `ElementSet`

Riešenie ako množina geometrických útvarov (trieda `Element`), uložené v zozname typu `List<Element>`. Táto trieda implementuje všetky metódy triedy `Solution`:

`Object clone()`
: Vytvorí novú kópiu riešenia, ktorá v zozname bude obsahovať kópie všetkých útvarov (použije sa na to metóda `clone()` triedy `Element`).

`void draw(Graphics2D canvas)`
: Vykreslí jednofarebné pozadie a následne vykreslí všetky útvary (použije sa na to metóda `draw(...)` triedy `Element`).

`boolean mutate(Mutator M)`

Pokúsi sa vykonať mutáciu `M`.

- Ak je to mutácia, ktorá mení geometrický útvar (`ElementMutator`), skúsi túto mutáciu aplikovať na *náhodný* útvar zo zoznamu.
- Ak je to mutácia, ktorá pridáva nový útvar (`AddElement`), trieda `ElementSet` tento útvar pridá na koniec zoznamu.
- Ak je to mutácia, ktorá odoberá útvary (`RemoveElement`), trieda `ElementSet` odoberie *náhodný* útvar zo zoznamu.
- Mutácie, ktorá menia pozadie, sa vykonávajú priamo v tejto triede.

3.5 Ohodnotenie riešenia

3.5.1 Trieda `ErrorMetric`

Aby sme mohli posúdiť kvalitu riešenia, zaviedli sme si pojem chybovej metriky, ktorému abstraktná trieda `ErrorMetric` odpovedá. Od implementujúcich tried sa vyžadujú nasledovné metódy:

`Object clone()`

Vytvorí novú kópiu konkrétnej chybovej metriky.

`double evaluate(Solution S)`

Vyhodnotí, aké dobré je riešenie `S`. Lepšie riešenie získa nižšiu odchýlku, presná reprezentácia získa odchýlku 0 (pozri definíciu chybovej metriky). Cieľom simulácie je minimalizovať túto odchýlku.

`void train(BufferedImage I)`

“Nacvičí sa”, ako by malo dobré riešenie vyzerať; trieda `BufferedImage` reprezentuje v knižnici *Java2D* bitmapu.

3.5.2 Trieda `RasterizationMetric`

Táto (stále abstraktná) trieda zastupuje metriky, ktoré vyhodnocujú kvalitu riešenia tak, že ho najprv vykreslia (rasterizujú), a potom vyhodnotia po pixe-

loch rozdiel medzi výsledkom a (zapamätaným) vzorom z metódy `train(...)`. Samotný výpočet je delegovaný na metódu `double evaluateRaster(Raster original, Raster approximation)`, ktorá je implementovaná až v odvodených triedach:

`class LinearMetric`	Výsledkom je aritmetický priemer odchýlok.
`class QuadraticMetric`	Výsledkom je kvadratický priemer odchýlok. (Toto je štandardná metrika.)
`class CubicMetric`	Výsledkom je kubický priemer odchýlok.

3.6 Geometrické útvary

3.6.1 Trieda `Element`

Abstraktná trieda reprezentujúca akýkoľvek geometrický útvar. Od implementujúcich tried sa vyžadujú nasledovné metódy:

`Object clone()`
: Vytvorí kópiu útvaru (používa sa na klonovanie vektorizácií `ElementSet` a tiež na vytváranie mutátorov `AddVertex`).

`void draw(Graphics2D canvas)`
: Vykreslí útvar; ten by mal obsahovať všetky na to potrebné informácie (tvar, pozíciu, výplň, typ čiar, priesvitnosť…).

`void summon(EntropySource C)`
: Nainicializuje všetky atribúty útvaru na náhodné, pričom hodnoty berie z daného zdroja náhodnosti (ten zodpovedá aj za správne rozsahy týchto hodnôt).

Akcie na útvaroch

Akékoľvek zmeny, ktoré je možné na útvaroch prevádzať, sú v kóde reprezentované ako rozhrania (interfaces). Toto rozhodnutie vychádza zo skutočnosti, že nie každá zmena sa dá vykonať na každom útvare[6]. Preto každý útvar, na ktorom

[6]Napríklad kruhu nemôžeme nikdy odobrať vrchol.

je možné vykonať zmenu X, implementuje rozhranie, ktoré zmene X odpovedá. Pre jednoznačnosť budeme tento druh zmien ďalej volať *akcie*.

