UNIVERSITATEA DIN BUCUREȘTI
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

LUCRARE DE LICENȚĂ

COORDONATOR ȘTIINȚIFIC
Lect. dr. Stupariu Mihai-Sorin

STUDENT
Roșcăneanu George

BUCUREȘTI
Februarie 2016

Cuprins

1. Introducere

Motivația

Într-o zi căutam un cadou pentru o prietenă prin zona Pieței Unirii, iar unul dintre lucrurile care mi-a atras atenția au fost tricourile cu texte sau desene tipărite pe ele. Trecând prin toate modele pe care le aveau nu am găsit ceva destul de potrivit. Am întrebat vânzătoarea dacă poate să facă tricouri personalizate, ea spunând că pozele trebuie să fie în format vectorial. Eram puțin uimit de ce driver-ul imprimantei nu este în stare sa folosească imagini raster obișnuite.

Din curiozitate am căutat pe internet dacă exista tool-uri sau aplicații care fac acest cu ușurința.

Primul lucru pe care l-am încercat a fost http://vectormagic.com . Acesta oferă rezultate foarte bune , având multe opțiuni și setări , dar timpul de procesare este relativ mare și costul unei singure licențe este de 295 $, iar versiunea trial oferă doar posibilitatea de a vedea cum arată fișierul vectorial ; nu poate fi salvat.

Următorul pe care l-am găsit este RaveGrid. Versiunea care am reușit sa găsesc este non comercială . Are doar trei niveluri de detaliu, timp relativ rapid de procesare, posibilitatea de a salva fișierul în format SVGZ (este nevoie de un editor ca InkScape pentru al transforma în SVG) sau EPS.GZ , dar deoarece este Shareware , de fiecare data când un utilizator încearcă sa salveze fișierul aplicația afișează o fereastră care stă deschisă timp de aproximativ 12 secunde și nu poate fi închisă de către utilizator.

Ultima variantă la care m-am uitat este Scan2CAD care are la dispoziție trial gratuit , Lite pentru 699$, Pro pentru 999$ și Business $1500 . Versiunea cea mai ieftină care conține convertirea la o poză vectorială este Lite.

Văzând toate aceste implementări diferite oferite la prețuri relativ mari , am decis sa creez câteva implementări pentru vectorizarea unei imagini, și să public codul sursă pe GitHub.

Definiții și scurt istoric

Grafica vectorială reprezintă folosirea primitivelor geometrice (puncte, segmente de

dreaptă , curbe , poligoane etc.) pentru a crea imagini digitale. Imaginile vectoriale sunt bazate pe vectori (sau perimetre / forme), ele având puncte de control sau noduri de control. Fiecare dintre aceste puncte au o poziție definită în raport cu axele OX și OY ale spațiului de lucru și determină direcția liniilor. Fiecare dintre aceste linii pot avea asignate proprietăți precum culoarea , forma și grosimea. Iar formele geometrice pot avea interiorul umplut cu o culoare sau nu. Toate aceste proprietăți nu cresc în mod semnificativ mărimea fizică a fișierului pe spațiul de stocare, deoarece informațiile doar descriu cum să fie desenate primitivele geometrice.

În comparație cu imaginile bazate pe pixeli , cele bazate pe primitive pot fi mărite oricât fără pierderea calității.

Termenul de grafică vectorială este de obicei folosit doar pentru obiecte grafice 2D, pentru a evidentia diferența fata de grafica raster pe baza de pixeli.

Unul dintre primele folosiri ale graficii vectoriale a fost la sistemele de apărare aeriene US SAGE[1]. Ele au fost de-asemenea implementate de către Ivan Sutherland pe calculatorul TX-2 de la MIT Lincoln Laboratory pentru a rula aplicația Sketchpad[2] în 1963.

Când vine vorba de tipărirea caracterelor folosind calculatoare, fie pe ecrane, fie pe foaie de hârtie folosind imprimante, fie pe alte suprafețe, se folosesc fișierele de tip font (TrueType, OpenType etc). Acestea descriu caracterele folosind curbe cubice sau pătratice cu puncte de control. Oricum, fonturile pe bază de bitmap încă sunt folosite. Convertirea caracterelor în ceva folosibil constă în trasarea curbelor pentru a forma perimetrul ei, iar apoi umplerea acestei figuri geometrice cu culorile dorite. Această convertire nu este un proces trivial din puncte de vedere computațional, de aceea programatorii preferă, să convertească întregul set de caractere în format bitmap și apoi folosindu-l ca un „sprite sheet”. Este foarte posibil ca programatorul să dorească mai multe „sprite sheet”-uri pentru fiecare font și dimensiune de font de care va avea nevoie în aplicația sa.

Una dintre aplicările mai moderne ale graficii vectoriale sunt spectacolele cu lumină laser, unde două oglinzi, una care se rotește pe axa X și cealaltă pe axa Y, sunt controlate pentru a crea linii curbe sau drepte pe o suprafața oarecare.

Comparație: Grafica vectorială VS Grafica raster

Imaginile raster sunt formate din pixeli. Un pixel este un singur punct sau cel mai mic element care poate fi afișat pe ecranul device-ului (Figura 1.1).

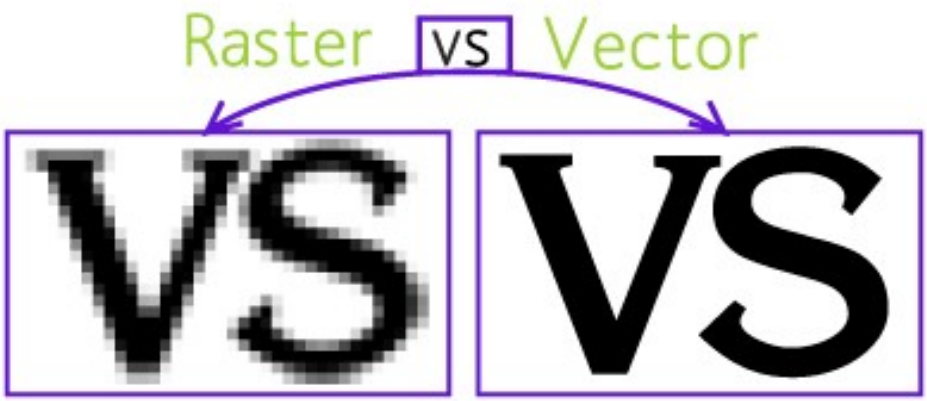


Figura 1.1 : http://vector-conversions.com/vectorizing/raster_vs_vector.html

Se poate observa că atunci când imaginea este mărita suficient de mult , încep sa apară diferențe evidente(Figura 1.2). Când cea raster a fost mărita, începe să piardă din calitate și claritate. Iar cea vectorială rămâne clară oricât de mult se va mări imaginea[3].

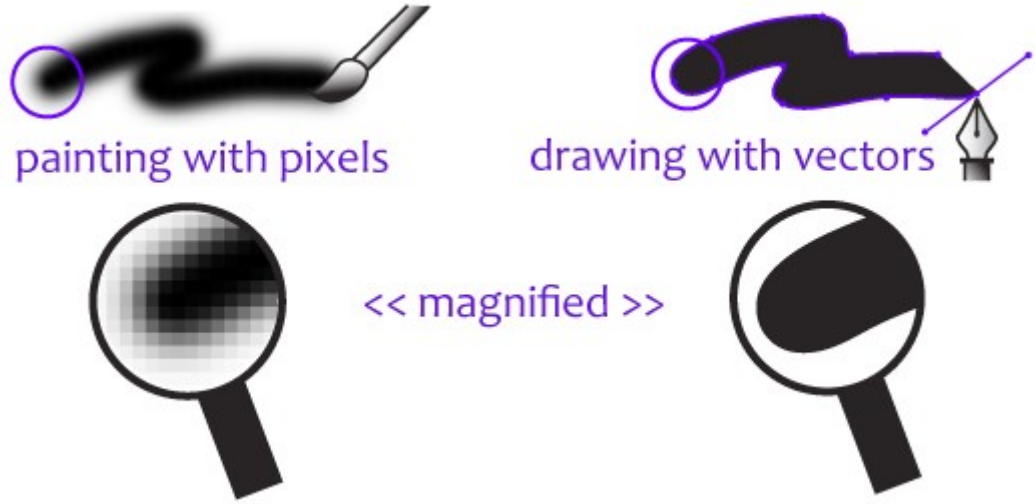


Figura 1.2 : http://vector-conversions.com/vectorizing/raster_vs_vector.html

	Avantaje	Dezavantaje
Vector	* Infinit scalabil : datorită proprietăților matematice ale figurilor geometrice, acestea se pot scala mai mic sau mare fără a pierde din calitate * Fișiere mai mici : pentru imagini conținând grafică simplă, precum figuri geometrice și tipografie	* Detalii limitate : figurile geometrice nu sunt practice pentru imagini complexe, chiar dacă acestea pot conține gradiente, nu se poate compara cu avantajul imaginilor raster de a avea culori independente pentru fiecare pixel * Efecte limitate : prin definiție, imaginile

	* Editabilitatea : în comparație cu formatele jpg și png care stochează toți pixelii pe un singur strat, imaginile vectoriale au figurile geometrice aranjate pe categorii, grupuri și straturi	vectoriale sunt create din linii și puncte, deci nu sunt la dispoziție efecte precum blurare sau umbre
Raster	* Detalii bogate : o poza care are valoarea dpi mai mare va conține mai multe detalii subtile * Editare precisa : oricare dintre pixeli poate fi modificat independent ; o persoană poate să editeze poza respectivă cu ușurință	* Blurată atunci când este mărita : deoarece exista un număr finit de pixeli, calculatorul va încerca sa umple spațiul dintre pixeli cu interpolari ale valorilor de pe canalele de culoare * Fișiere mai mari: cu cât dpi-ul este mai mare cu atât crește proporțional și mărimea fișierului ; deși exista formate precum jpg și png care ajuta la comprimare , ele tot pot ajunge la dimensiuni foarte mari

2. Descrierea facilităților folosite

Despre limbajul Java

Aplicația mea este realizată în Java. Nu am folosit nicio particularitate a acestui limbaj (în afară de faptul că este orientat pe obiecte), totuși, pentru completitudinea lucrării, aceasta include și o secțiune despre el.

Java este un limbaj de programare concurent, bazat pe ierarhia de clase, orientat obiect și în mod special proiectat pentru a avea cât mai puține dependențe de implementare posibile. Autorii lui și-au propus să îi lase pe dezvoltatorii de aplicații să “scrie odată, să ruleze oriunde” (“write once, run everywhere”, WORA). Aceasta înseamnă că un cod care rulează pe o platformă nu mai necesită recompilare pentru a rula pe o alta. Aplicațiile Java sunt în mod obișnuit compilate la cod binar (fișier .class) care poate rula pe orice mașină virtuală Java (Java Virtual Machine - JVM), indiferent de arhitectura sistemului de operare. Java este în mod curent una dintre cele mai populare limbaje de programare folosite, în mod particular pentru aplicațiile client-server, cu un număr 9 milioane de dezvoltatori raportați. Java a fost original dezvoltată de James Gosling în cadrul Sun Microsystems (ulterior cumpărat de Corporația Oracle) și lansată în 1995 sub forma unei componente de bază pentru Platforma Java. Limbajul moștenește mult din sintaxa C și C++, dar are mai puține facilități de nivel jos (“low-level”) față de ambele.

Compilatoarele originale, mașinile virtuale și bibliotecile *class* au fost dezvoltate de Sun din 1991 și lansate pentru prima dată în anul 1995. Din luna mai 2007, în concordanță cu specificațiile Java Community Process, Sun a schimbat licența majorității tehnologiilor sale Java la GNU General Public License.

Alții au dezvoltat de asemenea implementări alternative pentru aceste tehnologii Java, cum ar fi Compilatorul GNU pentru Java, GNU Classpath (biblioteci standard) şi IcedTea-Web (un plugin pentru aplicațiile de browser – applet-uri).

Deși limbajul Java are mai puține facilități low-level , exista JNI (Java Native Interface) , un framework de programare care permite codului scris în Java și care rulează într-o Mașina Virtuala Java să apeleze cod nativ sau să fie apelat de aplicații native și librării scrise în limbaje precum C , C++ sau chiar în limbaj de asamblare.

Avantajele JNI-ul ar fi că anumite părți dintr-un program pot fi înlocuite cu cod nativ pentru a mări viteza de procesare. Din păcate acest avantaj este ușor eclipsat de anumite dezavantaje, de exemplu dificultatea implementării este masiv sporită, codul scris nativ nu are garbage collector , programatorul trebuie să fie atent la diferențele subtile de pe diversele platforme, codul scris în nativ este foarte greu de debug-uit. Luând în considerare aceste dezavantaje am decis ca aplicația să nu conțină cod scris în nativ.

Pentru scrierea codului sursă , am folosit un IDE (IntelliJ IDEA).

Despre IDE-uri

Un mediu integrat de dezvoltare (integrated development environment - IDE) sau mediu interactiv de dezvoltare este o aplicație software care oferă facilități cuprinzătoare programatorilor, pentru dezvoltarea software. În mod normal, un IDE constă dintr-un editor de surse, unelte de construcție a “executabilului” (builder) și un depanator (debugger). Majoritatea IDE-urilor moderne oferă opțiuni de completare automată a codului (Intelligent code completion).

Unele IDE-uri conțin un compilator, un interpretor, sau ambele, cum ar fi cazurile NetBeams , IntelliJ IDEA și Eclipse; altele nu, cum ar fi SharpDevelop şi Lazarus. Limita dintre un mediu integrat de dezvoltare și alte tipuri de aplicații de dezvoltare software nu este bine definită. Câteodată un sistem de versionare a fișierelor și variate unelte care simplifică construcția GUI (graphical user interface = interfață grafică pentru utilizatori) sunt integrate în IDE. Multe IDE-uri conțin de asemenea un navigator pentru clase, un navigator pentru obiecte și o diagramă a ierarhiei claselor, pentru utilizare în dezvoltarea soft-ului orientat pe obiecte.

Despre IntelliJ IDEA

IntelliJ IDEA este un mediu integrat de dezvoltare pentru limbajul Java. A fost creat de compania JetBrains. Prima versiune a fost publicată în ianuarie 2001, și a fost unul dintre primele IDE-uri care oferea navigare avansată a codului și posibilitatea de a refactoriza codul deja integrat.

Într-un reportaj Infoworld din 2010, IntelliJ a primit cel mai mare scor într-un test în care au

participat și IDE-urile Eclipse, NetBeans și Oracle Jdeveloper.

În decembrie 2014, Google a anunțat versiunea 1.0 a programului Android Studio, un IDE open-source folosit la crearea aplicațiilor Android, bazat pe versiunea open-source de comunitate al aplicației IntelliJ IDEA. Alte medii de programare bazate pe IntelliJ sunt AppCode, PhpStorm, PyCharm, RubyMine, WebStorm și MPS.

Acest IDE are doua ediții : Community Edition (gratuit) și Ultimate Edition (plătit). Ambele pot fi folosite în scop comercial.

Ambele ediții au mai multe capabilități comune precum : Smart Code Completion, On-The-Fly Code Analysis, Advanced Refactoring, unelte de Version Control având interfață unificata pentru Git,SVN, Mercurial etc, și desigur suport pentru limbajul Java. Ediția Ultimate oferă mai multe lucruri precum unelte pentru lucru cu baze de date, designer pentru tabele UML, unelte pentru Web Development și altele.

Despre GitHub și Git

Git este un distributed revision control system (sistem distribuit de control al modificărilor) care a fost conceput pentru a avea viteză de procesare mare , păstrarea integrității datelor și suport pentru posibilitatea de avea „workflow”-uri distribuite și non-liniare. Git a fost original creat de Linus Torvalds pentru programarea kernel-ului Linux în 2005, și de atunci a devenit unul dintre cele mai adoptate sisteme de version control.

Ceea ce deosebește Git-ul fața de celelalte sisteme este faptul că fiecare director Git este un repozitor complet și „independent” care conține întreaga istorie a modificărilor, neavând nevoie de acces la rețea sau un server central după ce a fost descărcat sau copiat. Asemenea kernel-ului Linux, Git este un program oferit gratuit sub licența GNU General Public License revizia 2.

Design-ul Git-ului a fost inspirat după BitKeeper și Monotone. Acesta avea ca plan original să fie un sistem low-level pe care alte aplicații îl poate apela. În momentul acesta Git poate fi folosit direct din linia de comanda. Deși a fost puternic influențat de BitKeeper, Torvalds intenționat a evitat tehnicile convenționale, astfel ajungând la un design unic.

