

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.....	3
Motivația.....	3
Definiții și scurt istoric.....	3
Comparație: Grafica vectorială VS Grafica raster.....	5
2.Descrierea facilităților folosite.....	6
Despre limbajul Java.....	6
Despre IDE-uri.....	7
Despre IntelliJ IDEA.....	7
Despre GitHub și Git.....	8
Despre Scalable Vector Graphics (SVG).....	9
3.Despre implementarea lucrării.....	11
Descrierea clasei BaseVectorizer.....	12
Implementarea metodelor startJob și cancelLastJob.....	14
Descrierea clasei RectangleVectorizer.....	15
Implementarea clasei RectangleVectorizer.....	16
Subclasa Job a clasei RectangleVectorizer.....	17
Descrierea clasei TriangleVectorizer.....	19
Implementarea clasei TriangleVectorizer.....	20
Subclasa Job a clasei TriangleVectorizer.....	23
Descrierea clasei PolygonVectorizer.....	29
Implementarea clasei PolygonVectorizer.....	29
Subclasa Job a clasei PolygonVectorizer.....	31
4.Clase utilitare.....	35
Clasa Static Point Array.....	35
Clasa Utility.....	37
Clasa Image Panel.....	39
5.Teste de performanță.....	41
Teste pentru Triangle Vectorizer.....	41
Teste pentru Square Vectorizer.....	42
Teste pentru Polygon Vectorizer.....	44
6.Exemple de poze vectorizate.....	45
7.Bibliografie.....	47

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 să folosească imagini raster obișnuite.

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

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 să 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 a-l transforma în SVG) sau EPS.GZ, dar deoarece este Shareware, de fiecare dată când un utilizator încearcă să 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 să 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 evidenția diferența față 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 curbilor 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 suprafață 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).

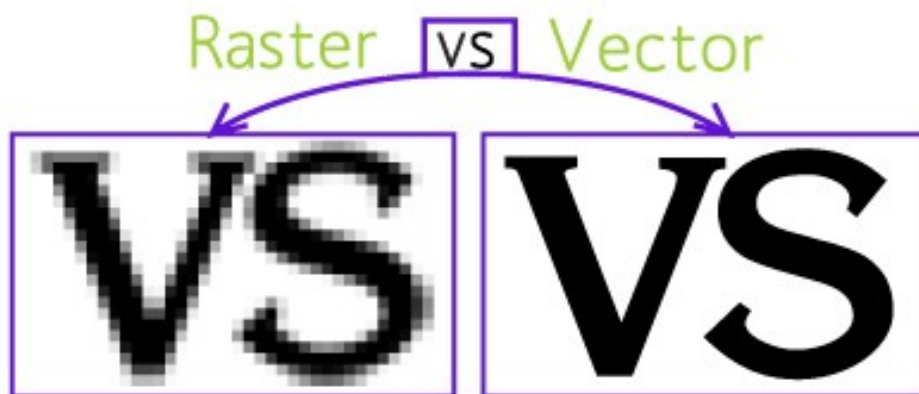


Figura 1.1 : http://vector-conversions.com/vectorizing/raster_vs_vector.html

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

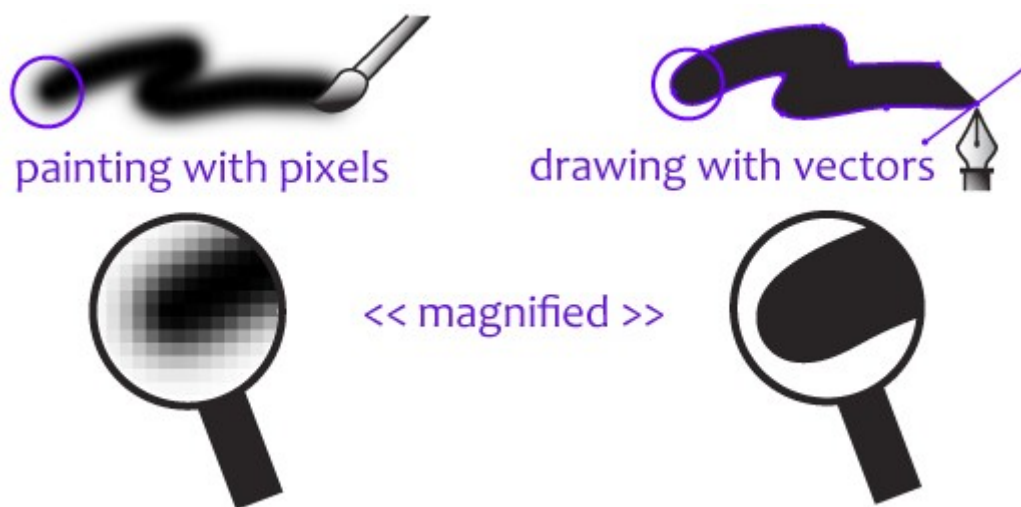


Figura 1.2 : http://vector-conversions.com/vectorizing/raster_vs_vector.html

	Avantaje	Dezavantaje
Vector	<ul style="list-style-type: none">* 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	<ul style="list-style-type: none">* Detalii limitate : figurile geometrice nu sunt practice pentru imagini complexe, chiar dacă acestea pot conține gradient, 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 precisă : oricare dintre pixeli poate fi modificat independent ; o persoană poate să editeze poza respectivă cu ușurință	* Blurată atunci când este mărită : deoarece există un număr finit de pixeli, calculatorul va încerca să umple spațiul dintre pixeli cu interpolări 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 există formate precum jpg și png care ajută 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 concurrent, 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) și IcedTea-Web (un plugin pentru aplicațiile de browser – applet-uri).