V našej práci sa používajú tieto druhy akcií:

`ColorChangeable`	Zmení farbu výplne útvaru o danú zmenu $\langle dr, dg, db \rangle$.
`Translatable`	Posunie celý útvar o daný vektor $\langle dx, dy \rangle$.
`Scalable`	Zväčší alebo zmenší celý útvar[7] daným koeficientom $q \in \mathbb{R}$.
`VertexMoveable`	Posunie náhodný vrchol o daný vektor $\langle dx, dy \rangle$.
`VertexPlaceable`	Presunie náhodný vrchol na zadanú pozíciu $\langle x, y \rangle$.
`VertexAddable`	Pridá vrcholu nový vrchol na danej pozícii $\langle x, y \rangle$.
`VertexRemovable`	Odoberie vrcholu náhodný vrchol.

3.6.2 Trieda `ShapeElement`

Táto trieda zastupuje všetky geometrické útvary, ktoré sú popísateľné pomocou tvaru a výplne. Výplň zastupuje rozhranie `Paint` a tvar zastupuje rozhranie `Shape`. Obe tieto rozhrania sú súčasťou knižnice *Java2D*, ktorá pre ne obsahuje aj množstvo rôznych implementácií.

V prípade výplne sa obmedzíme na triedu `Color`, ktorá predstavuje jednoduchú farbu, a v tejto triede priamo aj implementujeme akciu `ColorChangeable`. V knižnici *Java2D* nájdeme aj dve iné triedy, ktoré rozhranie `Paint` implementujú: trieda `GradientPaint` predstavuje farebný prechod a trieda `TexturePaint` vzor – tá ale v našom prípade nemá využitie, pretože by sme vektorizovali bitmapu bitmapou.

Ak by sme chceli, systém sa dá ľahko rozšíriť aj na používanie farebného prechodu. Problémom je, že zo sebou farebný prechod prináša (oproti farbe) viac parametrov – stupňov voľnosti. Dôsledkom je, že nájdenie ich optimálnych hodnôt (resp. priblíženie sa k nim) by bolo zložitejšie (vyžadovalo by viacej pokusov a zlepšujúcich krokov, a teda aj času).

Knižnica *Java2D* poskytuje množstvo tried na reprezentáciu tvaru – čiary, text, polygóny, elipsy či dokonca aj kvadratické a kubické krivky. My tento výber ponecháme na potomkov triedy `ShapeElement`.

[7] A to tak, že stred útvaru ostane na mieste.

3.6.3 Implementujúce triedy

Kruh – trieda `Circle`

Ako je z názvu jasné, táto trieda zastupuje kruh. Trieda, ktorá v tomto prípade implementuje rozhranie `Shape` je `Ellipse2D` (elipsa; v rámci jednoduchosti sme nevyužili voľnosť poskytovanú knižnicou a obmedzili sme sa iba na kruh).

Trieda `Circle` implementuje nasledovné akcie:

- `ColorChangeable` (dedením implementácie od `ShapeElement`)
- `Translatable`
- `Scalable`

Trojuholník – trieda `Triangle`

Aby sme mohli v obrázku reprezentovať hrany (na čo kruh očakávateľne nie je dobrý), použijeme (ako najjednoduchšiu možnosť) trojuholník. Z implementačného hľadiska je použitá trieda `Polygon`, nakoľko knižnica *Java2D* nepozná trojuholník ako elementárny typ.

Trieda `Triangle` implementuje nasledovné akcie:

- `ColorChangeable` (dedením implementácie od `ShapeElement`)
- `Translatable`
- `Scalable`
- `VertexPlaceable`
- `VertexMoveable`

Polygón – trieda `Polygon`

Posledným druhom útvaru v našej implementácii je polygón. Ten – za cenu väčšieho počtu parametrov – prináša schopnosť "vystihnúť" aj komplikovanejšie objekty (napríklad štvorcové okno). Polygóny vždy začínajú ako trojuholníky a vrcholy sú im pridávané a odoberané mutáciami neskôr.