Git permite unui developer să verifice ce modificări au fost făcute asupra unui proiect, sau să adauge noi modificări la un repozitor, crearea unei noi „ramuri” de proiect pentru a crea o facilitate nouă și apoi reunirea acestei ramuri la ramura principală pentru ca întregul proiect original sa conțină acestă nouă facilitate. De-asemenea dacă au fost făcute greșeli, se poate reîntoarce codul la un „commit” precedent.

Am decis să îl folosesc pentru a putea readuce proiectul la o stare precedentă în cazul în care am făcut greșeli.

Exista opțiunea de a avea repozitorul stocat doar local pe SSD-ul meu, dar nu vroiam să risc pierderea proiectului așa că am decis sa folosesc GitHub pentru a stoca proiectul.

GitHub este un serviciu online de găzduire a repozitoarelor Git, astfel având toate capabilitățile oferite de către Git. Spre deosebire de Git care oferă strict doar un program care se

folosește prin linii de comanda, GitHub oferă o interfață grafică pentru Web, desktop și telefoane smartphone. De-asemenea oferă administratorului posibilitatea de schimba nivelul de acces al celorlalte colegi care lucrează la proiect, și unelte de colaborare precum bug tracking, feature request, controlul task-urilor și crearea wiki-urilor.

Conturile gratuite de GitHub pot găzdui decât proiecte care sunt complet vizibile oricărei persoane care ori găsește repozitorul ori are link-ul direct către acesta. În anul 2015 , GitHub a raportat faptul că are peste 9 milioane de utilizatori și peste 21.1 milioane de repozitoare, astfel fiind cel mai mare service de găzduire a codurilor sursă de pe Pământ în acel moment.

GitHub este de obicei folosit doar pentru cod sursa, dar el permite în plus vizualizarea fișierelor grafice 3D sau fișiere native PSD de PhotoShop și compararea lor cu celelalte versiuni ale acestora.

Despre Scalable Vector Graphics (SVG)

Scalable Vector Graphics este un format de imagini vectoriale bazat pe XML pentru crearea imaginilor grafice bidimensionale cu suport pentru interactivitate și animații. Specificația SVG este un „open standard” developat de către World Wide Web Consortium (W3C) din 1999.

Deoarece imaginile SVG sunt definite prin fișiere XML, ele pot fi parsate, indexate, scriptate și comprimate efcient. De-asemenea ele pot fi editate chiar și cu un editor text, dar în majoritatea cazurilor se folosesc software-uri specializate.

Toate web browser-ele moderne sunt capabile de a afișa imagini vectoriale SVG.

SVG-ul permite trei tipuri obiecte grafice: grafică vectorială, grafică raster și text. Obiectele grafice, care includ imaginile raster PNG și JPEG, pot fi grupate, stilizate, transformate și combinate cu obiectele care au fost deja randate. El nu suporta în mod direct z-index-ing care separa ordinea desenării de ordinea lor în document pentru suprapunerea obiectelor, spre deosebire de alte standarde precum VML. Asupra obiectelor grafice se pot aplica transformări multiple, clipping paths, masti alpha și efecte de filtru.

Începând cu 2001 , specificația SVG a fost actualizată la versiunea 1.1. SVG Mobile Recommendation a introdus două profile simplificate de SVG 1.1: SVG Basic și SVG Tiny, menit pentru aparatele cu putere computațională redusă. O versiunea îmbunătățită numita SVG Tiny 1.2 a devenit mai târziu recomandarea autonoma.

În momentul acesta se lucrează la SVG 2, care incorporează capabilități noi și include și pe cele folosite în SVG 1.1 și SVG Tiny 1.2.

Deși specificația SVG se concentrează în mod primar pe limbajul de grafica vectorială, design-ul său include și câteva capabilități de baza pentru definirea aranjării obiectelor într-o pagină precum format-ul PDF de la Adobe. SVG-ul are informațiile necesare pentru aranja fiecare glyph și imagine pe o poziția aleasă pe o pagina de printat. Spre deosebire XHTML are ca scop principal să comunice conținutul său, nu prezentarea lui, deci HTML specifică ce obiecte sunt afișate , nu unde sunt aranjate.

Imaginile SVG pot fi dinamice și interactive. Modificările pe bază de timp asupra obiectelor

pot fi descrise cu SMIL (Synchronized Multimedia Integration Language) sau JavaScript. Deși W3C recomanda SMIL, Google Chrome l-a deprecat în August 2015. Evenimente precum *onmouseover* și *onclick* pot fi folosite pentru a produce modificări și animații în SVG.

Deoarece SVG este scris în format XML, sunt foarte multe bucăți de text repetate, deci el poate fi comprimat foarte ușor. Atunci când o imagine SVG a fost comprimată folosind algoritmul *gzip*, acesta este acuma referit ca fiind de tipul SVGZ. De obicei o imagine în format SVGZ ocupă aproximativ 20-50% din spațiul fizic de stocare față de aceeași imagine în format SVG.

Formatul SVG oferă la dispoziție 14 funcționalități pentru definirea lui.

* Path : figuri geometrice definite prin linii drepte sau curbe. Pentru a face mai compactă scrierea coordonatelor, se folosesc litere precum M(move) , L(line to) , Z(închide figura) ,C,S,Q,T,A. Dacă se folosește literă mare, coordonatele sunt absolute, dacă se folosește literă mică, coordonatele sunt relative fată de cele precedente.

* Basic shapes

* Text : suport pentru caractere Unicode. Este posibilă scrierea textului în ambele sensuri orizontale, dar și vertical pentru chineză. Textul poate de-asemenea să fie scris de-a lungul unei căi și să folosească diverse efecte vizuale.

* Painting : figurile pot fi umplute și/sau conturate folosind diverse culori, gradiente sau paterne.

* Color : culorile sunt definite fie prin cuvinte literale precum „black” , „red”, prin hexazecimale #63ff01 , #02f , prin decimale cu rgb(123,23,12) sau prin procente rgb(100%,0%,50%)

* Gradiente și paterne

* Clipping, masking și compositing : conturul de la alte elemente grafice, precume texte, căi și figuri pot fi folosite pentru a definii aceste lucruri

* Efecte de filtru : SVG are la dispoziție mai multe filtre precum : blend,gaussian blur , merge , tile etc

* Interactivitate : orice parte dintr-o imagine SVG poate avea „event listeners”

* Linking

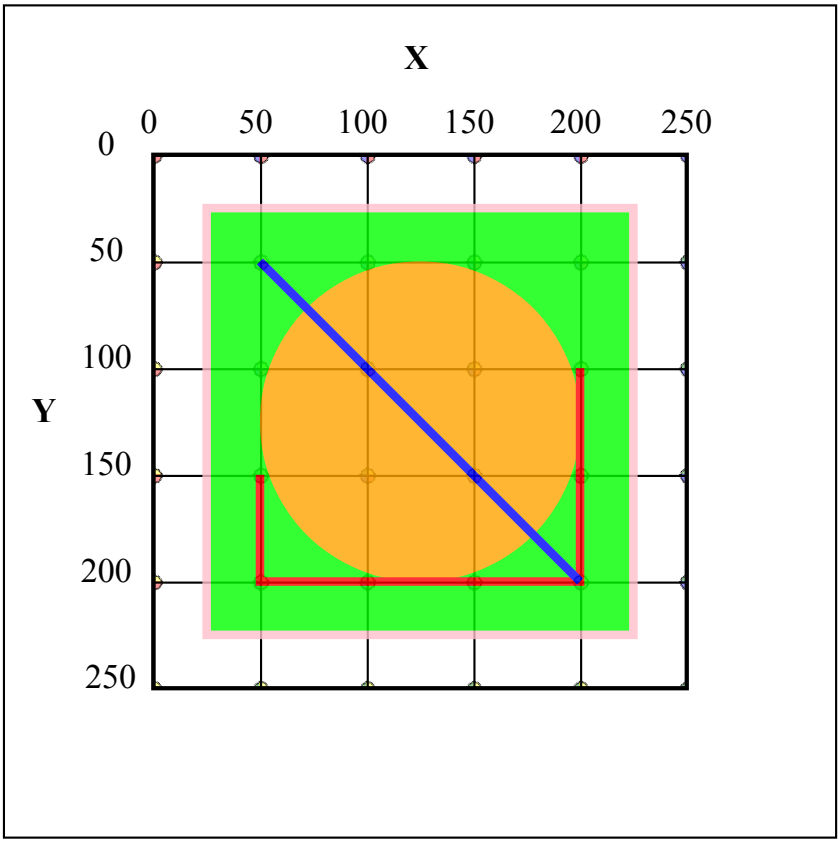
* Scriptare

* Animații : conținutul SVG poate fi animat folosind elemente precum <animate> , <animateMotion> și <animateColor>

* Fonturi

* Metadata

Exemplu :




```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <rect x="25" y="25" width="200" height="200" fill="lime" stroke-width="4" stroke="pink" />
  <circle cx="125" cy="125" r="75" fill="orange" />
  <polyline points="50,150 50,200 200,200 200,100" stroke="red" stroke-width="4" fill="none" />
  <line x1="50" y1="50" x2="200" y2="200" stroke="blue" stroke-width="4" />
</svg>
```

În acest exemplu este desenat totul în afară de grilă și texte. Acesta poate fi de-asemenea scris într-un notepad, salvat cu formatul SVG, și apoi deschis cu un browser modern precum Firefox sau Chrome.

3. Despre implementarea lucrării

În proiectul meu am creat trei clase de vectorizare care extind o clasă BaseVectorizer.

Clasa BaseVectorizer a fost creată pentru ușura tastarea codului în restul aplicației, astfel reducând cantitatea de cod care trebuie scrisă și lăsând posibilitatea de a integra cu ușurința vectorizatoarele în codul pentru interfața grafică.

Descrierea clasei BaseVectorizer

```
protected BufferedImage originalImage;

protected BufferedImage destImage;
```

În *originalImage* este păstrată referința către poza originala în format raster, iar în *destImage* se pastrează imaginea în format vectorial și apoi afișată în interfața grafică. Motivul pentru care am ales obiectele să fie de tipul *BufferedImage* este datorită funcției *getRGB(int x,int y)* care ajută la obținerea informațiilor. Din păcate nu am avut încredere deplină în viteza de procesare a funcției *getRGB* așa că am creat vectori ajutători.

```
protected char[] originalRedArray;
protected char[] originalGreenArray;
protected char[] originalBlueArray;
protected int[] originalColorArray;

public void calculateColorArrays()
```

În acești vectori sunt păstrate valorile culorilor roșu, verde, albastru și combinația lor. Motivul de ce primii trei vectori sunt de tip *char* este deoarece Java folosește acest tip de date pentru a stoca valori de la 0 la 255 spre comparație de tipul *byte* care stochează valori de la -128 la 127. Acești vectori vor fi folosiți pentru a calcula rapid media culorilor dintr-o zonă. În descrierea vectorizatoarelor vor fi mai multe detalii despre acestea.

Metoda *calculateColorArrays()* este chemată imediat după ce s-a setat un nou obiect *originalImage*.

```
protected short w,h;

protected int area;
```

Acestea sunt variabile ajutătoare pentru alte funcții.

```
public int threshold=-1;
```

Valoarea *threshold* este una dintre cele mai folosite în tot codul. Ea decide cât de agresiv este procesul de vectorizare atunci când vine vorba de compararea culorilor. Dacă suma modulelor diferențelor valorilor roșu, verde și albastru al culorilor dintre doi pixeli este mai mare decât valoarea *threshold* atunci acei doi pixeli nu pot aparține aceleași figuri geometrice. Astfel dacă *threshold* are o valoare mai mică atunci exista o șansă mai mare ca doi pixeli învecinați să aparțină unor figuri geometrice diferite astfel crescând mărimea fizică a fișierului dar și claritatea imaginii.

```
protected ImagePanel destImagePanel;
```

Clasa *ImagePanel* a fost creată deoarece nu există element de interfață grafică făcut special pentru a afișa o imagine.

```
protected JobThread lastJob;
```

Clasa *JobThread* a fost creată deoarece clasa Thread nu conține o metoda de încredere de a opri subit o linie de execuție. Cea mai potrivită ar fi fost metoda *stop()*, care din păcate este deprecată.

Aceasta clasa acum conține un boolean numit *canceled* care inițial are valoarea false. Atunci

când se apelează metoda *setCanceled(true)*, *canceled* va primi valoarea true. Acum depinde de ce se afla în implementarea metodei run(). Ea trebuie să verifice în mai multe puncte dacă valoarea *canceled* a devenit true și să oprească execuția într-un mod curat (exemplu: închiderea unor stream-uri sau thread-uri secundare).

```
protected final Object jobLock=new Object();

public abstract void startJob();
public abstract void cancelLastJob();
```

Obiectul mutex *jobLock* este folosit în blocurile *synchronized* din metodele *startJob* și *cancelLastJob* atunci când un vectorizator dorește crearea unui nou thread *jobThread* sau oprirea celui curent.

Pentru a permite userului sa încerce rapid diferite valori pentru *threshold*, codul trebuie să poate crea și opri cât mai rapid thread-urile dar în același timp să elimine posibile probleme de multi-threading precum „data racing". Metoda *startJob* are grijă mai întâi să verifice dacă există deja un obiect *jobThread* care rulează, dacă da, îi oprește execuția cu metoda *setCanceled* și așteaptă ca el să se oprească. După aceea se poate crea un nou obiect *jobThread* în siguranță. Metoda *cancelLastJob* execută la fel în afară de crearea noului thread.

```
public abstract void exportToSVG(OutputStream os,boolean isCompressed);
```

Atunci când se dorește exportarea rezultatului curent în format SVG sau SVGZ, se deschide un fișier , se obține OutputStream-ul lui, se alege dacă va fi exportat comprimat sau nu, și se apelează metoda. Dacă s-a ales opțiunea de comprimare a fișierului , folosesc un obiect de tip GZIPOutputStream pentru comprimare. Ce este interesant cu acest obiect este că metoda de comprimare este compatibila cu aplicația InkScape pentru a deschide fișierul nou creat. După aceea obiectul este înfășurat cu un BufferedOutputStream, acesta poate ajuta la procesele lente de scriere.

```
protected static final DecimalFormat singleDecimalFormat = new
DecimalFormat("#.#", DecimalFormatSymbols.getInstance(Locale.US));
```

Acest obiect este folosit pentru a crea string-uri bazate pe acel format „#.#" și numere reale trimise ca parametru la metoda *format*. Rolul celui de-al doilea parametru din constructor este pentru a preciza că partea întreagă și partea fracționară se desparte cu un punct. String-urile respective sunt folosite în metoda *exportToSVG*. Dacă partea întreaga și partea fracționară ar fi fost separate de o virgulă, atuncea formatul SVG nu ar fi fost scris corect, deoarece SVG folosește virgula pentru a separa coordonate.

Am ales să afișez doar o singură zecimală deoarece am dorit să păstrez dimensiunea fișierelor cât mai mică și un nivel bun de precizie.

Exemple de numere formatate : 3.14159 => „3.1" ; 42 => „42" ; 621.59 => „621.6" .

```
public char redOrig(int x,int y){return originalRedArray[y*w+x];}
public char blueOrig(int x,int y){return originalBlueArray[y*w+x];}
public char greenOrig(int x,int y){return originalGreenArray[y*w+x];}
public int colorOrig(int x,int y) {return originalColorArray[y*w+x];}
```

Metode ajutătoare pentru accesarea valorilor culorilor pentru anumiți pixeli. Numele metodelor sunt foarte simpliste și scurte deoarece ele sunt foarte des folosite și ar umple codul prea

mult.

```
public void initialize(){calculateColorArrays();}
```

Este chemată atunci când *setOriginalImage(BufferedImage image)* primește ca parametru un *image* care este diferit de fostul obiect *originalImage*.

```
public void setOriginalImage(BufferedImage image)
```

Metoda aceasta verifică dacă imaginea setată este *null* atunci să oprească *jobThread*-ul curent și să reseteze valorile auxiliare la valorile lor default. Altfel dacă noul *image* este diferit fată de cel vechi atunci se va executa procesul de inițializare.

```
protected StringBuilder svgStringBuilder = new StringBuilder(2000000);
protected StringBuilder svgzStringBuilder = new StringBuilder(2000000);
protected ByteArrayOutputStream baos = new ByteArrayOutputStream(2000000);
protected GZIPOutputStream gzos;

protected abstract void constructStringSVG();
```

Metoda *constructStringSVG* are rolul de a construi conținutul fișierelor în formate SVG și SVGZ și de a le stoca în obiectele *svgStringBuilder* și *svgzStringBuilder*. Motivul de ce păstrez referința acestor trei obiecte este pentru a anula costul de construire și deconstruire al acestora atunci când apelez metoda *constructStringSVG*. Ca și optimizare în plus le ofer un spațiu generos inițial la prima lor construcție. Metoda aceasta trebuie sa apeleze metoda *constructStringSVGZ* la final.

```
protected void constructStringSVGZ()
```

Metoda aceasta a fost creată pentru a încapsula convertirea string-ului din *svgStringBuilder* în formatul SVGZ și stocarea lui în *svgzStringBuilder*.