Deși limbajul Java are mai puține facilități low-level, există JNI (Java Native Interface), un framework de programare care permite codului scris în Java și care rulează într-o Mașina Virtuală 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-ului 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 NetBeans, IntelliJ IDEA și Eclipse; altele nu, cum ar fi SharpDevelop și 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 unificată 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 a 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 față 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 facilități nouă și apoi reunirea acestei ramuri la ramura principală pentru ca întregul proiect original să conțină această nouă facilități. De-asemena 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 să 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-asemena 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” dezvoltat 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 eficient. De-asemena 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 suportă 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.

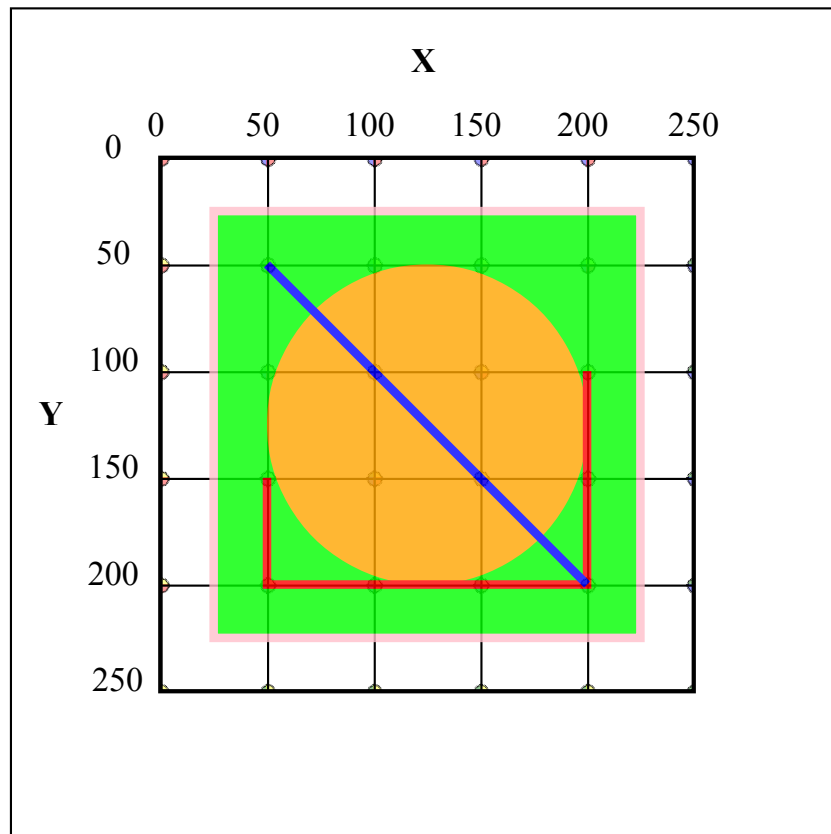
Imaginile SVG pot fi dinamice și interactive. Modificările pe bază de timp asupra obiectelor

pot fi descrise cu SMIL (Synchronized Multimedia Integration Language) sau JavaScript. Deși W3C recomanda SMIL, Google Chrome l-a deprecia 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 acum 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 față 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, precum texte, căi și figuri pot fi folosite pentru a defini 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 originală 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 aceleiași figuri geometrice. Astfel dacă *threshold* are o valoare mai mică atunci există 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 clasă 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 poată 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 compatibilă 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 întreagă și partea fracționară ar fi fost separate de o virgulă, atunci 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 formate : 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 față 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. Obiectul *gzos* este creat mai târziu atunci când este nevoie de el.

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 diferit 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 similară la prima vedere.

Se folosește tot obiectul *lastJob* pentru a asigura siguranța datelor. Se așteaptă oprirea execuției ultimului job curent. Și se apelează funcția *setIsDone(true)* pentru a anunța benchmark-ul că 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. Pentru a spori siguranța, clasa *RectangleFragment* conține o metodă *isValid()* care returnează *true* dacă marginile sunt valide.

Procesul de divizare în dreptunghiuri este recursiv, încercând prima dată un dreptunghi care ocupa 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 două valori *midX* și *midY* ambele aflându-se între o pătrime și trei pătrimi din lungimea și lățimea dreptunghiului original, astfel lăsând posibilitatea utilizatorului să reîncerce vectorizarea până când este mulțumit. Din cauza naturii aleatoare, este foarte posibil ca mărimea fișierului generat după o imagine sursă 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 aveau aspect de „caiet de matematică”; 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 această medie aproximativă, dreptunghiul este respins. În timp ce se iterează toți pixelii, se

însunează culorile lor pentru a calcula media adevărată a culorilor. Dacă un dreptunghi este respins acesta este tăiat în patru și se va încerca validarea acestora. Dacă un dreptunghi este valid, media adevărată a culorilor din acel dreptunghi este stocată 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 recursive 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 depuse în *LinkedList*-ul respectiv sunt acum depuse de către thread-ul principal într-un *ArrayList* de *SquareFragments*. Am ales *ArrayList* pentru performanța sporită 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. Deasemenea se creează un nou *BufferedImage* în care este randat o aproximație a imaginii vectoriale și apoi afișat utilizatorului.

Implementarea clasei *RectangleVectorizer*

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

Este nevoie de o valoare aleatoare pentru ca paternele care sunt generate să nu arate ca un desen „tehnic”. Fiind aleator există o șansă ca dreptunghiurile să se aranjeze în paterne mai organice.

```
private ArrayList<SquareFragment> lastSavedSquareList;
```

Aceasta doar ține o referință către ultima listă de dreptunghiuri care a fost generată de către un *Job*. Este posibil să renunț la această implementare dacă nu îi găsesc folos, pentru că lista de dreptunghiuri este totuși considerabilă ca și mărime.