Trieda `Polygon` implementuje nasledovné akcie:

- `ColorChangeable` (dedením implementácie od `ShapeElement`)

- `Translatable`
- `Scalable`
- `VertexPlaceable`
- `VertexMoveable`
- `VertexAddable`
- `VertexRemovable`

3.7 Mutácie

3.7.1 Zdroje náhodnosti

Trieda `EntropySource`

Predstavuje zdroj náhodnosti. Veľkosť relatívnych zmien (posun o vektor; pozri sekciu o akciách na geometrických útvaroch), je regulovaná parametrom $q \in \mathbb{R}$ (ten nie je dostupný zvonku, na prístup k nemu slúžia príslušné metódy).

Od implementujúcich tried sa požadujú nasledujúce metódy:

`Object clone()`
: Vytvorí kópiu (používa sa pri klonovaní celej simulácie).

`float getQ()`
`void setQ(float q)`
: Zistí alebo nastaví koeficient veľkosti mutácií.

Okrem týchto metód sú deklarované tiež (abstraktné) metódy na vracanie náhodných farieb, pozícií, zmien pozícií atď. Tých je ale veľké množstvo a nebudeme ich tu spomínať.

Trieda `RandomEntropySource`

Táto trieda implementuje abstraktné metódy triedy `EntropySource`, ktoré slúžia na generovanie náhodných hodnôt, pomocou delegovania na inštanciu triedy `java.util.Random`.

Používa na to predovšetkým na jej nasledujúce metódy:

`int nextInt(int n)`
Vráti číslo z množiny $\mathbb{Z}_n$ (každé číslo je rovnako pravdepodobné).

`float nextFloat()`
Vráti číslo s rovnomerným rozdelením $R(0, 1)$.

`double nextGaussian()`
Vráti číslo s normálnym rozdelením $N(0, 1)$.

3.7.2 Mutátory

Trieda `Mutator`

Táto abstraktná trieda predstavuje zaobalenie konceptu mutácie do objektu.

`Object clone()`
Vytvorí kópiu (používa si pri vytváraní nových mutátorov).

`void summon(EntropySource C)`
Náhodne nainicializuje mutáciu, príslušné hodnoty získa zo zdroja `C`.

Hierarchia mutátorov

Abstraktnú triedu `Mutator` ďalej rozširujú nasledujúce triedy:

`BackgroundMutator`
Abstraktná trieda, ktorá predstavuje mutácie, ktoré menia pozadie vektorovej aproximácie. Na túto zmenu slúži (abstraktná) trieda `Paint mutate (Paint P)`, ktorá ako parameter dostane pôvodné pozadie a ako výsledok vráti zmenené pozadie[8].
Príslušné implementujúce triedy sú `RandomBackgroundColor`, ktorá nastaví farbu na náhodnú, a `ChangeBackgroundColor`, ktorá farbu pozadia jemne pozmení.

`AddElement`
Mutátor, ktorý slúži na pridávanie nových útvarov. Nový útvar, ktorý sa má

[8] V našej práci sa používa iba jednofarebné pozadie, no nakoľko všetky triedy pracujú s rozhraním `java.awt.Paint`, reprezentujúcim myšlienku výplne, je možné našu implementáciu ľahko rozšíriť, aby používala napríklad farebné prechody. (To isté platí aj pre výplne útvarov).

pridať, nesie tento mutátor v sebe[9]. Tento útvar – inštancia triedy `Element` – v tomto kontexte odpovedá návrhovému vzoru *Prototype*.

`RemoveElement`
: Mutátor, ktorého účinok pri aplikovaní na riešenie je náhodné zahodenie niektorého útvaru (ak riešenie nejaký obsahuje; inak žiadny).

`ElementMutator`
: Abstraktná trieda zastupujúce mutátory, ktoré menia geometrický útvar. Táto trieda deklaruje abstraktnú metódu `Element mutate(Element E)`, ktorá slúži na prevedenie príslušnej zmeny. Metóda dostane ako parameter náhodný útvar, ktorý bol súčasťou riešenia, a jej výstupom je pozmenený útvar, ak sa mutácia podarila, alebo špeciálna hodnota `null`, ak mutácia úspešná nebola. Trieda `ElementSet` nahradí pôvodný útvar návratovou hodnotou tejto funkcie. Všetky implementované mutátory vrátia *referenciu* na pôvodný objekt, ktorý zmenia; tento mechanizmus by sa ale dal využiť aj na vytváranie mutátorov, ktoré menia typ objektu.