Implementarea metodelor startJob și cancelLastJob

```
public void startJob() {

    synchronized (jobLock) {
        if (lastJob != null) {
            lastJob.setCanceled(true);
            try {
                lastJob.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        setIsDone(false);
        lastJob = new Job();
        aproxCompletedPixelCount.set(0);
        lastJob.start();
    }
}
```

După ce este asigurat accesul unui singur thread la date importante, se verifică dacă există obiectul *lastJob* este diferite de null. Dacă da, este posibil el să conțină un thread care încă rulează. Acest thread este anulat prin apelarea metodei *setCanceled* și apoi se așteaptă oprirea execuției lui folosind metoda *join*.

Acum că sa asigurat faptul că nu exista niciun thread de tipul *Job* care rulează, obiectul

lastJob primește referința unui nou thread Job și pornește execuția lui.

Metoda *setIsDone* este folosit pentru a anunța testul de performanță dacă un thread de tipul *Job* a terminat de executat sau nu.

Ca ultima asigurare înainte să pornească noul thread, valoarea atomică *aproxCompletedPixelCount* este setată la 0.

```
public void cancelLastJob(){

	synchronized (jobLock){
		if (lastJob != null) {
			lastJob.setCanceled(true);
			try {
				lastJob.join();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		setIsDone(true);
	}
}
```

Metoda *cancelLastJob* este foarte similara la prima vedere.

Se folosește tot obiectul *lastJob* pentru a asigurare siguranța datelor. Se așteaptă oprirea executiei ultimului job curent. Și se apeleaza functia *setIsDone(true)* pentru a anunțat benchmark-ul ca vectorizatorul a terminat ceva.

Descrierea clasei RectangleVectorizer

Acest vectorizator creează o listă de dreptunghiuri care acoperă întreaga suprafață a pozei originale. Fiecare dintre dreptunghiuri are o culoare care este calculată făcând media culorilor pixelilor care aparțin dreptunghiului respectiv.

Pentru a ajuta codarea acestei clase, am creat o clasă ajutătoare *RectangleFragment* care stochează marginile left, right, top, down și culoarea dreptunghiului. Toate câmpurile sunt publice pentru a fi folosite mai rapid. Deoarece exista șansa ca un dreptunghi să fie generat greșit, am creat metoda *isValid(),* care returnează *true* dacă marginile sunt valide.

Procesul de divizare în dreptunghiuri este recursiv, încercând prima dată un dreptunghi care ocupă toată suprafața imaginii, și dacă acesta nu este considerat bun el va fi fragmentat în 4 dreptunghiuri și procesul continuă de acolo.

Am decis ca procesul de fragmentare să fie randomizat și în același timp limitat. Aleg aleator doua valori *midX* și *midY* ambele aflându-se intre o pătrime și trei pătrimi din lungimea și lățimea dreptunghiului original, astfel lăsând posibilitatea utilizatorului sa reîncerce vectorizarea până când este mulțumit. Din cauza naturii aleatoare, este foarte posibil ca mărimea fișierului generat după o imagine sursa și un *threshold* anume să difere considerabil.

În primele revizii ale programului valorile *midX* și *midY* erau pur și simplu alese să fie în mijlocul dreptunghiului, dar acestea generau imagini vectoriale care avea aspect de „caiet de

matematica”; arătând tabelat și științific. Acum că cele două valori sunt alese aleator imaginile generate au un aspect mai plăcut.

Procesul de validare constă mai întâi din generarea a unei medii aproximative a culori (calculată folosind doar vârfurile dreptunghiului). Dacă vreunul dintre pixeli este prea diferit de către aceasta media aproximativa, dreptunghiul este respins. În timp ce se iterează toți pixeli, se însumează culorile lor pentru a calcula media adevărata a culorilor. Dacă un dreptunghi este respins acesta este tăiat în patru și se va încerca validarea acestora. Dacă un dreptunghi este valid , media adevărata a culorilor din acel dreptunghi este stocata în câmpul *color* de la obiectul *RectangleFragment* folosit la validare, apoi el este adăugat unei liste care a fost trimisă ca argument.

Am decis să trimit ca parametru un *LinkedList* gol fiecărui thread muncitor care să adune toate rezultatele găsite pentru a putea abuza de multithreading fără a avea probleme de data racing când încerc să adaug obiecte la *LinkedList* comun. Din păcate din cauza acestei decizii acum numărul minim de dreptunghiuri care poate reprezenta o imagine este patru, dar acest detaliu este neglijabil având în considerare faptul că acum lucrează patru thread-uri, fiecare thread rulând metode recursiva cu patru *LinkedList*-uri independente și patru obiecte *RectangleFragment* care au fost generate aleator din *RectangleFragment*-ul principal.

Când fiecare din thread-uri își termina execuția, rezultatele sunt depuse în *LinkedList*-ul respectiv sunt acuma depuse de către thread-ul principal într-un *ArrayList* de obiecte *RectangleFragment*. Am ales *ArrayList* pentru performanța sa sporita la citire.

După ce s-a generat lista de dreptunghiuri, se construiesc string-urile care vor conține lista în formatele SVG și SVGZ. Am decis să construiesc acest string ca să pot afișa pe ecran mărimea fișierului când este salvat în formatele SVG și SVGZ. De-asemenea se creează un nou obiect *BufferedImage* în care este creată o aproximație a imaginii vectoriale și apoi este afișată utilizatorului.

Implementarea clasei RectangleVectorizer

```
private Random random = new Random(System.currentTimeMillis());
```

Valoare aleatoare folosită la fragmentarea dreptunghiurilor.

```
private ArrayList<RectangleFragment> lastRectangleList;
```

Obiectul acesta ține o referință către ultima lista de dreptunghiuri care a fost generată de către un thread *Job*.

```
public void drawFunction(ArrayList<RectangleFragment> list){

    if(destImage!=null || isInBenchmark) {
        Graphics2D g = destImage.createGraphics();
        for (RectangleFragment s : list) {
            g.setColor(new Color(s.color));
            g.fillRect(s.l, s.t, s.r - s.l + 1, s.d - s.t + 1);
        }
        if(!isInBenchmark) {
            destImagePanel.setImage(destImage);
        }
```

```
    }
}
```

Metoda aceasta este folosita pentru a afișa utilizatorului rezultatul aproximativ (și superficial) al vectorizării. Mai întâi se verifica dacă există referință către obiectul *ImagePanel*, sau dacă vectorizarea este făcută doar pentru un test de performanță.

Se iterează prin toată lista de obiecte *RectangleFragment*, la fiecare pas setând culoarea dreptunghiului, și umplerea ei folosind metoda *fillRect*. Am folosit metoda *fillRect* în loc de *fill* deoarece aceasta este specializata pentru a desena dreptunghiuri. Motivul de ce am adăugat 1 la lățime și lungime este deoarece metoda scade cu 1 valorile înainte de desenare.

Dacă vectorizarea a fost făcută cu scopul măsurării performanței sale, nu se va afișa nimic utilizatorului.

```
protected void constructStringSVG()
```

Metoda creata pentru construirea string-urilor care va conține imaginea vectoriala în formatele SVG și SVGZ.

```
svgStringBuilder.setLength(0);
```

Deoarece exista o șansă foarte mare ca utilizatorul să încerce mai multe *threshold*-uri până când este mulțumit, se creează un obiect *StringBuilder* care este resetez la zero de fiecare dată când se începe scrierea formatelor SVG/SVGZ. Chiar dacă el este resetat, el își păstrează vechiul buffer folosit dinainte, deci nu va fi overhead în distrugerea și recrearea „string-ului” său intern.

```
svgStringBuilder.append(String.format("<svg xmlns='http://www.w3.org/2000/svg'
version='1.1' width='%d' height='%d'>", w, h));
```

Prima linie pe care o conține SVG-ul va descrie în ce format este scris, și ce lățime și înălțime va avea imaginea vectorizată.

```
for(RectangleFragment sf : lastRectangleList) {
    svgStringBuilder.append(String.format("<rect x='%d' y='%d' width='%d'
height='%d' style='fill:#%06X'/>\n",
            sf.l,
            sf.t,
            sf.r-sf.l+2,
            sf.d-sf.t+2,
            sf.color&0xffffff));
}
```

Am decis sa folosesc tag-ul *rect* pentru al face mai natural de citit , și manipulat într-un editor de fișiere vectoriale precum Inkscape. Dacă aș folosi un tag mai compact precum *path* atuncea dimensiunea fișierelor în format SVG ar scădea foarte mult dar nu s-ar mai putea citi așa de ușor.

Am observat că obțin rezultate corecte din punct de vedere vizual dacă cresc cu 2 lățimea și lungimea dreptunghiurilor. Un mic avantaj la *RectangleVectorizer* este faptul că toate coordonatele sunt numere întregi, ceea ce face construirea string-ului mai rapidă.

```
svgStringBuilder.append("</svg>");
```

La finalul fișierului, se adaugă tag-ul de sfârșit al formatului SVG.

```
constructStringSVGZ();
```

Metoda aceasta este apelată pentru a crea string-ul în format SVGZ.

Subclasa Job a clasei RectangleVectorizer

Clasa aceasta are rolul de a crea în mod relativ eficient lista de dreptunghiuri a procesului de vectorizare. Ea extinde clasa JobThread care conține un flag boolean *canceled*.

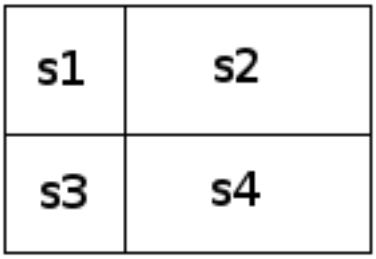
```
private ArrayList<RectangleFragment> fragList = new ArrayList<>();
```

Aici va tine ca referință lista de dreptunghiuri la care lucrează thread-ul, dacă operația de vectorizare se termină cu succes, referința aceasta va fi copiată în lastSavedRect*angle*List.

Am ales sa folosesc *ArrayList* pentru aceasta lista pentru a avea viteza constanta la citirea datelor din ea.

```
private void splitRectangleFragmentInFour(RectangleFragment s, RectangleFragment
s1, RectangleFragment s2, RectangleFragment s3, RectangleFragment s4){

    short midX = (short) (s.l + (random.nextFloat() / 2 + 0.25f) * (s.r - s.l));
    short midY = (short) (s.t + (random.nextFloat() / 2 + 0.25f) * (s.d - s.t));
    s1.set(s.l, midX, s.t, midY);
    s2.set((short) (midX + 1), s.r, s.t, midY);
    s3.set(s.l, midX, (short) (midY + 1), s.d);
    s4.set((short)(midX + 1), s.r, (short) (midY + 1), s.d);
}
```

Am creat metoda aceasta pentru a crea 4 fragmente dreptunghiulare dintr-unul singur. Mai întâi aleg aleator două valori de pe lungimea și lățimea dreptunghiului mare. Aceasta o calculez ca fiind interpolarea dintre valorile stânga cu dreapta și sus cu jos și o valori aleatoare alese uniform din mulțimea [0.25,0.75].



Exemplu de fragmentare

Metoda a fost creată special pentru a recicla fragmente vechi sau proaspăt create, de aceea trimite ca argumente obiectele *s1,s2,s3,s4*.

```
public void run()
```

În următoarele paragrafe voi explica conținutul metodei *run*.

```
if (originalImage == null || canceled) return;

RectangleFragment rectangleFragment = new RectangleFragment((short)0,(short) (w
- 1),(short) 0, (short)(h - 1), -1);
startTime = System.currentTimeMillis();
RectangleFragment s1 = new RectangleFragment();
RectangleFragment s2 = new RectangleFragment();
RectangleFragment s3 = new RectangleFragment();
RectangleFragment s4 = new RectangleFragment();
splitRectangleFragmentInFour(rectangleFragment,s1,s2,s3,s4);
```

Din prima linie se verifică dacă valoarea flag-ului *canceled* a fost deja setat fals. Este posibil ca utilizatorul sa mute rapid bara de *threshold*, deci este nevoie ca un Job thread să fie oprit imediat pentru a începe unul nou cu un nou *threshold*.

Se generează cele patru dreptunghiuri inițiale generate randomizat de către *splitRectangleFragmentInFour()*.

```
LinkedList<RectangleFragment> fragList1 = new LinkedList<>();

LinkedList<RectangleFragment> fragList2 = new LinkedList<>();
LinkedList<RectangleFragment> fragList3 = new LinkedList<>();
LinkedList<RectangleFragment> fragList4 = new LinkedList<>();
Thread t1 = new Thread(() -> {
    if (s1.isValid()) recFragCheck(s1,fragList1);
});
Thread t2 = new Thread(() -> {
    if (s2.isValid()) recFragCheck(s2,fragList2);
});
Thread t3 = new Thread(() -> {
    if (s3.isValid()) recFragCheck(s3,fragList3);
});
t1.start();
t2.start();
t3.start();
if (s4.isValid()) recFragCheck(s4,fragList4);
```

Se generează câte un container pentru fiecare obiect RectangleFragment. Cu ajutorul celor trei noi thread-uri generate se apelează câte o metodă *recFragCheck* pentru fiecare obiect *RectangleFragment* și lista ei corespunzătoare.

```
try {

    t1.join();
    t2.join();
    t3.join();
    if(canceled)return;
    fragList.ensureCapacity(fragList1.size()+fragList2.size()+
            fragList3.size()+fragList4.size());
    fragList.addAll(fragList1);
    fragList.addAll(fragList2);
    fragList.addAll(fragList3);
    fragList.addAll(fragList4);
} catch (InterruptedException e) {
    e.printStackTrace();
}

if(canceled)return;
```

După ce fiecare fir de execuție a terminat de executat metoda *recFragCheck*, se verifică dacă motivul de ce ele au fost oprite a fost din cauza utilizatorului sau nu prin testarea flag-ului *canceled*. Dacă procesul de vectorizare a terminat corect, cele patru liste generate anterior sunt colectate și depuse în vectorul *fragList.* Pentru a mai spori din performanța atunci când sunt copiate obiectele, mai întâi se folosește metoda *ensureCapacity* pentru a asigura un spațiu de memorie suficient de mare și continuu.

Mai fac încă o verificare a flag-ului *canceled* pentru că tocmai s-au executat câteva linii care au o durata considerabilă de timp de execuție.

```
lastRectangleList = fragList;

Thread th = new Thread(() -> {
```

```
    constructStringSVG();
    updateDetails(String.format("SVG:%s SVGZ:%s",
            Utility.aproximateDataSize(svgStringBuilder.length()),
            Utility.aproximateDataSize(svgzStringBuilder.length())));
});
th.start();
drawFunction(lastRectangleList);
try {
    th.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
setIsDone(true);
```

Acum că s-au colectat toate dreptunghiurile într-un container comun, se setează lista de dreptunghiuri a întregului obiect *RectangleVectorizer* cu aceasta.

După aceea, execut pe un thread separat construirea imaginilor în format SVG și SVGZ, iar pe linia de execuție principală creez imaginea de prezentat utilizatorului. Funcția set *isDone()* anunță testul de performanță momentul când a terminat vectorizatorul.

```
private void recFragCheck(RectangleFragment s, Collection<RectangleFragment>
localFragList)
```

Această metodă caută în mod recursiv toate dreptunghiurile care se afla în obiectul *s* și le depozitează în lista *localFragList*. Valorile sunt mai ușor de recuperat dacă trimit lista ca parametru.

```
int rTotal, gTotal, bTotal, avgColor;

rTotal=(redOrig(s.l,s.t)+redOrig(s.r,s.t)+redOrig(s.l,s.d)+redOrig(s.r,s.d))/4;
gTotal=(greenOrig(s.l,s.t)+greenOrig(s.r,s.t)+greenOrig(s.l,s.d)+greenOrig(s.r,s.d))/4;
bTotal=(blueOrig(s.l,s.t)+blueOrig(s.r,s.t)+blueOrig(s.l,s.d)+blueOrig(s.r,s.d))/4;
avgColor = 0xff000000 | (rTotal << 16) | (gTotal << 8) | bTotal;
```

Prima dată calculez media culorilor vârfurilor dreptunghiului pentru a aproxima culoare spre care trebuie să tinda restul dreptunghiului. Cu cât dreptunghiul este mai mic cu atât aproximarea va fi mai precisă.

```
boolean fail = false;

rTotal = 0; gTotal = 0; bTotal = 0;
for (int y = s.t; y <= s.d && !fail; y++)
     for (int x = s.l; x <= s.r && !fail; x++) {
          if (canceled) return;
          rTotal += redOrig(x,y); gTotal += greenOrig(x, y); bTotal += blueOrig(x,y);
          if (Utility.manhattanDistance(colorOrig(x, y), avgColor) > threshold)
               fail = true;
     }
```

Resetez sumele de culori și inițializez o valoarea boolean *fail* cu scopul de a monitoriza dacă dreptunghiul curent conține culori prea distincte față de media calculată mai devreme.