Funcțiile *startJob()* și *cancelLastJob()* sunt identice cu cele din *TriangleVectorizer*.

```
public void drawFunction(ArrayList<SquareFragment> list) {  
  
    if(destImage!=null || isInBenchmark) {  
        Graphics2D g = destImage.createGraphics();  
        for (SquareFragment 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 foarte similara cu functia *drawTriangles* din *TriangleVectorizer*. Am folosit metoda *fillRect* în loc de *fill* deoarece aceasta este specializata pentru a desena dreptunghiuri deci exista o șansa foarte mare și fie mai rapidă. Motivul de ce am adaugat 1 la latime și lungime este deoarece metoda scade cu 1 aceste valori înainte sa deseneze.

```
protected void constructStringSVG()
```

Metoda creata pentru construirea string-urilor care va conține imaginea vectoriala în formatele SVG și SVGZ. Ea este foarte similara cu ceea din *TriangleVectorizer*.

Mai jos voi explica singura diferența.

```
for (SquareFragment sf : lastSavedSquareList) {  
  
    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 fisiere vectoriale precum Inkscape. Dacă aş folosi un tag mai compact precum *path* atunci dimensiunea fisierelor în format SVG ar scadea foarte mult dar s-ar pierde intuitivitatea.

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

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<SquareFragment> fragList = new ArrayList<>();
```

Aici va tine ca referinta lista de dreptunghiuri la care lucrează thread-ul, dacă operația de vectorizare se termina cu succes , referinta aceasta va fi copiata mai târziu în *lastSavedSquareList*.

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

```
private void splitSquareFragmentInFour(SquareFragment s, SquareFragment  
s1, SquareFragment s2, SquareFragment s3, SquareFragment 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 doua valori de pe lungimea și latimea dreptunghiului mare. Aceasta o calculez ca fiind interpolarea dintre valorile stânga cu dreapta și sus cu jos și o valoarea aleatoare aleasa uniform din mulțimea [0.25,0.75].

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

```
public void run()
```

În următoarele paragrafe voi explica continutul metodei run.

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

```
SquareFragment squareFragment = new SquareFragment((short)0, (short) (w - 1),  
(short) 0, (short)(h - 1), -1);  
startTime = System.currentTimeMillis();  
SquareFragment s1 = new SquareFragment();  
SquareFragment s2 = new SquareFragment();  
SquareFragment s3 = new SquareFragment();  
SquareFragment s4 = new SquareFragment();  
splitSquareFragmentInFour(squareFragment, s1, s2, s3, s4);
```

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

Se genereaza cele patru dreptunghiuri initiale generate randomizat de către *splitSquareFragmentInFour()*.

După aceea se creaza 3 thread-uri muncitoare și 3 containere de SquareFragments asemenator ca în *TriangleVectorizer*. Acele thread-uri și împreuna cu thread-ul principal lucrează pe cele 4 fragmente generate anterior pentru a aduna întreaga lista de dreptunghiuri care alcatuieste poza.

Pe parcursul adăugării listelor de fragmente din thread-uri , mai verific boolean-ul *canceled* pentru a fi sigur ca userul dorește sau nu sa înceapă o noua vectorizare.

Acum ca s-au colectat toate dreptunghiurile într-o lista comuna, se seteaza lista de dreptunghiuri a intregului obiect *SquareVectorizer* cu aceasta.

După aceea , execut pe un thread separat construirea imaginilor în format SVG și SVGZ , iar pe linie de execuție principala creez imaginea de prezentat userului. Functia set isDone() anunța testul benchmark momentul când a terminat vectorizatorul treaba.

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

Aceasta metoda cauta în mod recursiv toate dreptunghiurile care se afla în obiectul *s* și le depoziteaza 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 data calculez media culorilor varfurilor dreptunghiului pentru a aproxima culoare spre care trebuie sa tinda restul dreptunghiului. Cu cât dreptunghiul este mai mic cu atât aproximarea va fi mai precisa.

```

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 initializez un valoarea boolean *fail* cu scopul de a monitoriza dacă dreptunghiul curent conține culori prea distincte fata de media calculata mai devreme.

Pentru fiecare pixel în parte adun valorile culorilor lor în sumele lor respective și verific dacă diferența dintre culoarea ei și a valorii *avgColor* este mai mica sau egal cu cea a lui *threshold*, altfel procesul se va întrerupe atribuindui boolean-ul *fail* valoarea false.

Dacă după ce s-au iterat toate punctele variabila *fail* a rămas falsa , înseamnă ca acest dreptunghi este bun și va fi adăugat în lista *localFragList* după ce i se atribui campului *color* al acestuia media corecta a culorilor bazate pe sumele *rTotal*, *gTotal* și *bTotal*.

Dacă valoarea lui *fail* este true , atunci dreptunghiul curent este spart în patru folosind metoda *splitSquareFragmentInFour* și se verifica în mod recursiv cele patru noi dreptunghiuri dacă sunt sau nou potrivite pentru ariile lor respective.

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-aseamana 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 lungă a triunghiului. Linia care va fragmenta triunghiul în două este trasă de la colțul opus al laturii mari la un punct interpolat între cele două puncte ale laturii mari folosind valoarea r . Din cauza naturii aleatoare, este foarte posibil ca mărimea fișierului generat după o imagine sursă și un *threshold* anume să difere considerabil. Deoarece r este ales aleator, las posibilitatea utilizatorului să reînceze 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 aproximativă și pixelul verificat este mai mare decât valoarea *threshold*, atunci acest triunghi este respins. Dacă un triunghi este respins acesta este tăiat în două și se va încerca validarea acestora. Este posibil ca din cauza erorilor de calcul ca un triunghi să ajungă să aibă aria mai mică 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 stocată 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 două 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 sporită 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 verifică 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 vectorială 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 resetat 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.

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

Din cauza erorilor de calcul cu valori *float* încep să apară mici spații între 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 ajută la ascunderea liniilor albe dintre triunghiuri, dar nu rezolvă perfect situația. Dacă valoarea aceea de 0,5 este crescută 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&0xffffffff,
        t.color&0xffffffff));
}