Mutátory útvarov

Sú to triedy, ktoré rozširujú `ElementMutator` a implementujú všetky jej abstraktné metódy. Ich vonkajšie rozhranie tvorí vždy iba zdedená metóda `Element mutate(Element E)`, líšia sa tým, akú akciu na elementoch prevádzajú[10].

V práci sú implementované nasledujúce mutátory útvarov:

`AddVertex`	Implementuje akciu `VertexAddable`. (Ak to útvar podporuje, je mu pridaný nový vrchol.)
`ChangeElementAlpha`	Implementuje akciu `ColorChangeable`. (Mení sa priesvitnosť útvaru.)
`ChangeElementColor`	Implementuje akciu `ColorChangeable`. (Mení sa farba útvaru.)

[9] Aby jeho metóda `summon(...)` mala očakávaný účinok, jej volanie ďalej kaskáduje na metódu `summon(...)` obsiahnutého útvaru.

[10] Všetky mutátory v tejto práci používajú práve jednu akciu; to nie je nevyhnutnosť, nenašli sme však rozumnú kombináciu akcií, ktorá by bola prospešná.

`Grow`	Implementuje akciu `Scalable`. (Útvar sa zväčší.)
`MoveVertex`	Implementuje akciu `VertexMoveable`. (Ak to útvar podporuje, jeden vrchol sa posunie.)
`PlaceVertex`	Implementuje akciu `VertexPlaceable`. (Ak to útvar podporuje, jeden vrchol sa premiestni.)
`RemoveVertex`	Implementuje akciu `VertexRemovable`. (Ak to útvar podporuje a je to práve možné, odstráni sa náhodný vrchol.)
`Shrink`	Implementuje akciu `Scalable`. (Útvar sa zmenší.)
`Translate`	Implementuje akciu `Translatable`. (Útvar sa posunie.)

3.7.3 Vytváranie mutátorov

Trieda `MutatorFactory`

Táto abstraktná trieda slúži na produkovanie mutátorov. Je na implementujúcich triedach rozhodnúť, aké typy mutátorov sa budú vytvárať a ako budú inicializované. Ako to jej názov prezrádza, táto trieda teda odpovedá návrhovému vzoru *Abstract factory*.

`Object clone()`
: Vytvorí kópiu (používa sa pri klonovaní simulácie).

`Mutator generate(EntropySource C)`
: Vytvorí novú objekt mutácie, čerpajúc náhodnosť z daného zdroja `C`.
Táto metóda odpovedá návrhovému vzoru *Factory method*.

Trieda `MutatorCloner`

Táto trieda rozširuje triedu `MutatorFactory`. Vytváranie nových mutátorov prebieha *klonovaním* z existujúcich, ktoré sú pridávané metódou `void addP-`

`rototype(Mutator M, float p)` Parameter `Mutator M` predstavuje *prototyp* mutátoru, parameter `float p` pravdepodobnosť[11] jeho vyprodukovania metódou `generate(...)`. Nebude preto žiadnym prekvapením, že mutátor `M` v tomto kontexte odpovedá návrhovému vzoru *Prototype*.

3.8 Stratégie

3.8.1 Trieda `Strategy`

Kontrolu nad celou simuláciou má abstraktná trieda `Strategy`. Zabezpečuje inicializáciu, vytvorenie pracovných vlákien (worker threads), komunikáciu medzi nimi, *thread-safe* vonkajšie rozhranie (na zisťovanie stavu simulácie) a napokon správnu finalizáciu pri zastavení simulácie.