Pentru fiecare pixel în parte adun valorile culorilor lor în sumele lor respective și verific dacă culoarea pixelului este suficient de similar cu culoarea *avgColor*. Dacă este prea diferit procesul este întrerupt prin setarea flag-ului *fail* cu valoarea false.

```
if (!fail) {

    int count = (s.d-s.t+1)*(s.r-s.l+1);
    rTotal /= count; bTotal /= count; gTotal /= count;
```

```
    avgColor = 0xff000000 | (rTotal << 16) | (gTotal << 8) | bTotal;
    s.color = avgColor;
    localFragList.add(s);
    int x = aproxCompletedPixelCount.addAndGet(s.area());
    updateDetails(String.format("Progress : %.1f%%", 100.f * x / area));
```

Dacă variabila *fail* a rămas falsă, înseamnă că acest dreptunghi este bun și va fi adăugat în lista *localFragList.* Înainte de adăugare, se setează culoarea corectă a dreptunghiului bazată pe mediile aritmetice a valorilor culorilor pixelilor testați. După aceea se incrementează contorul de pixeli completați și se reîmprospătează valoarea progresului.

```
} else {

    RectangleFragment s1 = new RectangleFragment();
    RectangleFragment s2 = new RectangleFragment();
    RectangleFragment s3 = new RectangleFragment();
    RectangleFragment s4 = new RectangleFragment();
    splitRectangleFragmentInFour(s,s1,s2,s3,s4);
    if (s1.isValid()) recFragCheck(s1,localFragList);
    if (s2.isValid()) recFragCheck(s2,localFragList);
    if (s3.isValid()) recFragCheck(s3,localFragList);
    if (s4.isValid()) recFragCheck(s4,localFragList);
}
```

Dacă variabila *fail* a devenit true, înseamnă că acest dreptunghi a eșuat și va fi fragmentat în patru. Fiecare dintre cele patru noi dreptunghiuri vor fi testate folosind metoda *recFragCheck* și vor fi stocate în *localFragList* dacă sunt valide.

Descrierea clasei TriangleVectorizer

Acest vectorizator creează o lista de triunghiuri care acoperă întreaga suprafață a pozei originale. Fiecare dintre triunghiuri are o culoare care este calculată făcând media aritmetică a culorilor pixelilor care aparțin suprafeței triunghiului respectiv.

Pentru a ajuta codarea acestei clase, am creat o clasa ajutătoare *Triangle* care reține coordonatele vârfurilor reținute în valori *float*, culoarea într-o valoare *integer* și câteva valori ajutătoare precum *xMin*, *yMin*, *xMax* și *yMax* care reprezintă bounding-box-ul triunghiului având lungimea și lățimea paralele cu axele oX și oY ale imaginii. De-asemenea în fiecare obiect de tipul Triangle se mai află și un obiect de tipul Path2D.Float numit *path* care este folosit la optimizarea creării imaginii demonstrative utilizatorului.

Clasa *Triangle* a fost notată ca fiind serializabila și valorile și obiectul ajutător sunt notate ca fiind transient ceea ce înseamnă că ele nu vor fi scrise la serializare.

Procesul de divizare în triunghiuri este recursiv. În faza inițială imaginea care este dreptunghiulară este despărțită în două triunghiuri pe una dintre diagonalele ei. Fiecare triunghi este verificat dacă satisface valoarea curentă a *threshold*-ului. Dacă da, acesta este adăugat la o lista de obiecte *Triangle*. Altfel acesta este fragmentat în 2 triunghiuri și procesul continuă de acolo.

Am decis ca procesul de fragmentare să fie randomizat și în același limitat. Aleg aleator o valoare uniformă *r* între 0.2 și 0.8, apoi caut latura cea mai lunga a triunghiului. Linia care va fragmenta triunghiul în două este trasă de la colțul opus al laturii mari la un punct interpolat intre cele două puncte ale laturi mari folosind valoarea *r.* Din cauza naturii aleatoare, este foarte posibil ca mărimea fișierului generat după o imagine sursa și un *threshold* anume să difere considerabil.

Deoarece *r* este ales aleator, las posibilitatea utilizatorului sa reîncerce vectorizarea până când este mulțumit de rezultat.

În primele revizii ale programului valoarea *r* era aleasă direct ca fiind 0.5, dar aceste genera un efect caleidoscopic care deși arăta interesant era prea predictiv.

Valorile alese aleator pentru r da un aspect mai plăcut imaginii vectoriale generate.

Procesul de validare al unui triunghi începe cu calcularea unei medii aproximative a culorilor folosind doar vârfurile sale. După aceea iterând fiecare pixel care aparține suprafeței triunghiului se verifică suma modulelor diferențelor valorilor culorilor dintre media aproximativa și pixelul verificat este mai mare decât valoarea *threshold,* atunci acest triunghi este respins. Dacă un triunghi este respins acesta este tăiat în doua și se va încerca validarea acestora. Este posibil ca din cauza erorilor de calcul ca un triunghi sa ajungă sa aibă aria mai mica de 0.5, dacă se întâmplă acest lucru triunghiul va fi ignorat. Dacă un triunghi este valid, media aritmetică a culorilor din acel triunghi este calculată și stocata în câmpul *color* de la obiectul *Triangle* folosit la validare, iar apoi este adăugat unei liste care a fost trimisă ca argument.

Am decis să trimit ca parametru un *LinkedList* gol care să adune toate rezultatele găsite pentru a putea abuza de multithreading fără a avea probleme de data racing când încerc să adaug obiecte la același *LinkedList*. În momentul acesta procesul de vectorizare folosește doar doua thread-uri, cele care au fost create când s-au creat primele două triunghiuri ale imaginii. Deoarece performanța este satisfăcătoare am decis să nu folosesc mai multe thread-uri, dar este într-adevăr posibilă folosirea mai multor thread-uri.

Când fiecare din thread-uri își termina execuția, rezultatele depuse în *LinkedList*-ul respectiv sunt acum depuse de către thread-ul principal într-un *ArrayList*. Am ales *ArrayList* pentru performanța sporita la citire.

După ce s-a făcut rost de *ArrayList*-ul cu toate triunghiurile, se construiesc string-urile care vor conține lista în formatele SVG și SVGZ. Am decis să construiesc aceste string-uri devreme pentru a afișa utilizatorului mărimea fișierului când este salvat în formatele SVG și SVGZ.

Implementarea clasei TriangleVectorizer

```
private Random random = new Random(System.currentTimeMillis());
```

Obiectul *random* este folosit pentru a genera valori aleatoare.

```
private ArrayList<Triangle> lastSavedTriangleList = null;
```

Aceasta doar ține o referință către ultima lista de triunghiuri care a fost generată de către un *JobThread*.

```
private void drawTriangles(ArrayList<Triangle> list){

    if(destImagePanel!=null || isInBenchmark){
        Graphics2D g = destImage.createGraphics();
        for(Triangle t:list){
            g.setColor(new Color(t.color));
            g.fill(t.getPath());
```

```
            }
            if(!isInBenchmark) {
                destImagePanel.setImage(destImage);
            }
        }
}
```

Metoda aceasta este folosita pentru a afișa userului rezultatul aproximativ (și superficial) al vectorizării. Mai întâi se verifica dacă există referință către obiectul *ImagePanel* , sau dacă vectorizarea este făcută doar pentru un test de performanță.

Se iterează prin toată lista de obiecte *Triangle*, la fiecare pas setând culoarea triunghi-ului, și umplerea figurii geometrice definite de către obiectul triunghi (t.*getPath()*).

Dacă vectorizarea a fost făcută cu scopul măsurării performanței sale, nu se va afișa nimic utilizatorului.

```
protected void constructStringSVG()
```

Metoda creată pentru construirea string-urilor care vor conține imaginea vectoriala în formatele SVG și SVGZ.

```
svgStringBuilder.setLength(0);
```

Deoarece exista o șansă foarte mare ca utilizatorul să încerce mai multe *threshold*-uri până când este mulțumit, se creează un obiect *StringBuilder* care este resetez la zero de fiecare dată când se incepe scrierea formatelor SVG/SVGZ. Chiar dacă el este resetat, el își păstrează vechiul buffer folosit dinainte, deci nu va fi overhead în distrugerea și recrearea „string-ului” său intern.

```
svgStringBuilder.append(String.format("<svg xmlns='http://www.w3.org/2000/svg'
version='1.1' width='%d' height='%d'>", w, h));
```

Prima linie pe care o conține SVG-ul va descrie în ce format este scris, și ce lățime și înălțime va avea imaginea vectorizată.

```
svgStringBuilder.append(String.format("<g stroke-width='0.5'>"));
```

Din cauza erorilor de calcul cu valori *float* încep să apară mici spații intre triunghiurile care ar trebui să fie vecine. Atunci când un utilizator deschidea imaginea vectorială într-un editor sau într-un browser, el/ea putea să observe numeroase linii albe între triunghiuri. Efectul este și mai evident dacă poza este mărită folosind zoom.

Așa că aplic o proprietate comună la toate triunghiurile, toate vor avea perimetrul său desenat cu o linie de grosime 0,5. Creșterea acestei valori ajuta la ascunderea liniilor albe dintre triunghiuri, dar nu rezolva perfect situația. Dacă valoarea aceea de 0,5 este crescuta mai mult , triunghiurile cresc mai mult și încep să se suprapună.

```
for(Triangle t:lastSavedTriangleList){

    svgStringBuilder.append(String.format("<path d='M%s,%sL%s,%sL%s,%sZ' fill='#%06X'
stroke='#%06X'/>",
                decimalFormat.format(t.x0),
                decimalFormat.format(t.y0),
                decimalFormat.format(t.x1),
                decimalFormat.format(t.y1),
                decimalFormat.format(t.x2),
```

```
                decimalFormat.format(t.y2),
                t.color&0xffffff,
                t.color&0xffffff));
}
```

Iterând fiecare triunghi din ultima listă salvată de triunghiuri, încep să adaug fiecare dintre ele în string-ul pentru formatul SVG.

Am ales tag-ul *path* pentru a mai reduce din numărul total de caractere. Litera 'M' are rolul de a muta „pensula”, fără a desena o linie, la coordonatele absolute definite de următoarele două numere. Litera 'L' are rolul de a trasa o linie de la ultima poziție a pensulei la următoarea coordonată absolută. Iar litera 'Z' este folosită pentru a închide figura geometrică desenată.

Exemplu de triunghi definit cu *path, polyline* și *polygon:*

```
<path d='M0.4,0.2L0.4,5.6L5.1,0.2' fill='#ff0000' stroke='ff0000'/>
<polyline points='0.4,0.2 0.4,5.6 5.1,0.2 0.4,0.2' fill='#ff0000' stroke='ff0000'/>
<polygon points='0.4,0.2 0.4,5.6 5.1,0.2' fill='#ff0000' stroke='ff0000'/>
```

Atributele *fill* și *stroke* primesc ca valoare o culoare definită asemănător stilului HTML. Format-ul *%06X* definește un număr hexazecimal în care dacă numărul său de cifre nu depășește 6 va fi umplut la stânga cu cifre de 0 până când are 6 cifre.

Am ales să scriu culorile în formatul hexazecimal deoarece este cea mai scurta opțiune la dispoziție. A trebuit să am o precauțiune în plus la definirea culorilor, așa ca păstrez numai biții care sigur au numai valori pentru roșu, verde și albastru. Operația aceasta de filtrare a biților o realizez prin AND binar cu numărul *0xffffff*.

```
svgStringBuilder.append("</g>");

svgStringBuilder.append("</svg>");
```

La finalul fișierului SVG, se adaugă tag-ul de sfârșit pentru proprietatea care dă la toate triunghiurile o linie groasă, și tag-ul de sfârșit al întregului format SVG.

O să presupun aici ca JIT-ul o să mai optimizeze codul atunci când sunt create foarte multe obiecte String. Dar am mai lucrat cu *StringBuilder* și am obținut performanțe ridicate.

Acesta a fost doar conținutul imaginii în format SVG. Din fericire convertirea ei în SVGZ este foarte ușoară de făcut în Java.

```
constructStringSVGZ();
```

Metoda aceasta este apelată pentru a crea string-ul în format SVGZ.

Subclasa Job a clasei TriangleVectorizer

Clasa aceasta are rolul de a crea în mod relativ eficient lista de triunghiuri a procesului de vectorizare. Ea extinde clasa *JobThread* care conține un flag boolean *canceled*.

```
private ArrayList<Triangle> triangles = new ArrayList<>();
```

Aici se va ține ca referință lista de triunghiuri la care lucrează thread-ul. Dacă operația de vectorizare se termină cu succes, referința aceasta va fi copiata mai târziu în *lastSavedTriangleList*.

Am ales sa folosesc *ArrayList* pentru aceasta lista pentru a avea viteză constantă la citirea obiectelor din ea.

```
public void run()
```

În următoarele paragrafe voi explica conținutul metodei *run*.

```
if(canceled)return;

final Triangle t1 = new Triangle(0,0,w-1,0,w-1,h-1);
final Triangle t2 = new Triangle(0,0,0,h-1,w-1,h-1);
```

Din prima linie se verifica dacă valoarea flag-ului *canceled* a fost deja setată ca fiind fals. Este posibil ca utilizatorul să mute rapid bara de *threshold*, deci este nevoie ca un *Job* thread să fie oprit imediat pentru a începe unul nou cu un nou *threshold*.

Cele două triunghiuri sunt folosite pentru a împărți în jumătate lucrul funcției. Triunghiul *t1* este deasupra diagonalei principale, iar *t2* este dedesubtul diagonalei principale.

```
LinkedList<Triangle> triangleArray1 = new LinkedList<>();

LinkedList<Triangle> triangleArray2 = new LinkedList<>();
Thread th = new Thread(() -> {
     recTriangulation(t1,triangleArray1);
});
th.start();
recTriangulation(t2, triangleArray2);
try {
     th.join();
} catch (InterruptedException e) {
     e.printStackTrace();
}
if(canceled)return;
```

Aceste linii de cod par puțin exagerate, dar dacă sunt urmărite cu atenție, ele defapt executa *recTriangulation()* pe două fire de execuție. Una pe cea curentă, una pe un thread nou.

Inițializez două liste dublu înlănțuite care au rolul de a depune rezultatele celor două linii de execuție. Motivul de ce am ales sa folosesc *LinkedList* în loc de *ArrayList*, este deoarece aceste liste vor avea foarte multe operații de adăugare la sfârșitul lor. Un *ArrayList* va avea probleme de performanța atunci când este epuizat buffer-ul sau intern de la prea multe adăugări, dar un *LinkedList* nu va avea aceeași problemă.

Creez un nou thread, acesta va executa *recTriangulation* pe *t1* și va depune rezultatele în prima lista. După aceea pornesc acel thread cu *start()*. Acum că el a pornit vectorizarea pe triunghiul *t1*, încep vectorizarea pe firul principal executând *recTriangulation(t2, triangleArray2);*

Dacă dintr-un motiv vectorizarea pe *t2* a reușit să termine mai repede, trebuie să mă asigur că celălalt triunghi a fost terminat și el. Execut metoda *join()* pe thread-ul creat anterior, astfel dacă thread-ul nu a terminat, firul principal va aștepta până când acesta termină, sau dacă thread-ul deja a terminat, *join()* va returna imediat.

Exista posibilitatea ca flag-ul *canceled* să fi fost setat pe true. Chiar dacă el a fost setat pe true, pentru a avea o organizare mai bună a memoriei interne de către JIT, trebuie așteptată mai întâi terminarea execuției thread-ului anterior, și abia apoi după ce acel thread-ul a terminat în siguranță se poate închide thread-ul Job.

```
triangles.ensureCapacity(triangleArray1.size() + triangleArray2.size());

triangles.addAll(triangleArray1);
triangles.addAll(triangleArray2);
if(canceled)return;
```

Dacă cele două linii de execuție au executat cu succes *recTriangulation*, trebuie culese informațiile adunate de către ele într-o listă comună. Prima dată mă asigur că *ArrayList*-ul *triangles* va avea o zonă de memorie suficient de mare pentru a putea include toate datele dintr-o dată. După aceea ele sunt adăugate una câte una.