```

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 scurtă opțiune la dispoziție. A trebuit să am o precauție în plus la definirea culorilor, așa ca păstrez numai biți care sigur au numai valori pentru roșu, verde și albastru. Operația aceasta de filtrare a biților o realizez prin AND binar cu numărul *0xffffffff*.

```

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.

```

try {

    if (gzos == null)
        gzos = new GZIPOutputStream(baos, true);
    baos.reset();
    gzos.write(svgStringBuilder.toString().getBytes());
    gzos.flush();
}

```

```

        svgzStringBuilder.setLength(0);
        svgzStringBuilder.append(baos.toString());
    } catch (IOException e) {
        e.printStackTrace();
    }

```

Obiectul *gzos* este de tipul *GZipOutputStream*, iar obiectul *baos* este de tipul *ByteArrayOutputStream*. Obiectul *baos* a fost inițializat cu un buffer de 2000000 de bytes, ceea ce ar trebui să fie suficient pentru start, iar obiect-ul *gzos* este inițializat aici dacă nu a mai fost creat până acum.

Obiectul *gzos* funcționează folosindu-se de încă un *OutputStream* deși el este la rândul lui un *OutputStream*.

Resetarea obiectului *baos* are rolul de a reseta counter-ul sau intern, buffer-ul intern nu este deconstruit și reconstruit.

Sunt puțin îngrijorat de linia *gzos.write()*, deoarece în acel punct se generează un obiect *String* destul de mare ceea ce ar putea deranja performanța aplicației, dar se pare că JIT-ul se descurcă destul de bine. După ce *string*-ul a fost scris în *gzos*, apelez explicit metoda *flush()* pentru a mă asigura că procesul de comprimare s-a efectuat cu succes.

Conținutul vechi al lui *svgzStringBuilder* este resetat, și apoi conținutul obiectului *baos* este scris în el.

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 copiată mai târziu în *lastSavedTriangleList*.

Am ales să 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 verifică 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 să 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 performanță 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 listă. 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 durată 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 (x_0, y_0) la (x_1, y_1) . Dacă punctul acesta de intersecție se afla în segmentul $(x_0, y_0)(x_1, y_1)$, coordonata *x* a acestui va fi memorată 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 acum.

```
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 $(x_1, y_1)(x_2, y_2)$. 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 acum 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 să iasă din segmentul [xMin,xMax].

```

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 să 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 că 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 un pixel, procesul de vectorizare în acel triunghi este oprit. Altfel se calculează media adevărată a culorilor.

```

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

    triangle.color = 0xff000000 | (rTotal<<16) | (gTotal<<8) | bTotal;
    triangles.add(triangle);
    int x = approxCompletedPixelCount.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 trebui 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.

```

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 să 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 dată pentru un *threshold* anume.

Una dintre dificultățile codului a fost faptul că perimetrele poligoanelor erau dificil de calculat, odată chiar încercând să 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 scădea din punctele redundante folosite la definirea unui poligon. Ele sunt inițializate în constructor având mărimea maxima egala cu aria imaginii, ceea ce elimina 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};
```

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 împreună în cache. Cache-ul este semnificativ mai rapid decât memoria RAM, dar aceasta este limitată (exemplu: procesorul Intel i5 4460 are 6MB de cache). Dacă un cod accesează în mod consecutiv 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. Mai multe detalii despre cele patru valori vor fi prezentate mai jos.

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

Aceasta doar ține o referință către ultima listă de poligoane care a fost generată de către un Job. Este posibil să renunț la această implementare dacă nu îi găsesc folos, pentru că lista de poligoane este totuși considerabilă ca și mărime.

Funcțiile *startJob()* și *cancelLastJob()* sunt identice cu cele din *TriangleVectorizer*.

```
private void drawFunction(LinkedList<ColoredPolygon> coloredPolygonList)
```

Metoda aceasta este foarte similară cu funcția *drawTriangles* din *TriangleVectorizer*.

```
protected void constructStringSVG()
```

Metoda creată pentru construirea string-urilor care va conține imaginea vectorială în formatele SVG și SVGZ. Ea este foarte similară cu cea din *TriangleVectorizer*.

Mai jos voi explica singura diferență.

```
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 & 0xffffffff));
}

```

Am decis sa folosesc tag-ul *path* pentru al face mai natural de citit , și manipulat într-un editor de fisiere vectoriale precum Inkscape. Litera M are rolul de a muta „pensula” la aceea 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 geometrica desenata. Un mic avantaj la PolygonVectorizer este faptul ca toate coordonatele sunt numere întregi , ceea ce face construirea string-ului mai rapidă.