`Strategy(EntropySource chaos, ErrorMetric metric, MutatorFactory factory)`
: Konštruktor, ktorý nainicializuje objekt stratégie do stavu, v ktorom sa už dá spustiť simulácia. Pri vytváraní inštancie triedy `Strategy` sa konštruktoru predajú už inicializované inštancie tried `EntropySource`, `ErrorMetric` a `MutatorFactory`. V prípade triedy `ErrorMetric` to znamená, že už musí mať zadaný aj cieľový raster (metódou `ErrorMetric.train(...)`); inštancia triedy `MutatorFactory` už musí vedieť, aké mutátory má vyrábať (v prípade implementujúcej triedy `MutatorCloner` už táto musí dostať všetky prototypy (inštancie triedy `Mutator`).

`void start(int threads, Solution S)`
: Spustí novú simuláciu, ktorá bude využívať (najviac) `threads` vlákien. Simulácia začína s počiatočným riešením `S` – typické využitie je začať s prázdnym riešením, t.j. s novou inštanciou *implementujúcej podtriedy* trieda `Solution`. Je ale tiež možné začať už s nejakým existujúcim čiastočným riešením (výsledkom inej simulácie; takisto môžeme napríklad začať s jednoduchou vektorizáciou získanou iným spôsobom).

[11] Relatívnu početnosť, že sa vygeneruje práve mutátor `M`. Skutočnú pravdepodobnosť získame až prenormovaním (predelením) súčtom q všetkých prototypov.

`void stop()`

Zastaví celú simuláciu. Toto volanie je blokujúce – objekt stratégie oznámi všetkým ostatným vláknam, že sa má simulácia zastaviť, a potom čaká, kým tieto vlákna skončia (použije sa metóda `Thread.join()`).

`Solution best()`

Vráti doteraz najlepšie nájdené riešenie. Toto volanie môže byť blokujúce. (Nakoľko simulácia prebieha na viacerých vláknach, môže byť nutné počkať, kým niektorý z nich prestane s riešením manipulovať.)

`double bestError()`

Vráti odchýlku najlepšieho doposiaľ nájdeného riešenia. Toto volanie môže byť z rovnakých dôvodov takisto blokujúce.

3.8.2 Trieda `Hillclimbing`

Táto trieda prestavuje *greedy* prístup, t.j. vždy konverguje k *lokálnemu* optimu. Spôsob, akým to dosahuje, je akceptovanie všetkých priaznivých zmien a ignorovanie nepriaznivých. Z implementačného hľadiska si táto trieda pamätá doteraz najlepšie nájdené riešenie (`Solution`) a práca prebieha samostatne v pracovných vláknach.

Na vytváranie, vykonávanie a testovanie mutácií slúži vnútorná trieda (inner class) `Worker`. Táto trieda rozširuje triedu `java.lang.Thread` (ide o štandardnú triedu jazyka Java), ktorá reprezentuje vlákno. Trieda `Worker` reimplementuje zdedenú metódu `void run()` – vykonáva sa v nej cyklus, ktorý funguje nasledovne:

1. Vytvorí sa kópia najlepšieho riešenia (použije sa metóda `Solution.clone()`).
2. Z objektu typu `MutatorFactory` sa vygeneruje nová mutácia (`Mutator`).
3. Aplikuje sa na túto kópiu.
4. Vyhodnotí sa odchýlka tohto riešenia.
5. Ak je toto riešenie lepšie ako doteraz najlepšie, dôjde k náhrade.
6. Ak simulácia nemá skončiť, pokračuj bodom 1.

Všetky prístupy k premenným reprezentujúcim najlepšie riešenie a jeho od-

chýlku, sú súčasťami navzájom sa vylučujúcich kritických sekcií. (Použitý je jazykový konštrukt `synchronized (variable) { ... }`, kde `variable` je akákoľvek inštancia triedy `java.lang.Object`. Nemusí existovať žiadny súvis medzi `variable` a kódom v kritickej sekcii, ide len o jednoznačné označenie, ktoré určuje, ktoré kritické sekcie sa navzájom vylučujú.)

Jednotlivé vlákna, odpovedajúce inštanciám triedy `Hillclimbing.Worker`, spolu nekomunikujú, jediná ich interakcia spočíva v prístupe ku zdieľaným premenným vonkajšej triedy `Hillclimbing`. Nakoľko tieto prístupy prebiehajú vo vzájomne vylučújucich sa kritických sekciách, simulácia funguje korektne[12] aj v prostredí s viacerými paralelne sa vykonávajúcimi vláknami.