Mai fac încă o verificare a flag-ului *canceled* pentru că tocmai s-au executat câteva linii care au o durata considerabilă de timp de execuție.

```
lastSavedTriangleList = triangles;

th = new Thread(new Runnable() {
    @Override
    public void run() {
        constructStringSVG();
        updateDetails(String.format("SVG:%s SVGZ:%s",
                Utility.aproximateDataSize(svgStringBuilder.length()),
                Utility.aproximateDataSize(svgzStringBuilder.length())));
    }
});
th.start();
drawTriangles(triangles);
try {
    th.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
setIsDone(true);
```

Acum că s-au colectat toate triunghiurile într-o listă comună, se setează lista de triunghiuri a întregului obiect *TriangleVectorizer* cu aceasta.

După aceea, execut pe un thread separat construirea imaginilor în format SVG și SVGZ, iar pe linie de execuție principală creez imaginea de prezentat userului. Funcția set *isDone()* anunță testul de performanță momentul când a terminat vectorizatorul.

```
public void recTriangulation(Triangle triangle,AbstractList<Triangle> triangles)
```

Această metodă caută în mod recursiv toate triunghiurile care se află în obiectul *triangle* și le depozitează în lista *triangles*. Valorile sunt mai ușor de recuperat dacă trimit lista ca parametru.

```
float i0=0,i1=0,man;

int flag;
float a;
float x0 = triangle.x0;
float x1 = triangle.x1;
float x2 = triangle.x2;
float y0 = triangle.y0;
float y1 = triangle.y1;
float y2 = triangle.y2;
float xMin = triangle.xMin;
float xMax = triangle.xMax;
float yMin = triangle.yMin;
float yMax = triangle.yMax;
int rTotal=0,gTotal=0,bTotal=0,count=0;
```

```
int rMed=0,gMed=0,bMed=0;
int m;
```

Aici definesc și inițializez diverse variabile ajutătoare. Valorile min și max sunt generate de către triunghi atunci când este construit cu cele trei coordonate ale sale. Marea majoritate dintre aceste variabile le-am pus aici pentru a mai reduce din numărul de caractere scrise în cod , și pentru al face mai ușor de citit.

```
rMed += redOrig((int)x0,(int)y0);

rMed += redOrig((int)x1,(int)y1);
rMed += redOrig((int)x2,(int)y2);
bMed += blueOrig((int) x0, (int) y0);
bMed += blueOrig((int) x1, (int) y1);
bMed += blueOrig((int) x2, (int) y2);
gMed += greenOrig((int) x0, (int) y0);
gMed += greenOrig((int) x1, (int) y1);
gMed += greenOrig((int) x2, (int) y2);
rMed/=3;
gMed/=3;
bMed/=3;
```

Aproximez o media a culorilor după care se va încerca să se facă comparația cu *threshold*-ul curent ales. Adun valorile culorilor RGB din cele trei colturi ale triunghiului curent verificat, apoi împart la 3. Chiar dacă fac împărțire int la int văd că nu sunt probleme la vectorizare.

```
boolean fail = false;

for (int y = (int) yMin; y <= yMax && !fail; y++) {
```

Iterez de sus în jos toate liniile orizontale care se intersectează cu triunghiul. De-asemenea verific dacă variabila *fail* a devenit true, deoarece este posibil ca un pixel să fi fost prea diferit pentru valoarea curentă *threshold*.

```
flag = 0;

a = 1.f * (y - y0) / (y1 - y0);
flag += ((a >= 0) && (a <= 1)) ? 1 : 0;

if (flag == 1) {

    i0 = (int) (x0 + (x1 - x0) * a);
}
```

Aici se calculează punctul de intersecție dintre linia y și linia de la (x0,y0) la (x1,y1). Dacă punctul acesta de intersecție se afla în segmentul (x0,y0)(x1,y1), coordonata x a acestui va fi memorata în i0.

Procesul de calculare și respingere este unul rapid și eficient. Variabila *flag* are rolul de număra câte puncte de intersecție s-au găsit până acuma.

```
a = 1.f * (y - y1) / (y2 - y1);

flag += ((a >= 0) && (a <= 1)) ? 1 : 0;
if (flag == 1) {
    i0 = (int) (x1 + (x2 - x1) * a);
} else if (flag == 2) {
    i1 = (int) (x1 + (x2 - x1) * a);
}
```

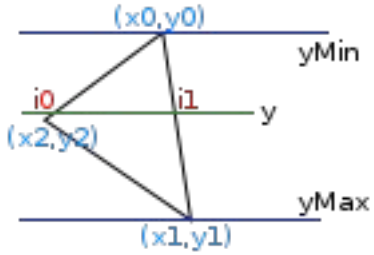
Se repeta aceeași verificare dar pe linia (x1,y1)(x2,y2). În funcție de valoarea lui *flag* al doilea punct de intersecție va fi notat fie pe i0 fie pe i1.

```
if (flag == 1) {

    a = 1.f * (y - y2) / (y0 - y2);
    i1 = (int) (x2 + (x0 - x2) * a);
}
```

Este posibil ca *flag* să fi rămas egal cu 1, deci asta înseamnă că s-a descoperit decât un singur punct de intersecție până acuma din primele două segmente ale triunghiului, deci cu siguranța trebuie să facă intersecție cu ultimul segment al triunghiului și el va fi notat în i1.

```
if (i0 > i1) {

    man = i0;
    i0 = i1;
    i1 = man;
}
i0 = (int) Math.floor(i0);
i1 = (int) Math.ceil(i1);
if(i0 < xMin) i0 = xMin;
if(i1 > xMax) i1 = xMax;
```

Acesta este procesul de aranjare și rafinare a datelor găsite. Mai întâi ele sunt orientate corect de la stânga la dreapta, apoi se rotunjesc marginile spre numere întregi fără să li se dea voie sa iasă din segmentul [xMin,xMax].



Exemplu de căutare a valorilor i0 și i1

În figura din dreaptă apare un exemplu de triunghi în care se caută valorile valorile *i0* și *i1*pentru dreapta de intersecție *y*.

```
if(canceled)return;
```

Se verifica dacă s-a apelat metoda de anulare.

```
for (int x = (int) i0; x <= i1 && !fail; x++) {

    if(canceled)return;
```

Se iterează de la stânga la dreapta toate coordonatele *x* între cele două puncte de intersecție găsite. La fiecare pas se verifica variabila *fail* dacă a devenit adevărat și dacă da să se oprească procesarea triunghiului. Deoarece acest *for* este destul de strâns, am decis sa pun și aici încă o verificare pentru flag-ul *canceled*.

```
count++;

rTotal += redOrig(x, y);

gTotal += greenOrig(x, y);
bTotal += blueOrig(x, y);
```

Se incrementează contorul de pixeli, și se adaug valorile culorilor la sumele lor corespunzătoare. Acestea vor fi folosite mai târziu pentru a calcula medie adevărată a culorilor.

```
m = Math.abs(rMed - redOrig(x, y)) +

        Math.abs(gMed - greenOrig(x, y)) +
```

```
        Math.abs(bMed - blueOrig(x, y));
```

Se calculează suma modulelor diferențelor valorilor culorilor dintre pixelul curent și media aproximativă a culorilor.

```
if(m > threshold)

    fail = true;
```

Dacă valoarea *m* este mai mare decât *threshold*, înseamnă ca s-a descoperit un pixel care este prea diferit de media aproximativă a culorilor, deci analiza triunghiului este oprită.

```
if(canceled)return;

if(count==0)
    return;
else {
    rTotal /= count;
    gTotal /= count;
    bTotal /= count;
}
```

Dacă dintr-un vreun motiv nu s-a găsit nici măcar nu pixel, procesul de vectorizare în acel triunghi este oprit. Altfel se calculează media adevărata a culorilor.

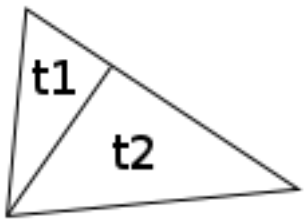
```
if(!fail || triangle.area<=3){

    triangle.color = 0xff000000 | (rTotal<<16) | (gTotal<<8) | bTotal;
    triangles.add(triangle);
    int x = aproxCompletedPixelCount.addAndGet((int)Math.ceil(triangle.area * 100));
    updateDetails(String.format("Progress : %.1f%%",1.f*x/area));
```

Se verifica dacă vreunul dintre pixelii verificați a eșuat testul *threshold* (valoarea lui *fail* e true), dacă da acel triunghi nu va fi adăugat. A trebuit să adaug o excepție la regula aceasta, codul câteodată intra într-un loop recursiv cu triunghiuri din ce în ce mai mici, și aveam nevoie de o metoda eleganta de a lăsa numai anumite triunghiuri sa treacă. Dacă aria triunghiului este mai mică sau egal cu 3, el va avea aproximativ cel mult 3 pixeli întregi și va fi adăugat la listă.

După ce triunghiul a fost adăugat se adaugă aria triunghiului la contorul de pixeli completați și se apelează metoda *updateDetails* pentru a anunța interfața grafică de faptul că trebuie sa reîmprospăteze procentul de completare al vectorizării.

```
}else{

    double dist0 = Math.sqrt((x0-x1)*(x0-x1) + (y0-y1)*(y0-y1));
    double dist1 = Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
    double dist2 = Math.sqrt((x0-x2)*(x0-x2) + (y0-y2)*(y0-y2));
```

Dacă triunghiul a fost respins, acesta va trebuit separat în două. Pentru ca triunghiurile să nu devină prea subțiri sau să aibă un unghi prea obtuz, tăierea triunghiului se face începând de la un punct de pe segmentul cel mai lung și colțul opus.



Exemplu de fragmentare

```
float r = random.nextFloat()*0.6f + 0.2f;

Triangle t1,t2;
if(dist0>=dist1 && dist0>=dist2){
    t1 = new Triangle(Utility.interpolate(x0, x1, r),Utility.interpolate(y0, y1,
```

```
r),x2,y2,x0,y0);
    t2 = new Triangle(Utility.interpolate(x0, x1, r),Utility.interpolate(y0, y1,
r),x2,y2,x1,y1);
}else if(dist1>=dist0 && dist1>=dist2){
    t1 = new Triangle(Utility.interpolate(x2, x1, r),Utility.interpolate(y2, y1,
r),x0,y0,x2,y2);
    t2 = new Triangle(Utility.interpolate(x2, x1, r),Utility.interpolate(y2, y1,
r),x0,y0,x1,y1);
}else{
    t1 = new Triangle(Utility.interpolate(x2, x0, r),Utility.interpolate(y2, y0,
r),x1,y1,x2,y2);
    t2 = new Triangle(Utility.interpolate(x2, x0, r),Utility.interpolate(y2, y0,
r),x1,y1,x0,y0);
}
```

Generez un număr aleator *r* din mulțimea [0.2,0.8]. Folosind valoarea *r* apoi despart triunghiul în două mai mici. Punctul de despărțire este ales prin interpolarea celor două puncte ale laturii cele mai mari cu valoarea numărului *r*.

```
if(t1.area>0.5)recTriangulation(t1,triangles);

if(t2.area>0.5)recTriangulation(t2,triangles);
```

Exista posibilitatea ca după tăierea triunghiului în două, unul dintre ele o să fie foarte foarte mic. Dacă aria triunghiului a devenit mai mică decât o jumătate de pixel, nu va mai fi luat în considerare.

Atunci când un triunghi este considerat suficient de mare, se continuă procesul recursiv.

Descrierea clasei PolygonVectorizer

Acest vectorizator creează o listă de poligoane care acoperă întreaga suprafață a pozei originale. Fiecare dintre poligoane are o culoare care este calculată făcând media aritmetică a culorilor pixelilor care aparțin poligonului respectiv.

Pentru a ajuta codarea acestei clase, am creat o clasa ajutătoare *ColoredPolygon* care reține coordonatele vârfurilor stocate într-un vector de *short*-uri folosind un obiect de tipul *StaticPointArray* și culoarea într-o valoare *integer*. De-asemenea în fiecare obiect de tipul *ColoredPolygon* se mai afla și un obiect de tipul *Path2D.Float* numit *path* care este folosit la optimizarea creării imaginii demonstrative utilizatorului.

Clasa *StaticPointArray* a fost creată pentru a spori viteza procesării datelor.

Clasele *ColoredPolygon* și *StaticPointArray* au fost notate ca fiind serializabile și obiectul ajutător *path* a fost notat ca fiind *transient* ceea ce înseamnă ca ele nu vor fi scrise la serializare.

Procesul de divizare în poligoane constă în căutarea zonelor de culori care sunt similare și continue. Mulțimea de poligoane descoperita nu este cea mai eficientă deoarece am ales sa folosesc metoda Greedy pentru descoperirea lor. Folosind metoda Greedy am scăzut dificultatea codului de a-l scrie. Din păcate procesul se rulează decât pe un singur thread, dar este posibilă divizarea imaginii în celule și apoi aplicarea procesului doar pe acele celule pentru fiecare thread. Procesul în sine nu este aleator, o imagine va da aceleași rezultate de fiecare data pentru un *threshold* anume.

Una dintre dificultățile codului a fost faptul ca perimetrele poligoanelor erau dificil de calculat, odată chiar încercând sa folosesc backtracking, dar am ajuns la un cod care găsește în timp liniar perimetrul unei mulțimi conectate de pixeli.

De fiecare dată când se începe un proces de divizare se inițializează o matrice de flag-uri cu valori dacă un pixel este vizitat sau nevizitat.

Atunci când se găsește un pixel nevizitat, se caută toți pixelii conectați de acesta și care suma modulelor diferențelor culorilor dintre pixeli nu depășește *threshold*-ul curent. Toți pixeli conectați sunt apoi considerați vizitați. Fiecare poligon descoperit este adăugat la un *LinkedList*.

După ce s-a făcut rost de lista cu toate triunghiurile, se construiesc string-urile care vor conține lista în formatele SVG și SVGZ. Am decis să construiesc aceste string-uri devreme pentru a afișa utilizatorului mărimea fișierului când este salvat în formatele SVG și SVGZ.

Implementarea clasei PolygonVectorizer

```
private StaticPointArray list;

private StaticPointArray workList;
```

Aceste liste sunt folosite pentru a scade din punctele redundante folosite la definirea unui poligon. Ele sunt inițializate în constructor având mărimea maximă egala cu aria imaginii, ceea ce elimină orice posibilitate de „overflow” dar sacrifică foarte mult spațiu de pe RAM.

```
private static final int DIR_X[] = new int[]{1,1,0,-1,-1,-1,0,1};

private static final int DIR_Y[] = new int[]{0,1,1,1,0,-1,-1,-1};
```

5	6	7
4		0
3	2	1

Ordinea offset-urilor

Atunci când se caută pixeli învecinați, în loc de a folosi opt condiții *if*, folosesc un *for* care iterează toate aceste valori pentru a calcula pozițiile vecinilor. Motivul de ce am ales primul vecin verificat să fie cel drept (1,0) este pentru a putea folosi cât mai mult posibil cache-ul procesorului.

Atunci când un procesor face un „request” pentru informații de pe RAM, acesta de obicei obține și date imediat învecinate și le stochează împreuna în cache. Cache-ul este semnificativ mai rapid decât memoria RAM, dar acesta este limitat ca și spațiu (exemplu : procesorul Intel i5 4460 are 6MB de cache). Dacă un cod accesează în mod consecutiv de la stânga la dreapta biți de informație de pe RAM, procesorul va folosi foarte des și corect cache-ul său. Dacă un cod accesează biți în mod aleator de pe RAM, cache-ul va fi folosit ineficient, și viteza de procesare a informațiilor va scădea semnificativ.

```
private char visitMatrix[];

private char workMatrix[];
```

Cele două matrice conțin una din patru valori numerice despre fiecare pixel în parte, și sunt folosite la descoperirea poligoanelor de culori continue. Matricea *workMatrix* stochează informațiile relevante doar la poligonul curent procesat, iar matricea *visitMatrix* stochează informații despre toate poligoanele descoperite în momentul procesării.

```
private LinkedList<ColoredPolygon> coloredPolygons = new LinkedList<>();
```

Aceasta doar ține o referință către ultima lista de poligoane care a fost generată de către un thread *Job*.

```
private void drawFunction(AbstractList<ColoredPolygon> coloredPolygonList){

    if(destImagePanel!=null || isInBenchmark) {
        Graphics2D g = destImage.createGraphics();
        for ( coloredPolygon c : coloredPolygonList) {
            g.setColor(new Color(c.color));
            g.fill(c.getPath());
        }
        if(!isInBenchmark) {
            destImagePanel.setImage(destImage);
        }
    }
}
```

Metoda aceasta este folosită pentru a afișa utilizatorului rezultatul aproximativ (și superficial) al vectorizării. Mai întâi se verifica dacă există referință către obiectul *ImagePanel*, sau dacă vectorizarea este făcută doar pentru un test de performanță.