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 tine ca referința lista de poligoane la care lucrează thread-ul, dacă operația de vectorizare se termina 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 sa creez o metoda de a recicla array-urile cele 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. Functia *Arrays.fill()* doar itereaza array-ul între cele doua valori 0 și h*w cu un simplu *for*. Este posibil ca prin folosință repetata ca JIT-ul sa eficientizeze acele loop-uri.

```

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 = approxCompletedPixelCount.addAndGet(w);
    updateDetails(String.format("Progress : %.1f%%", 100.f * k / area));
}

```

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

Următoarele linii de cod sunt similare cu cele din TriangleVectorizer. Lista locala de

poligoane este copiata la lista principala *coloredPolygons*, iar apoi folosind thread-ul principal și unul secundar se afișează imaginea rezultat aproximativă și se construiesc fișierele în formatele SVG și SVGZ. Functia `set isDone()` anunța testul benchmark momentul când a terminat vectorizatorul treaba.

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

Returnează valoarea de vizitare dacă coordonatele x și y aparțin imaginii pe care se lucrează.

```
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 returneaza figura geometrică care reprezintă zona continuă de culori asemănătoare pe baza valorii `threshold`, care conține pixelul de pe coordonata (x,y) și este formata din pixeli care anterior nu au fost vizitați de către apelări precedente ale 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 sa le caut. În *rTotal*, *gTotal* și *bTotal* stochez suma canalelor de culori pentru a calcula media adevărată 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 ca pixelul este vizitat , este 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 resetată și se inserează toate punctele învecinate pixelului pentru a se verifica 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 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 , deasemenea 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, dacă acel pixel a fost sau nu vizitat. 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 nu 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 vecini săi sunt adăugați la container-ul *list*.

```
y = minY;
```

```
x = minX;
```

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

Se caută primul pixel de linia *minY* 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 siguranța parte din perimetru , deci el va fi notat cu valoarea 3. El este adăugat în lista pentru a ajuta 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 verifică toate cele opt punctele învecinate acestuia dacă sunt valide (în interiorul imaginii) și dacă este considerat perete nevalidat (cu valoarea 2). Din cauza felului cum sunt implementate array-urile *DIR_X* și *DIR_Y* și a algoritmului de parcurgere a suprafeței imaginii se va găsi 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 un punct interior obișnuit și se vor adăuga valorile culorilor sale la sumele lor respective. Dacă unul dintre punctele învecinate este chiar punctul inițial, înseamnă ca lista a fost construită corect, și se poate întrerupe ciclul *while*.

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

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

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

```
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 utile

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 variabila 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 de fapt 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 puncte. 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. În momentul acesta 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 contin 32 de biti 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 biti, 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 biti dintr-o variabila int folosite pentru a stoca culor reprezintă culoarea albastră. După aceea trebuie sa împing la dreapta cu 8 biti întreaga variabila folosind operația Right Shift .

După aceea se pot repeta operatiile precedente pentru a obtine bitii pentru verde și pentru roșu. Nu repet acelesi lucru și pentru valoarea alpha deoarece nu am de gând sa procesez și imagini care au transparenta.

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

Am numit functia manhattanDistance deoarece doar insumez valorile absolute ale diferentelor culorilor pentru a obtine valoarea care o doresc. Sunt mulțumit de rezultate și nu doresc

sa calculez distanța euclidiană .

Deoarece folosesc operatii pe biti , întreaga funcție ar trebui să fie destul de rapid, 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 da eroare dacă încerc sa returnez char, deoarece operația de scădere într-o doua variabile char returneaza o variabila int , și nu am vrut să mai creez overhead cu typecast-ul diferentei înapoi în char.

```
public static void writeTo(String s, OutputStream os) throws IOException {  
    os.write(s.getBytes());  
}
```

Funcție creata doar de dragul convenientei.

```
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 eficiența pentru calcularea distantei de la un punct la o linie. Linia este definita de punctele (x1,y1) și (x2,y2) , iar punctul folosit pentru a afla distanța de la el la linie este reprezentat de (x0,y0) . Functia pare destul de rapidă și de eficienta, dar din păcate aceasta folosește o operație de calculare a radicalului.

Nu am pus protectii împotriva cazurilor speciale deoarece oricum functia nu este niciodată folosită în cazuri precum : linie definita de doua puncte identice , coordonatele se afla la distance extreme , etc.

```
public static float interpolate(float a, float b, float x)  
{  
    return a + x*(b-a);  
}
```

Funcție creata deoarece am început sa scriu în mod repetat o bucata de cod și am vrut sa o incapsulez.

```
public static String approximateDataSize(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 sa experimentez cu fisiere din ce în ce mai mari pentru a măsura dimensiunea fișierelor în formatele SVG și SVGZ a început sa devină din ce în ce mai greu de observa rezultate. În versiunile mai vechi afisam dimensiunea în bytes, ceea ce era greu de citit (exemplu: SVG: 5012583B SVGZ: 331215B)

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

Clasa Image Panel

Din prima zi în care am început sa scriu codul proiectului , mi-am readus aminte de un lucru destul de important despre Swing , aparent el nu conține din start un panou pentru afisarea unei imagini. Nu am găsit nici măcar optiunea de a seta un panou obișnuit cu background imagine. Așa ca am decis sa implementez o clasa scurta și simpla.

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

```
private BufferedImage image;
```

Motivul de ce am ales sa folosesc BufferedImage este deoarece aveam acces ușor la pixelii stocati 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 sa 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 chemat explicit functia repaint() .

La prima vedere , metoda paintComponent pare ca este greșită , dar ea funcționează corect. Odată este faptul ca nu verific dacă valoarea obiectului image este egala sau nu cu null, sau nici măcar nu îl inconjurez cu un block try/catch pentru a prinde erori , dar aparent din documentatia metodei drawImage , acesta deja are grija de astfel de situații și actioneaza în mod corespunzător.