3.9 Grafické rozhranie

Na ovládanie simulácie slúži grafické používateľské rozhranie, vytvorené pomocou technológií *Swing*, *MigLayout* a *JTattoo*. Toto rozhranie bolo navrhnuté, aby bolo dobre použiteľné na obrazovky s rozlíšením 1024×768 obrazových bodov alebo ľubovoľným väčším. (Veríme, že týmto sa aplikácia stáva použiteľná na všetkých počítačoch (okrem mobilných zariadení); špeciálne na bežných rozlíšeniach prenosných počítačov 1366×768 resp. 1280×800 obrazových bodov.) Za účelom dostupnosti širšiemu publiku je toto rozhranie vyhotovené v anglickom jazyku.

Hlavné okno aplikácie je vertikálne na dve časti. V dolnej časti, v poradí zľava doprava sa nachádzajú tieto ovládacie prvky:

Panel Images:

Tlačidlo *Load...* umožňuje používateľovi nahrať vzor pre simuláciu (preferovaný formát je `.png`, ale podporované sú aj `.jpg` a `.bmp`.); tlačidlo *Save...* slúži na uloženie rasterizácie výsledky simulácie do súboru (vo formáte `.png`). Na uloženie vektorovej aproximácie slúži tlačidlo *Export...*, ktoré ukladá súbor vo formáte `.svg`. Každé z týchto tlačidiel po stlačení zobrazí

[12] Môže sa stať, že sa naraz pokúsi "nechať sa akceptovať" viacero úspešných zmien. V tomto prípade je po spracovaní týchto požiadavok, bez ohľadu na poradie, v ktorom vznikli, za najlepšie riešenie vyhlásené riešenie s najmenšou odchýlkou, čo je korektné správanie.

dialóg, pomocou ktorého je možné vybrať vstupný (výstupný) súbor.

Panel Simulation:
Aktívnymi ovládacími prvkami sú tlačidlá *Start*, ktoré spustí simuláciu, a tlačidlo *Stop*, ktoré ju zastaví. Zvyšné komponenty udávajú stav momentálne bežiacej simulácie (hodnotu chybovej metriky, dĺžku *postupnosti aproximácií* a ubehnutý čas).

Panel Performance:
Slúži na nastavenie množstva systémových zdrojov (konkrétne jadier[13]), ktoré bude simulácia používať. Počet dostupných jadier sa zobrazuje v pravom dolnom rohu, zvyšné komponenty slúžia na nastavenie počtu jadier pridelených simulácii.

Vrchná časť hlavného okna aplikácie obsahuje dve prepínateľné záložky:

Záložka Preview:
Obsahuje panel *Original*, na ktorom je vykreslený vzorový obraz, a panel *Approximation*, ktorý zobrazuje vždy zatiaľ najlepšie nájdené riešenie práve prebiehajúcej simulácie.

Záložka Profile:
Obsahuje ovládacie prvky pre každý nastaviteľný parameter simulácie. Tieto nastavenie sa dajú uložiť do (načítať z) súboru `.xml` pomocou tlačidiel *Load...* a *Save...*.

[13] Pre naše účely považujeme za jadrá virtuálne jadrá tak, ako ich vidí operačný systém. Ak počítač disponuje technológiou *Intel HyperThreading*, každému fyzickému jadru prislúchajú dve virtuálne.

Záver

V tejto práci sme predstavili nový prístup k vektorizácii rastrového obrazu, ktorý bol inšpirovaný evolučnými algoritmami a pracoval na báze *stochastického hill-climbingu*. Na posúdenie vhodnosti tohto prístupu pre daný problém sme vytvorili referenčnú implementáciu v jazyku Java. Pomocou tejto implementácie sme náš prístup otestovali na fotografických obrazoch; vyhodnotenie jedného takéhoto testu je obsahom prílohy A.

Pre už známe efektívne algoritmy, ktorým sme venovali kapitolu 1, nepredstavuje náš prístup, v jeho najjednoduchšej, naivnej formulácii, konkurenciu. Náš algoritmus, ako aj iné evolučné algoritmy, je značne náročný na výpočtový výkon, a ani pri poskytnutí rádovo väčšieho množstva prostriedkov (procesorového času) oproti klasickým algoritmom neprináša uspokojivé výsledky.