Se iterează prin toată lista de obiecte *ColoredPolygon*, la fiecare pas setând culoarea poligonului, și umplerea ei folosind metoda *fill*. Dacă vectorizarea a fost făcută cu scopul măsurării performanței sale, nu se va afișa nimic utilizatorului.

```
protected void constructStringSVG()
```

Metoda creata pentru construirea string-urilor care va conține imaginea vectoriala în formatele SVG și SVGZ.

```
svgStringBuilder.setLength(0);
```

Deoarece exista o șansă foarte mare ca utilizatorul să încerce mai multe *threshold*-uri până când este mulțumit, se creează un obiect *StringBuilder* care este resetez la zero de fiecare dată când se începe scrierea formatelor SVG/SVGZ. Chiar dacă el este resetat, el își păstrează vechiul buffer folosit dinainte, deci nu va fi overhead în distrugerea și recrearea „string-ului” său intern.

```
svgStringBuilder.append(String.format("<svg xmlns='http://www.w3.org/2000/svg'
version='1.1' width='%d' height='%d'>", w, h));
```

Prima linie pe care o conține SVG-ul va descrie în ce format este scris, și ce lățime și înălțime va avea imaginea vectorizată.

```
for(ColoredPolygon c : coloredPolygons){

    StaticPointArray spa = c.pointArray;
    svgStringBuilder.append("<path d='M");
    int n = spa.size()-1;
    for(int i=0;i<n;i++){
        svgStringBuilder.append(spa.getX(i));
        svgStringBuilder.append(',');
        svgStringBuilder.append(spa.getY(i));
        svgStringBuilder.append('L');
    }
    svgStringBuilder.append(spa.getX(n));
    svgStringBuilder.append(',');
    svgStringBuilder.append(spa.getY(n));
    svgStringBuilder.append(String.format("Z' fill='#%06X'/>\n",c.color&0xffffff));
}
```

Am decis sa folosesc tag-ul *path* pentru al face mai natural de citit, și mai ușor de manipulat într-un editor de fișiere vectoriale precum Inkscape. Litera M are rolul de a muta „pensula” la acea poziție absolută, litera L are rolul de a trasa o linie la aceea poziție absolută iar litera Z are rolul de a închide cu o linie figura geometrică desenată. Un mic avantaj la *PolygonVectorizer* este faptul că toate coordonatele sunt numere întregi, ceea ce face construirea string-ului mai rapidă.

```
svgStringBuilder.append("</svg>");
```

La finalul fișierului, se adaugă tag-ul de sfârșit al formatului SVG.

```
constructStringSVGZ();
```

Metoda aceasta este apelată pentru a crea string-ul în format SVGZ.

Subclasa Job a clasei PolygonVectorizer

Clasa aceasta are rolul de a crea în mod relativ eficient lista de poligoane a procesului de vectorizare. Ea extinde clasa *JobThread* care conține un flag boolean *canceled*.

```
private LinkedList<ColoredPolygon> localList = new LinkedList<>();
```

Aici se va ține ca referință lista de poligoane la care lucrează thread-ul, dacă operația de vectorizare se termină cu succes, referința aceasta va fi copiată mai târziu în *coloredPolygons*.

```
if(visitMatrix==null || visitMatrix.length < h*w) {

    visitMatrix = new char[h*w];
}
if(workMatrix==null || workMatrix.length < h*w){
    workMatrix = new char[h*w];
}
```

Deoarece este foarte posibil ca utilizatorul sa reîncerce diferite poze a trebuit să creez o metoda de a recicla vectorii cei vechi.

```
Arrays.fill(visitMatrix,0,h*w,(char)0);

Arrays.fill(workMatrix,0,h*w,(char)0);
```

Ultimul pas al reciclării este curățarea conținuturilor lor. Funcția *Arrays.fill()* doar iterează vectorul intre cele doua valori 0 și *h*w* cu un simplu *for*. Este posibil ca prin folosință repetată ca JIT-ul să eficientizeze acele loop-uri.

Semnificația valorilor de vizitare:

* 0 : punctul nu a fost vizitat

* 1 : punctul a fost vizitat și este considerat punct intern și nu poate fi parte din perimetru

* 2 : punctul a fost vizitat și este posibil ca acesta să facă parte din perimetrul final

* 3 : punctul a fost vizitat și face parte din perimetrul final al poligonului verificat

```
for(y0=0;y0<h;y0++){

    for(x0=0;x0<w;x0++){
        pixel = y0*w+x0;
        if(canceled)return;
        if (visitMatrix[pixel] == 0) {
            ColoredPolygon coloredPolygon = findShape(x0,y0);
            localList.add(coloredPolygon);
        }
    }
    int k = aproxCompletedPixelCount.addAndGet(w);
    updateDetails(String.format("Progress : %.1f%%", 100.f * k / area));
}
```

Caut fiecare figură geometrică prin iterarea tuturor pixelilor din imagine. Atunci când o figură geometrica este descoperită, pixelii care îi aparțin acestuia sunt vizitați. Se va încerca căutarea unei figuri geometrice numai și numai dacă acest pixel nu a fost notat ca fiind vizitat.

```
if(canceled)return;

coloredPolygons = localList;
```

Se verifică flag-ul *canceled* și se copiază lista de poligoane în *coloredPolygons*.

```
Thread th = new Thread(() -> {

    exportSvgTime = System.currentTimeMillis();
    constructStringSVG();
    exportSvgTime = System.currentTimeMillis()-exportSvgTime;
    updateDetails(String.format("SVG:%s SVGZ:%s",
            Utility.aproximateDataSize(svgStringBuilder.length()),
            Utility.aproximateDataSize(svgzStringBuilder.length())));
});
th.start();
drawFunction(coloredPolygons);
try {
    th.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
setIsDone(true);
```

Se execută pe un thread separat construcția fișierelor și reîmprospătarea interfeței, iar pe thread-ul principal se creează imaginea vectorizată aproximativă. Funcția set *isDone()* anunță testul de performanță momentul când a terminat vectorizatorul.

```
private char getWorkPixel(int x,int y)
```

Returnează valoarea de vizitare a pixelului de pe coordonatele x și y din matricea *workMatrix*. Dacă valorile x și y nu sunt corecte se returnează 0.

```
private boolean isWorkPixelNotVisited(int x,int y)
```

Returnează dacă pixelul este valid și dacă este nevizitat.

```
private boolean isThereAnyEmptySpaces(int x0,int y0)
```

Se verifica dacă în jurul pixelului se afla cel puțin un pixel nevizitat, sau dacă se afla pe marginea imaginii.

```
private ColoredPolygon findShape(short x,short y)
```

Metoda aceasta returnează figura geometrică care reprezintă zona continuă de culori asemănătoare pe baza valorii *threshold*, care conține pixelul de pe coordonata (x,y) și este formată din pixeli care anterior nu au fost vizitați de către apelări precedente ale metodei *findShape*.

Următoarele paragrafe vor explica conținutul metodei *findShape.*

```
ColoredPolygon coloredPolygon = new ColoredPolygon();

int startColor = colorOrig(x,y);
int rTotal=0,gTotal=0,bTotal=0;
int count=0;
int currentColor;
```

Obiectul *coloredPolygon* reprezintă obiectul ce va fi adăugat. Dacă procesul nu a fost anulat el va conține informații relevante.

Pentru a cauta cât mai rapid poligonul, o să presupun că pixelul inițial conține media aproximativă a culorilor pe care trebuie să le caut. În *rTotal,gTotal* și *bTotal* stochez suma canalelor de culori pentru a calcula media adevărata a culorilor. Variabila *count* reprezintă numărul de pixeli care aparțin poligonului construit, iar *currentColor* este o variabila ajutătoare.

```
workMatrix[y*w+x] = 2;
```

Valoarea 2 reprezintă faptul că pixelul este vizitat și considerat perete, dar nu a fost încă adăugat la lista finală.

```
short x0=x,y0=y,x1,y1;

list.clearAll();
list.push((short) (x0 - 1), y0);
list.push(x0, (short) (y0-1));
list.push(x0, (short) (y0+1));
list.push((short) (x0+1),y0);
```

Deși obiectul *list* se numește așa , în următoarele linii de cod el va fi folosit ca o stivă. Lista este golită și se inserează toate punctele învecinate pixelului. Ele vor fi verificate dacă sunt valide pentru a face parte din interiorul poligonului sau din perimetrul poligonului.

```
rTotal += redOrig(x, y);

gTotal += greenOrig(x, y);
bTotal += blueOrig(x, y);
count++;
```

Se adaugă valorile culorilor ale punctului inițial și se incrementează variabila *count*.

```
short minX=x,maxX=x,minY=y,maxY=y;
```

Valorile acestea vor fi folosite mai jos pentru a descoperii cel mai nordic pixel care are valoarea de vizitări egala cu 2 (perete care nu este încă adăugat la lista de puncte).

```
while(!list.isEmpty()){

    if(canceled)return coloredPolygon;
    x0 = list.getLastX();
    y0 = list.getLastY();
    list.deleteLast();
```

Cât timp mai sunt puncte în liste, acestea vor fi verificate. De-asemenea se verifică dacă job-ul curent a fost anulat sau nu.

```
int index = y0*w+x0;

if(x0<0 || x0>=w || y0<0 || y0>=h)continue;
if(workMatrix[index]!=0)continue;
currentColor = colorOrig(x0,y0);
```

Variabila *index* are rol ajutător. Se verifică dacă pixelul acesta se află în imagine, și dacă da, apoi se verifică dacă a fost vizitat sau nu. După aceea se stochează culoarea curentă .

```
if(minX>x0)minX = x0;

if(minY>y0)minY = y0;
if(maxX<x0)maxX = x0;
if(maxY<y0)maxY = y0;
```

Se reactualizează valorile min și max.

```
if(x0==0 || x0==w-1 || y0==0 || y0==h-1 ||

        visitMatrix[index]!=0 ||
        Utility.manhattanDistance(startColor, currentColor)>threshold)
{
    workMatrix[index]=2;
}
```

Dacă pixelul se afla pe marginea imaginii, sau dacă a fost vizitat de un poligon creat anterior , sau dacă eșuează testul *threshold*, atunci acesta este considerat pixel pentru perimetru și valorile culorilor sale nu vor fi adăugate la media culorilor.

```
else

{
    workMatrix[index]=1;
    rTotal += Utility.red(currentColor);
    gTotal += Utility.green(currentColor);
    bTotal += Utility.blue(currentColor);
    count++;
    if(isWorkPixelNotVisited(x0,y0+1))
        list.push(x0, (short) (y0+1));
    if(isWorkPixelNotVisited(x0+1,y0))
        list.push((short) (x0+1),y0);
    if(isWorkPixelNotVisited(x0-1,y0))
        list.push((short) (x0-1),y0);
    if(isWorkPixelNotVisited(x0,y0-1))
        list.push(x0, (short) (y0-1));
}
```

Dacă pixelul nu a eșuat niciuna dintre testele precedente, atuncea acesta este un pixel interior poligonului căutat. Valorile culorilor sale sunt adunate, iar vecinii săi sunt adăugați la container-ul *list*.

```
y = minY;

x = minX;
while(getWorkPixel(x,y)!=2)x++;
```

Se caută primul pixel de linia *minY* (cel mai nordic pixel) care face parte din perimetru.

```
list.push(x,y);

workMatrix[y * w + x] = 3;
boolean done = false;
int dir=0,dir2;
```

Acel punct găsit mai devreme face cu siguranță parte din perimetrul adevărat, deci el va fi notat cu valoarea 3. El este adăugat în lista pentru a ajutat la găsirea următoarelor puncte în ordinea corectă.

```
while(!done) {

    if (canceled) return coloredPolygon;
    x1 = list.getLastX();
    y1 = list.getLastY();
    for (dir2 = dir; dir2 < 8 + dir; dir2++) {
        x0 = (short) (x1 + DIR_X[dir2 % 8]);
        y0 = (short) (y1 + DIR_Y[dir2 % 8]);
        if (x0 < 0 || x0 >= w || y0 < 0 || y0 >= h) continue;
        if (y0 == y && x0 == x) {
            done = true;
            break;
        } else if (workMatrix[y0 * w + x0] == 2) {
            if (isThereAnyEmptySpaces(x0, y0)) {
                workMatrix[y0 * w + x0] = 3;
                list.push(x0, y0);
                dir = (dir2 + 5) % 8;
                break;
            } else {
                workMatrix[y0 * w + x0] = 1;
                rTotal += redOrig(x0, y0);
                gTotal += greenOrig(x0, y0);
                bTotal += blueOrig(x0, y0);
                count++;
            }
        }
    }
}
```

De fiecare data când se repeta ciclul *while* se verifica toate cele opt punctele învecinate acestuia dacă sunt valide(in interiorul imaginii) și dacă este considerat perete nevalidat (cu valoarea 2). Din cauza felului cum sunt implementați vectorii *DIR_X* și *DIR_Y* și a algoritmului de parcurgere a suprafeței imaginii se va găsii întotdeauna cel puțin un punct care este corect.

Este posibil ca un punct care a fost notat ca fiind perete nevalidat să fie fals. El poate fi perete adevărat doar dacă are cel puțin un spațiu gol învecinat. Dacă este perete adevărat atunci acesta este adăugat la lista și se iese din ciclul *for;* altfel acel punct va fi notat ca fiind punct interior obișnuit și se vor adaugă valorile culorilor sale la sumele lor respective. Dacă unul dintre punctele învecinate este chiar punctul inițial, înseamna că lista a fost construită corect, și se poate întrerupe ciclul *while*.

Rolul variabilelor *dir* și *dir2* este de a alege poziția corecta din care să se înceapă căutarea, în așa fel încât primul punct valid este și cel corect, astfel algoritm-ul nu are nevoie sa folosească metoda backtrack-ing.

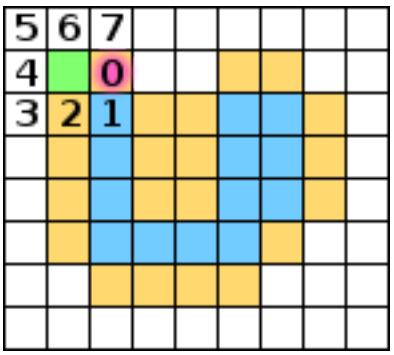


Figura 1

În figura 1 se poate vedea o figură geometrică simplă. Punctul cel verde simbolizează faptul că a fost notat cu valoarea de vizitare 3. Punctele cele portocalii au valoarea 2, cele albastre au valoarea 1, iar cele transparente au valoarea 0 (nu au fost vizitate).

Deoarece acel punct a fost selectat primul pentru crearea perimetrului, prima dată va încerca să caute următorul punct cu valoarea 2 pe poziția 0, care într-adevăr este acolo. Punctul cel nou va fi notat cu valoarea 3 (figura 2).

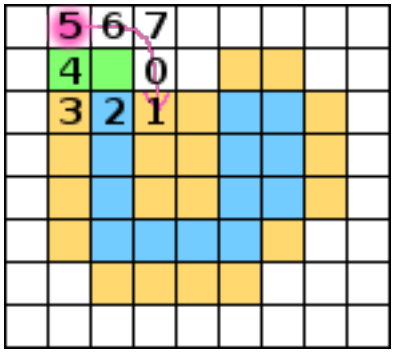


Figura 2

Deoarece punctul precedent a găsit acest punct pe poziția 0, căutarea noului punct va începe de la poziția (0+5)%8 => *5*. Pozițiile vor fi încercate în sensul acelor de ceasornic până când se găsește noul punct de pe poziția 1. Dacă algoritmul nu s-ar fi oprit pe aceea poziție, ar fi continuat spre 3 sau poate chiar 4, care a fost notat ca fiind punctul inițial.

În figura 3 căutarea următorului punct începe de la poziția (1+5)%8 => *6* și se oprește pe poziția 0. Tot în figură se poate vedea cum punctele de pe pozițiile 1 și 2, care deși au fost notate ca fiind potențiale puncte de perimetru, nu vor mai fi procesate.