A doua problema este faptul ca nu chem metoda super.paintComponent(g) . Aceasta ar fi o problemă dacă în loc JPanel aş fi extins o clasa mai complicata care avea deja ca default sa

deseneze ceva , dar JPanel deja nu deseneaza mai nimic deci nu are rost sa chem aceea metoda.

Renuntand la folosirea metodei `super.paintComponent(g)` și a block-urilor `if/else` și `try/catch` am mai redus puțin din overhead-ul de performanța.

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.

Metodele sunt rapide, scurte și ușor de citit.

```
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 ideea că este perfect opac, niciun pixel să fie transparent , și deci nu se va încerca desenarea lucrurilor din spatele ei, dacă acestea există. Dacă returnează `false` , framework-ul va încerca să deseneze și lucrurile din spatele obiectului , chiar dacă vor fi sau nu vizibile.

Am decis să returnez `true` , pentru a reduce din overhead-ul de performanță .

5. Teste de performanță

Teste pentru Triangle Vectorizer

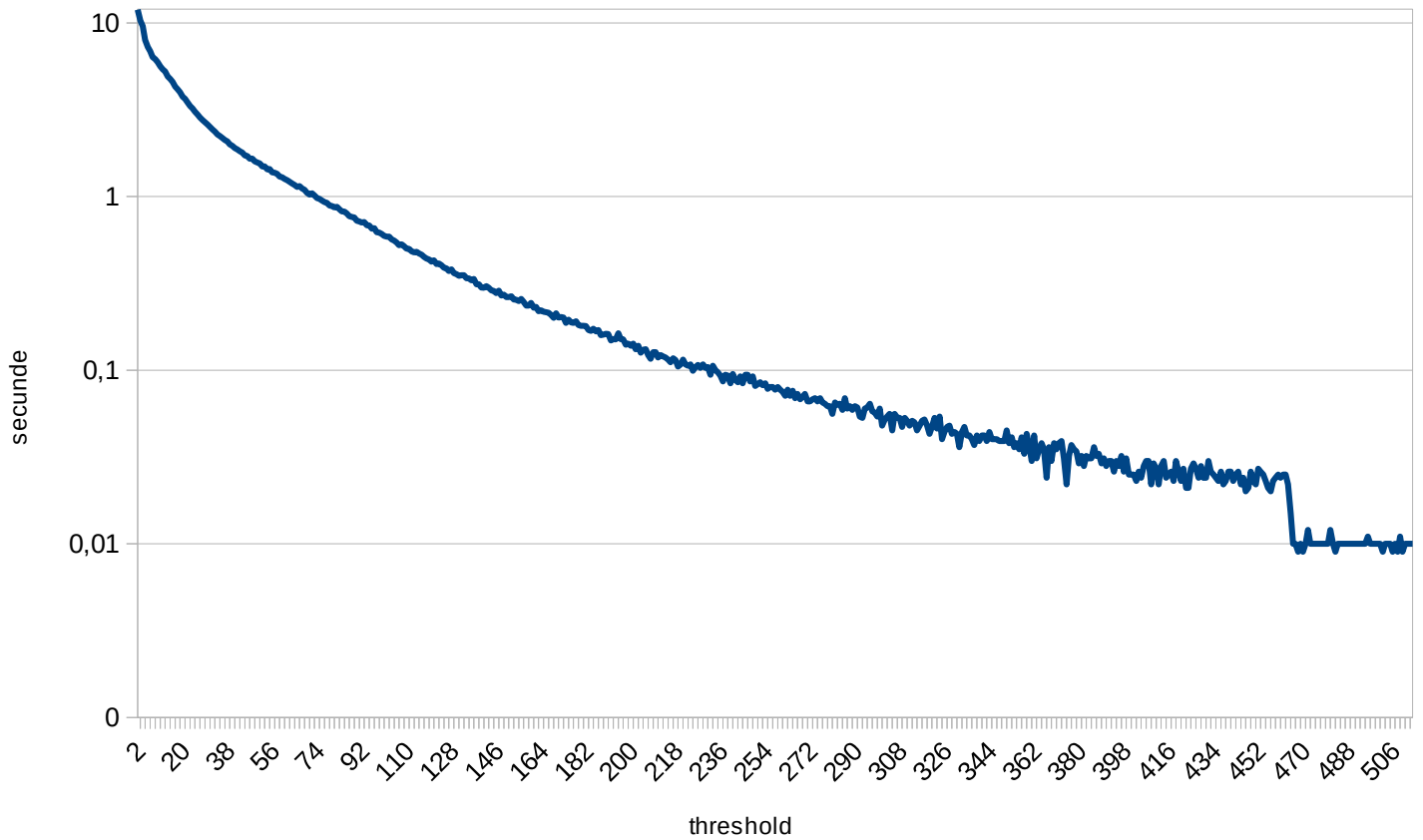
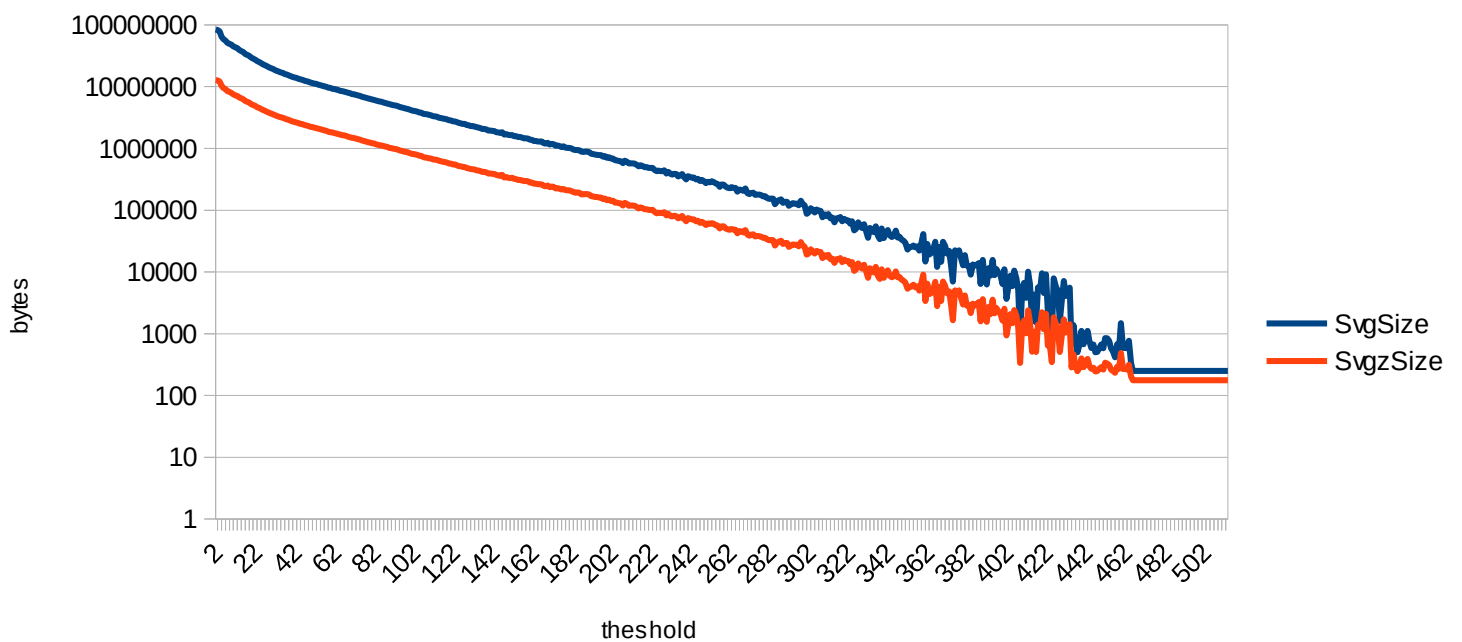
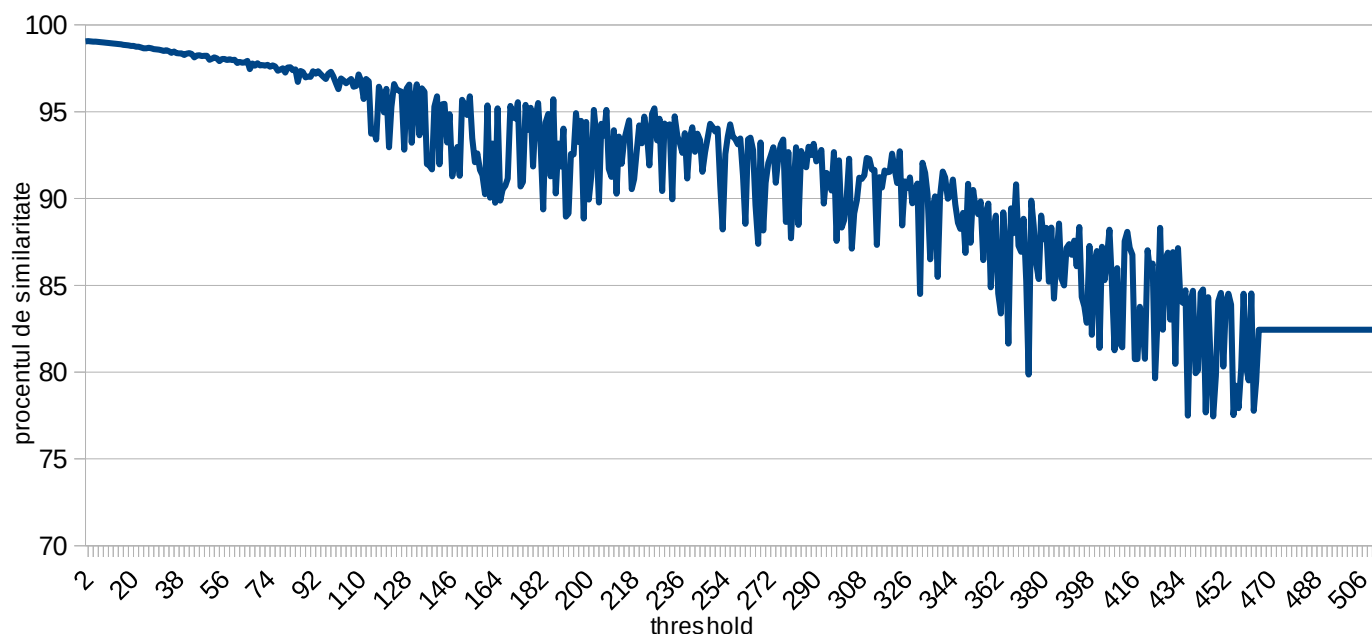


Figura aceasta măsoară în secunde timpul de vectorizare al imaginii.

Având în vedere faptul ca procesul folosește doar două thread-uri, sunt mulțumit de performanța pe care o oferă.



Se poate observa imediat fenomenul bizar când *threshold* este foarte mare, timpul de execuție este foarte mic. *Threshold*-ul este atât de mare încât vectorizarea returnează doar două triunghiuri mari care ocupă toată imaginea. Figura această măsoară dimensiunea fișierelor SVG și SVGZ în bytes.