Do budúcnosti

Myslíme si, že napriek absencii praktického použitia v súčasnej podobe má evolučný prístup svoje využitie (je napríklad prirodzene vysoko paralelizovateľný, čo mu v kombinácii so súčasným trendom nárastu počtu jadier v zariadeniach otvára nové možnosti) a ďalšie skúmanie iných modelov vektorového riešenia, komplikovanejších evolučných algoritmov či adaptívnej kalibrácie niektorých parametrov simulácie by mohlo výrazne zvýšiť efektivitu tohto prístupu.

Pre ďalší výskum v tejto oblasti je možné znovu použiť našu referenčnú implementáciu, ktorá bola navrhnutá so zreteľom na všeobecnosť a rozšíriteľnosť. Zaujímavou technikou by sa tiež mohlo ukázať skombinovanie klasického a evolučného prístupu. Pre zrýchlenie počiatočnej fázy simulácie by mohlo byť prospešné začiatočnú konfiguráciu, ktorá v našej implementácii predstavuje prázdnu

množinu útvarov, získať konvenčnou analýzou vstupného obrazu. Pre zrýchlenie konvergencie k lokálnym optimám by bolo možné niektoré atribúty, menovite farbu útvarov, nastaviť na optimálnu hodnotu deterministickou analýzou danej oblasti vstupného obrazu. K rýchlejšiemu vystihnutiu rysov (*features*) obrazu by bolo možné zaviesť cielené mutácie, ktoré by mali zvýšenú šancu zasahovať do častí obrazu, ktoré sa viac odlišujú od vzorového obrazu.

Chybová metrika v evolučnom prístupe nám umožňuje *bez zmeny zvyšku algoritmu* zavádzať nové požiadavky: môžeme minimalizovať počet útvarov, ich prekryvy, alebo penalizovať útvary mimo preferovanej veľkosti. Niektoré tieto požiadavky môžu pomôcť zvýšiť rýchlosť nachádzanie lepších riešení a v neposlednom rade je možné takéto požiadavky podľa predstáv umelca (Napríklad obraz vektorizovaný prekrývajúcimi sa krúžkami zhruba rovnakej veľkosti silne pripomína spôsob maľby umeleckého smeru impresionizmus.)

Literatúra

[1] Roger Alsing. Genetic programming: Evolution of Mona Lisa. http://rogeralsing.com/2008/12/07/genetic-programming-evolution-of-mona-lisa/.

[2] Hedlena Bezerra, Elmar Eisemann, Doug DeCarlo, and Joëlle Thollot. Diffusion constraints for vector graphics. *ACM Transactions on Graphics*, 27(3), 2008.

[3] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1), 1993.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. Addison-Wesley, third edition, 2005.

[6] Yu-Kun Lai, Shi-Min Hi, and Ralph R. Martih. Automatic and topology-preserving gradient mesh generation for image vectorization. *ACM Transactions on Graphics*, 28(3), 2009.

[7] Alexandrina Orzan, Adrien Bousseau, Holger Winnemöller, Pascal Barla, Joëlle Thollot, and David Salesin. Diffusion curves: A vector representation for smooth-shaded images. *ACM Transactions on Graphics*, 27(3), 2008.

[8] Jian Sun, Lin Liang, Fang Wen, and Heung-Yeung Shum. Image vectorization using optimized gradient meshes. *ACM Transactions on Graphics*, 26(3), 2007.

[9] Tian Xia, Binbin Liao, and Yizhou Yu. Patch-based image vectorization with automatic curvilinear feature alignment. *ACM Transactions on Graphics*, 28(5), 2009.

Príloha A

Ukážky

Pôvodný obrázok.

Aproximácia po 15 minútach.

Aproximácia po 5 minútach.

Aproximácia po 60 minútach.

Intel Core i7-2670QM, využitých všetkých 8 virtuálnych jadier.

Príloha B

Zdrojový kód

Zdrojový kód referenčnej implementácie sa nachádza na priloženom CD.