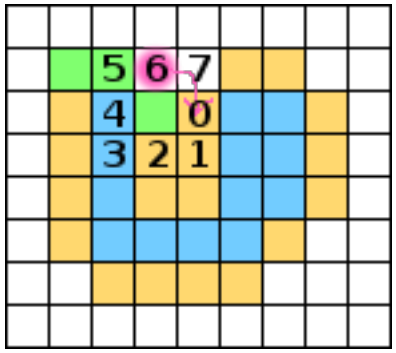


Figura 3

În figura 4 căutarea începe de la poziția 5 deoarece ultima direcție validă a fost pe poziția 0. Algoritmul va găsi pe poziția 7 următorul punct.

```
coloredPolygon.pointArray = list.cloneUpTo(list.size());
```

După ce s-a creat lista redusă , poligonul va primi o clonă a listei respective , dar cu mărimea maxima redusa la exact cât este necesar.

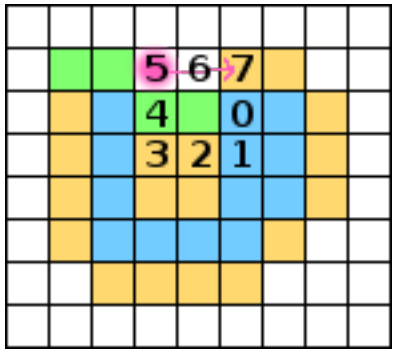


Figura 4

```
int index;

for(y0=minY;y0<=maxY;y0++)
    for(x0=minX;x0<=maxX;x0++) {
        if(canceled)return coloredPolygon;
        index = y0 * w + x0;
        if(workMatrix[index]==1||
(workMatrix[index]==3&&(x0==0||x0==w-1||y0==0||y==h-1))){
            visitMatrix[index] = workMatrix[index];
        }
        workMatrix[index]=0;
    }
```

Se copiază în *visitMatrix* valorile de vizitare care sunt relevante: valorile de 1 care reprezintă faptul că pixelul a fost în interiorul poligonului, și valorile de 3 care sunt pe marginea imaginii. Și în același timp se șterg valorile curente din *workMatrix*.

Ca să fie mai eficientă parcurgerea , se traversează doar zona cuprinsă de *minX,minY,maxX* și *maxY*.

```
return coloredPolygon;
```

La final se returnează poligonul generat.

4. Clase utilitare

Clasa Static Point Array

Pentru a mări viteza de procesare a perimetrelor la vectorizatorul de poligoane am creat o nouă clasă care să memoreze coordonatele punctelor perimetrelor.

Am denumit-o *Static Point Array* deoarece odată ce este creat acest obiect, el are la dispoziție un vector de mărimea *size* cu care să facă toate operațiile. Nu funcționează ca un *ArrayList* în care dacă se depășește spațiul disponibil să se realoce un vector nou.

Am ales să folosesc doi vectori de tipul *short* ca să memorez coordonatele, deoarece nu mă aștept ca vreun utilizator să folosească poze care au înălțimea sau lățimea mai mare de 32000.

Exista o variabilă cu numele *count* care memorează lungimea curentă a datelor relevante din vectori.

Descrierea metodelor :

```
public void push(short x,short y)
```

Împinge la sfârșitul vectorilor un punct nou. Metoda aruncă excepții dacă operația nu este posibilă. Operația are timp constant deoarece ea mereu introduce la finalul vectorilor.

```
public void deleteLast()
```

Șterge punctul final din vectori. Metoda aruncă excepții dacă operația nu este posibilă. Tot ce se face este să se decrementeze variabila *count*. Operația are timp constant deoarece ea mereu șterge de la finalul listei.

```
public void clearAll()
```

Se șterg toate punctele din vectori. Tot ce se face este să fie setată variabila *count* la 0. Operația are timp constant.

```
public boolean isEmpty()
```

Se returnează dacă *count* este egal cu 0.

```
public int size()
```

Deși numele pe care l-am ales pentru funcție este *size*, el defapt returnează *count*.

```
public short getX(int i)
```

```
public short getY(int i)
```

Funcții getter pentru puncte de pe anumiți indexi, a trebuit să fac două deoarece nu vroiam să fac un struct care să conțină ambele valori. Operația are timp constant deoarece se citește dintr-un vector.

```
public short getLastX()
```

```
public short getLastY()
```

Funcții getter specializate, citește ultima valoare din vectori.

```
public void setX(short i,short x)
```

```
public void setY(short i,short y)

public void setXY(int i,short x,short y)
```

Metode setter pentru puncte de pe anumite poziții. Nu se verifică dacă index-ul se află între 0 și *count*.

```
public StaticPointArray cloneUpTo(int s)
```

Metodă folosită pentru a crea un nou obiect *StaticPointArray* de mărime *s* care copiază din vectorii originali până la indexul *s*. Timpul de execuție este liniar.

```
public void delete(int i)
```

Metodă pentru a șterge un punct de la o anumită poziție. Din păcate timpul de execuție este liniar , din cauza vectorilor.

```
public void copyFrom(StaticPointArray spa)
```

Am creat metoda aceasta pentru a copia rapid valori de la un obiect *StaticPointArray* la altul. Timpul de copiere este liniar, și aruncă excepție dacă nu este suficient spațiu pentru a copia vectorii.

Clasa Utility

Deoarece majoritatea claselor folosesc mai multe funcții comune am decis să le mut pe toate într-o clasa utilitară.

```
public static int manhattanDistance(int c1,int c2){

        char b1 = (char)(c1 & 255);
        c1 >>= 8;
        char g1 = (char)(c1 & 255);
        c1 >>= 8;
        char r1 = (char)(c1 & 255);
        char b2 = (char)(c2 & 255);
        c2 >>= 8;
        char g2 = (char)(c2 & 255);
        c2 >>= 8;
        char r2 = (char)(c2 & 255);
        return abs(r1,r2)+abs(g1,g2)+abs(b1,b2);
}
```

Folosesc variabile de tipul int pentru a stoca culori, metoda aceasta este des întâlnită.

În Java variabilele de tipul int conțin 32 de biți de informație. Folosesc diferite regiuni pentru a defini culoarea care ma interesează. Primii 8 biți sunt de obicei folosiți pentru a descrie valoarea alpha a culorii. Următorii 8 sunt pentru cât roșu conține culoarea, apoi încă 8 pentru verde apoi ultimii 8 pentru albastru.

Pentru a accesa ultimii opt biți, se aplică operația binară AND folosind un număr care are ultimii 8 biți egali cu valoarea unu și restul biților cu zero. Acest număr este 255, sau 0xFF scris în baza hexazecimală.

Aplicând AND binar cu 255 , obțin ultimii opt biți dintr-o variabila int. Ultimii opt biți dintr-o variabilă int folosite pentru a stoca culori reprezintă culoarea albastră. După aceea trebuie sa împing la dreapta cu 8 biți întreaga variabila folosind operația Right Shift .

După aceea se pot repeta operațiile precedente pentru a obține biții pentru verde și pentru

roșu. Nu repet același lucru și pentru valoarea alpha deoarece nu am de gând să procesez și imagini care au transparență.

După ce obțin valorile culorilor de la prima variabila, repet același proces pentru a doua variabilă.

Am numit funcția *manhattenDistance* deoarece doar însumez valorile absolute ale diferențelor culorilor pentru a obține valoarea care o doresc. Sunt mulțumit de rezultate și nu doresc sa calculez distanța euclidiana .

Deoarece folosesc operații pe biți, întreaga funcție ar trebuii să fie destul de rapidă, ca și bonus, am creat o funcție specială care calculează modulul cu variabile char.

```
public static int abs(char a,char b){

	if(a<b) return b-a;
	return a-b;
}
```

Java dă eroare dacă încerc sa returnez char, deoarece operația de scădere într-o doua variabile char returnează o variabila int, și nu am vrut să mai creez overhead cu typecast-ul diferenței înapoi în char.

```
public static void writeTo(String s,OutputStream os) throws IOException {

	os.write(s.getBytes());
}
```

Funcție creată pentru convenienţă.

```
public static float distanceFromPointToLine(float x0,float y0,float x1,float y1,float
x2,float y2){
		float a = (y2-y1)*x0-(x2-x1)*y0+x2*y1-y2*x1;
		a = Math.abs(a);
		float b = (y2-y1)*(y2-y1)+(x2-x1)*(x2-x1);
		b = (float) Math.sqrt(b);
		return a/b;
	}
```

Aveam nevoie de o funcție rapidă și eficientă pentru calcularea distanței de la un punct la o linie. Linia este definită de punctele (x1,y1) și (x2,y2), iar punctul folosit pentru a afla distanța de la el la linie este reprezentat de (x0,y0). Funcția pare destul de rapidă și de eficienta, dar din păcate aceasta folosește o operație de calculare a radicalului.

Nu am pus protecții împotriva cazurilor speciale deoarece oricum funcția nu este niciodată folosită în cazuri precum : linie definită de două puncte identice, coordonatele se afla la distanțe extreme , etc.

```
public static float interpolate(float a, float b,float x)

{

    return a + x*(b-a);
}
```

Funcție creată deoarece am început sa scriu în mod repetat o bucată anume de cod și am vrut să o încapsulez.

```
public static String aproximateDataSize(int x) {
```

```
    if(x < 0)
        return "-1 B";
    if(x<1000)
        return x+" B";
    if(x<1000000)
        return String.format("%.2f KB",x/1024.f);
    if(x<1000000000)
        return String.format("%.2f MB",x/1048576.f);

    return String.format("%.2f GB",x/1073741824.f);
}
```

Cu cât am început să experimentez cu fișiere din ce în ce mai mari, pentru a măsura dimensiunea fișierelor în formatele SVG și SVGZ, a început să devină din ce în mai greu de observat rezultate. În versiunile mai vechi afișam dimensiunea în bytes, ceea ce era greu de citit (exemplu: SVG: 5012583B SVGZ: 331215B)

Acum afișez valoarea într-un format mai ușor de citit pentru utilizatorii obișnuiți. Exemplul anterior devine SVG: 4.78 MB SVGZ: 323 KB. Modificările acestea mici dar subtile ajut foarte mult la user-experience-ul aplicației.

Clasa Image Panel

Din prima zi în care am început sa scriu codul proiectului, mi-am adus aminte de un lucru destul de important despre Swing, aparent el nu conține din start un panou pentru afișarea unei imagini. Nu am găsit nici măcar opțiunea de a seta un panou obișnuit cu background imagine. Așa că am decis să implementez o clasă scurta și simplă.

Prin repozitorul Git, se poate vedea că aceasta clasa cândva avea mai multe funcționalități dar o să menționez doar pe cele care le folosesc.

```
private BufferedImage image;
```

Motivul de ce am ales sa folosesc *BufferedImage* este deoarece aveam acces ușor la pixelii stocați în imagine. În momentul acesta nu mai este nevoie de asta, dar mai demult *ImagePanel* avea cod care utiliza acești pixeli pentru a calcula și aproxima bound-urile din imagine.

```
protected void paintComponent(Graphics g) {

        g.drawImage(image, 0, 0, getWidth(), getHeight(), null);
}
public void setImage(BufferedImage img){
        image = img;
        repaint();
}
```

Deoarece clasa *ImagePanel* era una „custom” care extinde clasa *JPanel* a trebuit să definesc ce anume mai exact este desenat în el. Metoda *paintComponent* este chemata atunci când framework-ul Swing dorește desenarea obiectului, sau atunci când este chemată explicit metoda *repaint()*.

La prima vedere, metoda *paintComponent* pare că este definită greșit, dar ea funcționează corect. Odată este faptul că nu verific dacă valoarea obiectului *image* este egală sau nu cu null, sau nici măcar nu îl înconjur cu un block try/catch pentru a prinde erori, dar aparent din documentația metodei *drawImage*, acesta deja are grijă de astfel de situații și acționează în mod corespunzător.

A doua problemă este faptul că nu chem metoda *super.paintComponent(g)*. Aceasta ar fi o problemă dacă în loc *JPanel* aș fi extins o clasă mai complicată care avea deja ca default să deseneze ceva, dar *JPanel* deja nu desenează nimic deci nu are rost să chem aceea metodă.

Renunțând la folosirea metodei *super.paintComponent(g)* și la block-urile if/else și try/catch am mai redus puțin din overhead-ul de performanță.

Am făcut obiectul *image* să fie privat deoarece doream ca atunci când apelez funcția setter pentru ea să facă și un *repaint* al întregului obiect. Un lucru care mi s-a părut foarte interesant este faptul că obiectul *ImagePanel* funcționează corect chiar și când funcția *setImage* este apelată repetitiv, ea neavând nevoie de un obiect de sincronizare.

```
public boolean isOpaque() {return true;}
```

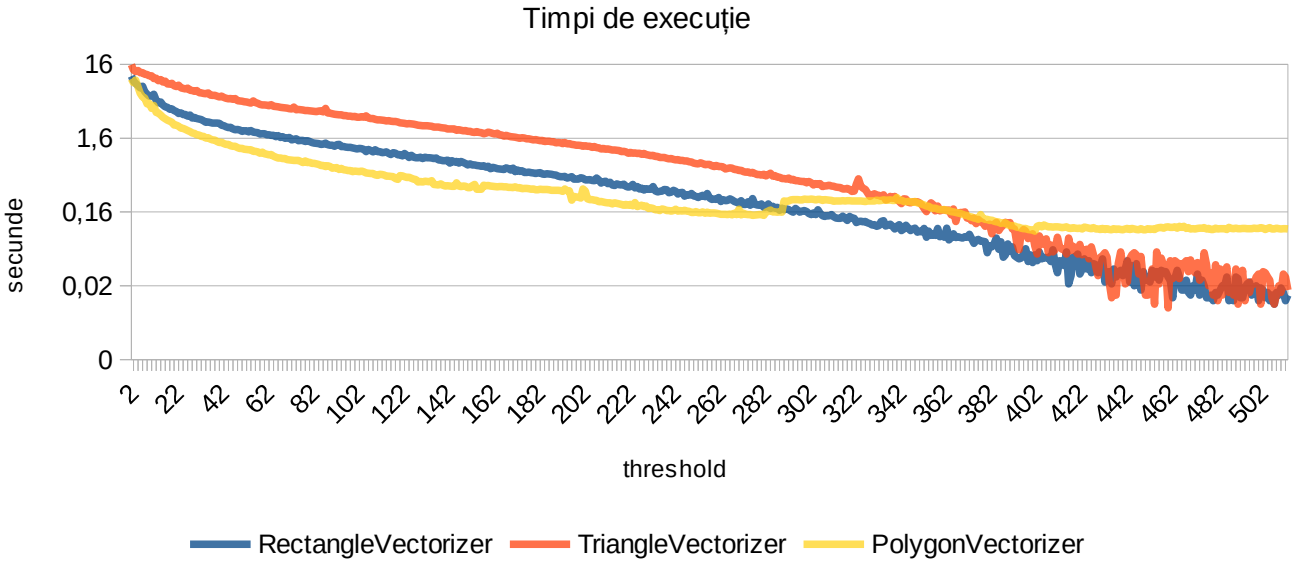
Metoda *isOpaque* este apelată de către framework-ul GUI pentru a ajuta la desenarea tuturor obiectelor. Dacă obiectul care extinde *JPanel* returnează true, atunci obiectul este desenat cu idea că este perfect opac, niciun pixel să fie transparent, deci nu se va încerca desenarea lucrurilor din spatele ei, dacă acestea exista. Dacă returnează false, framework-ul va încerca să deseneze și lucrurile din spatele obiectului, chiar dacă vor fi sau nu vizibile.

Am decis sa returnez true , pentru a reduce din overhead-ul de performanța .

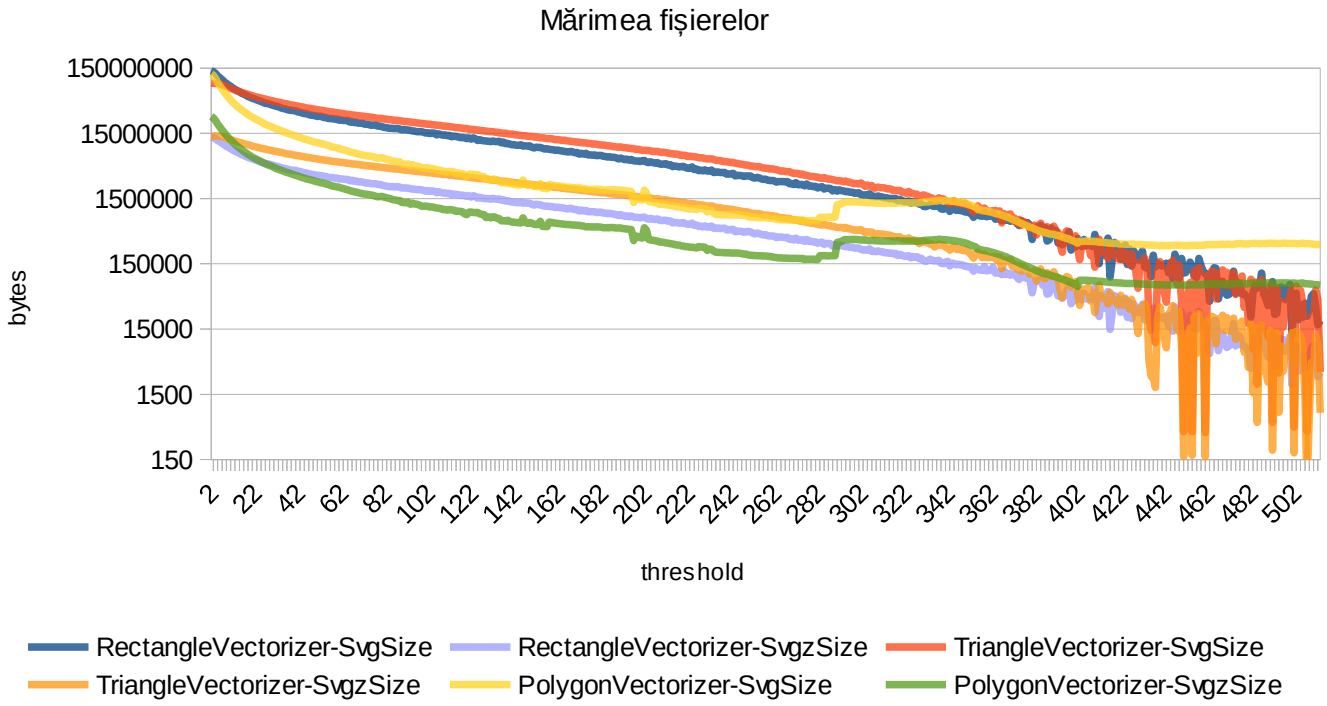
5. Teste de performanță

Acest capitol vă v-a prezenta câteva teste cu scopul de a măsura timpul de execuție și dimensiunea fișierelor la fiecare tip de vectorizare pentru diverse valori *threshold.*

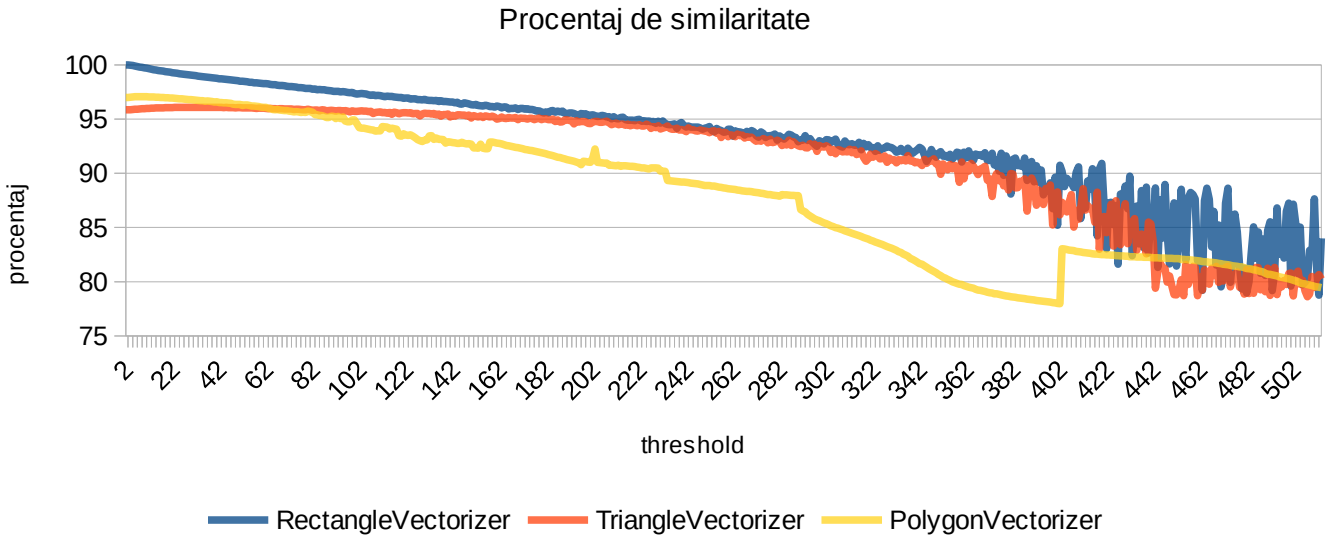
Testele au fost făcute folosind poza cea din dreapta, ea având dimensiunile 1920*1080, pe un calculator cu procesor Intel i5 4460, 20 GB de RAM DDR3 și Java SE Runtime Environment 8.



În diagrama *Timpi de execuție* se poate vedea cât da rapid lucrează fiecare vectorizator. *TriangleVectorizer* a fost ceea mai înceată, având timpul cel mai mare de 15,9 secunde. *PolygonVectorizer* este foarte rapid în comparație cu celelalte două, deși ele folosesc mai multe fire de execuție. Motivul de ce *RectangleVectorizer* și *TriangleVectorizer* începe să varieze foarte mult la valori *threshold* mai mari datorită naturi lor aleatoare.



În diagrama *Mărimea fișierelor* se poate observa faptul că în majoritatea cazurilor *PolygonVectorizer* generează fișierele cele mai mici. Motivul de ce *PolygonVectorizer* crește în dimensiune pe la valoarea 300 este deoarece el a generat un număr mic de poligoane cu foarte multe puncte de perimetru care nu a reușit sa le optimizeze.

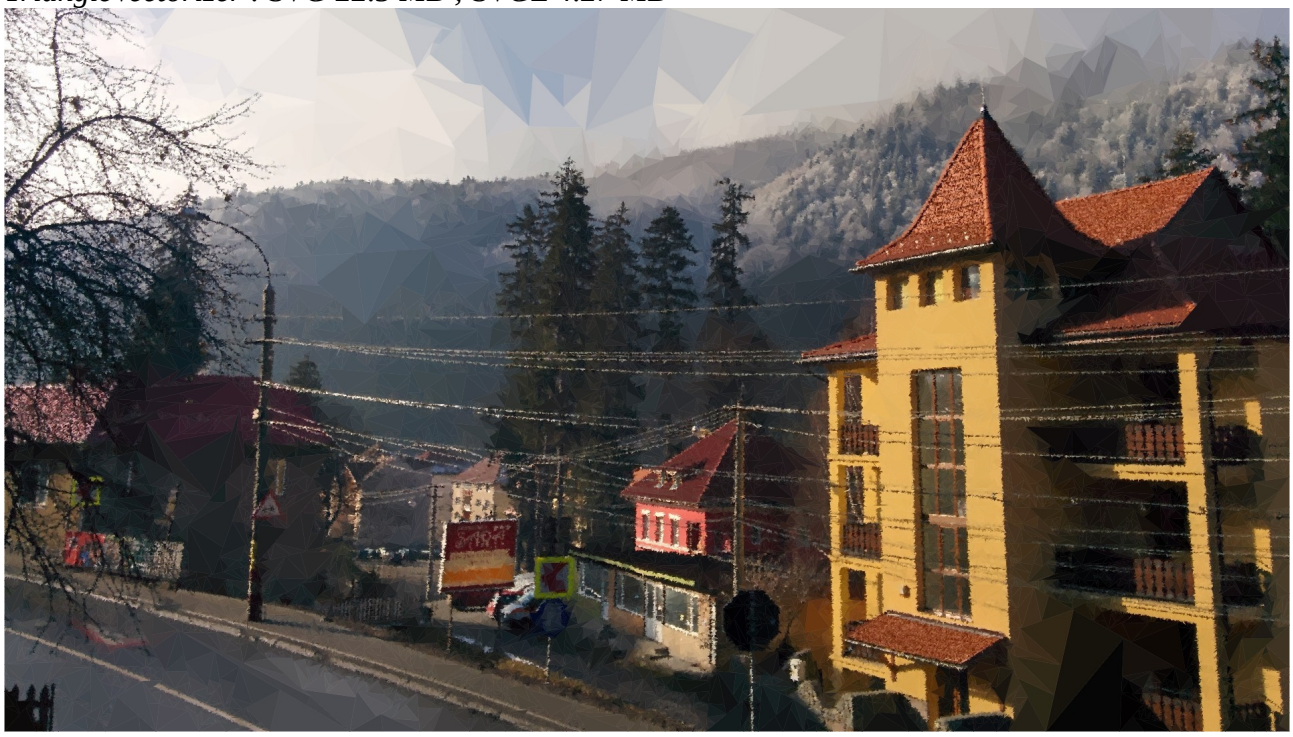


Deși *PolygonVectorizer* produce mai rapid fișiere mai mici decât celelalte două, în figura *Procentaj de similaritate* se vede cum acesta este semnificativ mai prost calitativ. Dar atunci când se folosește valori *threshold* mai mici de 50, se poate vedea ca acesta depășește *TriangleVectorizer*.

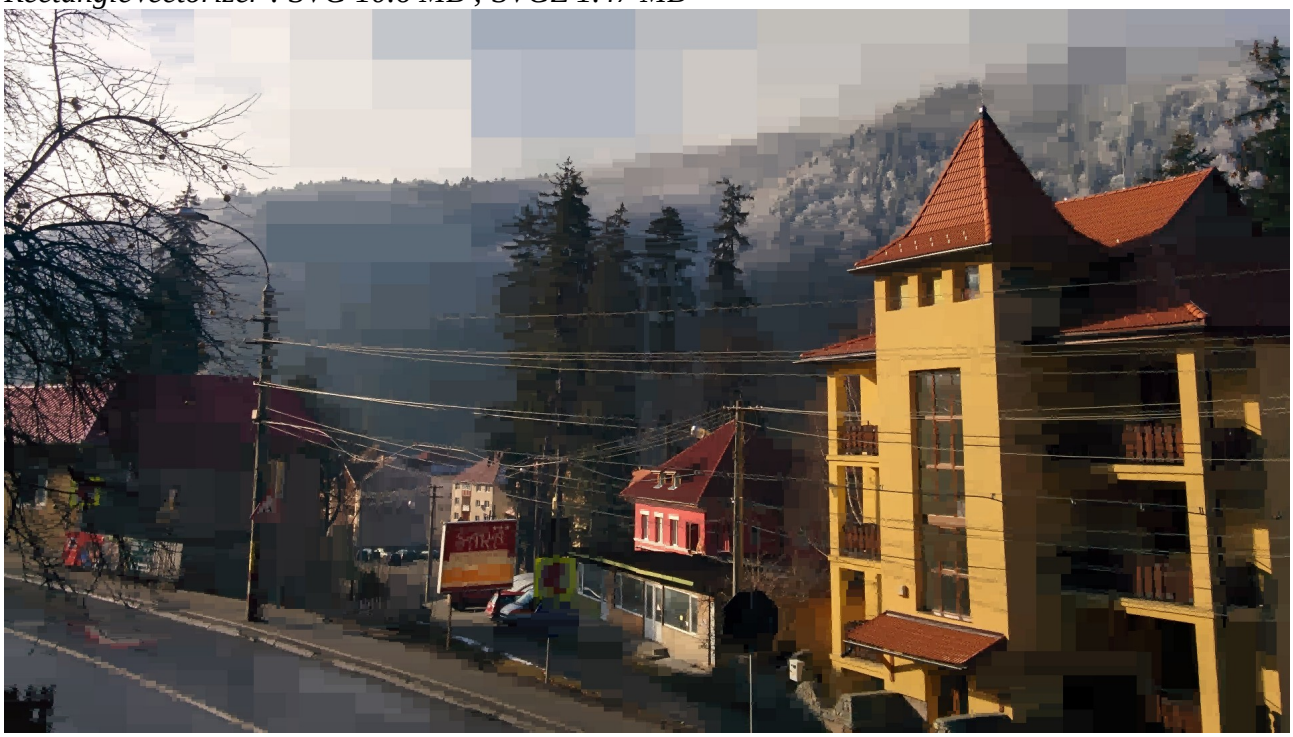
6. Exemple de poză vectorizată

*Poza originală : PNG 1920 * 1080 3.48 MB*

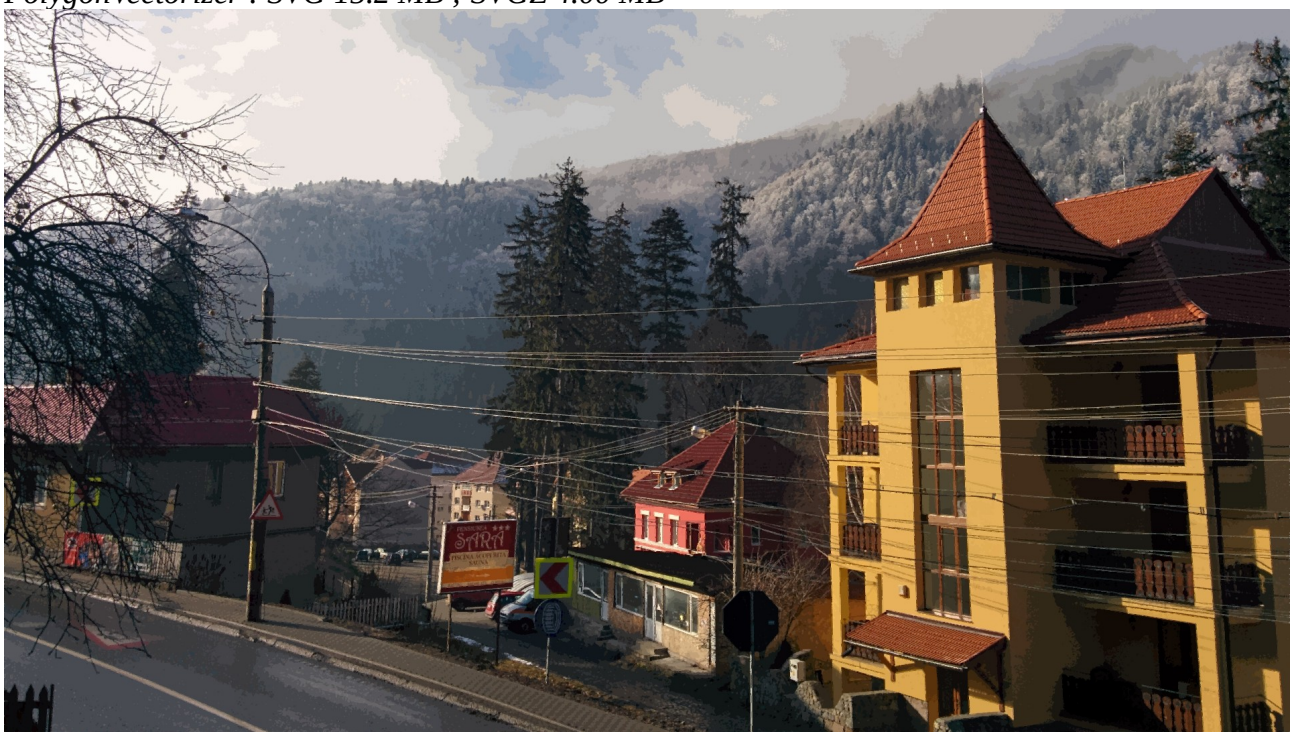
TriangleVectorizer : SVG 22.5 MB ; SVGZ 4.27 MB



RectangleVectorizer : SVG 10.6 MB ; SVGZ 1.47 MB



PolygonVectorizer : SVG 15.2 MB ; SVGZ 4.00 MB



7. Bibliografie

1. Redmond, K.C. & Smith, T.M. (2000). *From Whirlwind to MITRE, The R&D Story of the SAFE Air Defense Computer.* Obținut de pe pagina *http://books.google.com*

2. *Looking Back The TX-2 Computer and Sketchpad,* revista *Lincoln Laboratory Journal, Volume 19, Number 1, 2012 ,* I. Sutherland. Obținut de pe pagina https://www.ll.mit.edu/publications/labnotes/LookingBack_19_1.pdf

3. Foley, J. D. & van Dam, A. & Feiner, S.K. & Hughes, J.F. (1997). *Computer Graphics principles and practice second edition in C.* Obținut de pe pagina http://books.google.com

4. Chacon S. & Straub B. *Pro Git, Everything you need to know about Git.* Obținut de pe pagina http://books.google.com

5. Tanasă Ș. & Andrei Ș. & Olaru C. (2011). *Java de la 0 la expert – ediția a II-a rev. Iași, Editura POLIROM*