La prima vedere, nu pare a fi prea mare diferența. Dar aceasta este o scară logaritmică. În majoritatea cazurilor se vede cum SVGZ este aproximativ de 10 ori mai mic decât SVG.

Această figură reprezintă procentul de similaritate cu poza originală. Se vede imediat cât de mult variază la diferite valori de *threshold*, dar până la urmă scade puțin câte puțin.

Chiar dacă valoarea de *threshold* este 0, vectorizarea folosind triunghiuri nu poate să reproducă 100% poza originală.

Teste pentru Square Vectorizer

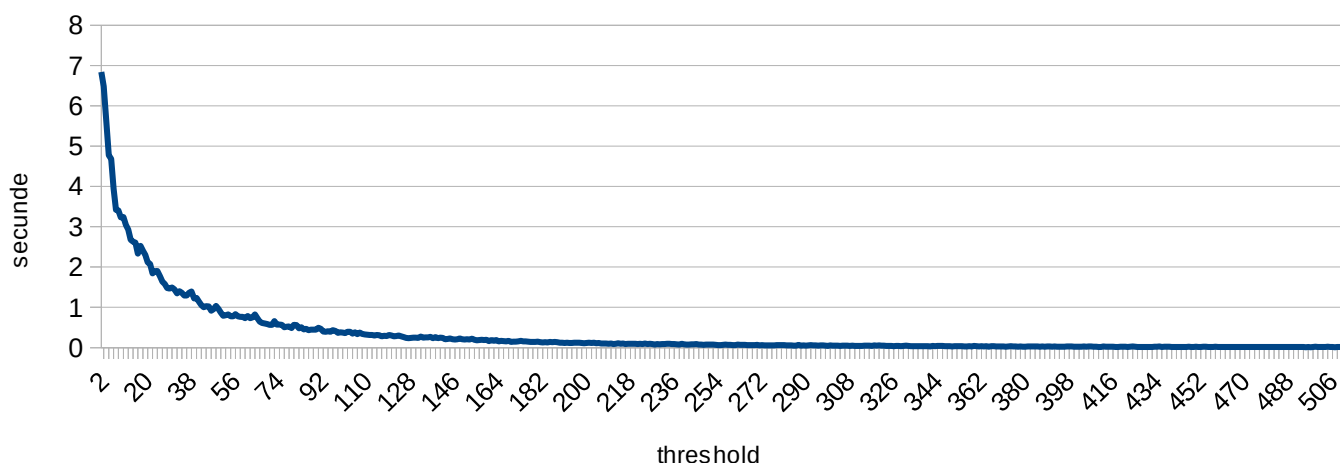


Figura aceasta măsoară în secunde timpul de vectorizare a imaginii.

Având în vedere faptul că vectorizarea cu dreptunghiuri folosește doar patru thread-uri, sunt mulțumit de performanța pe care o oferă.

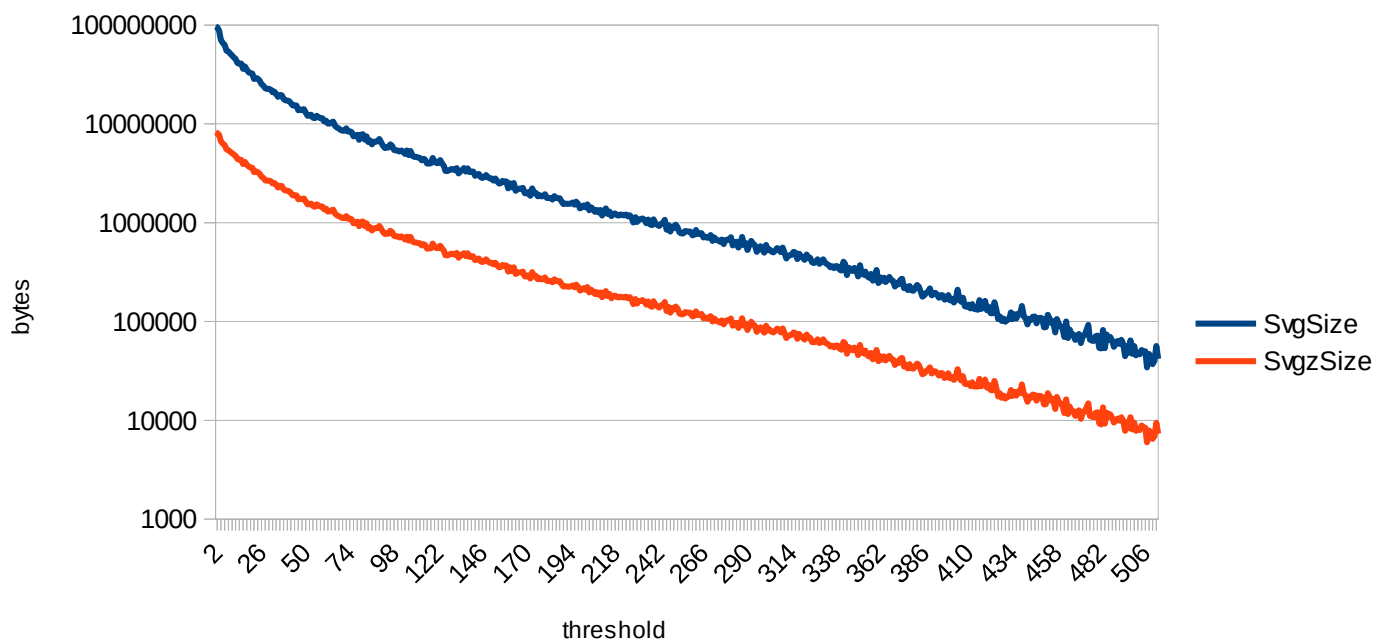
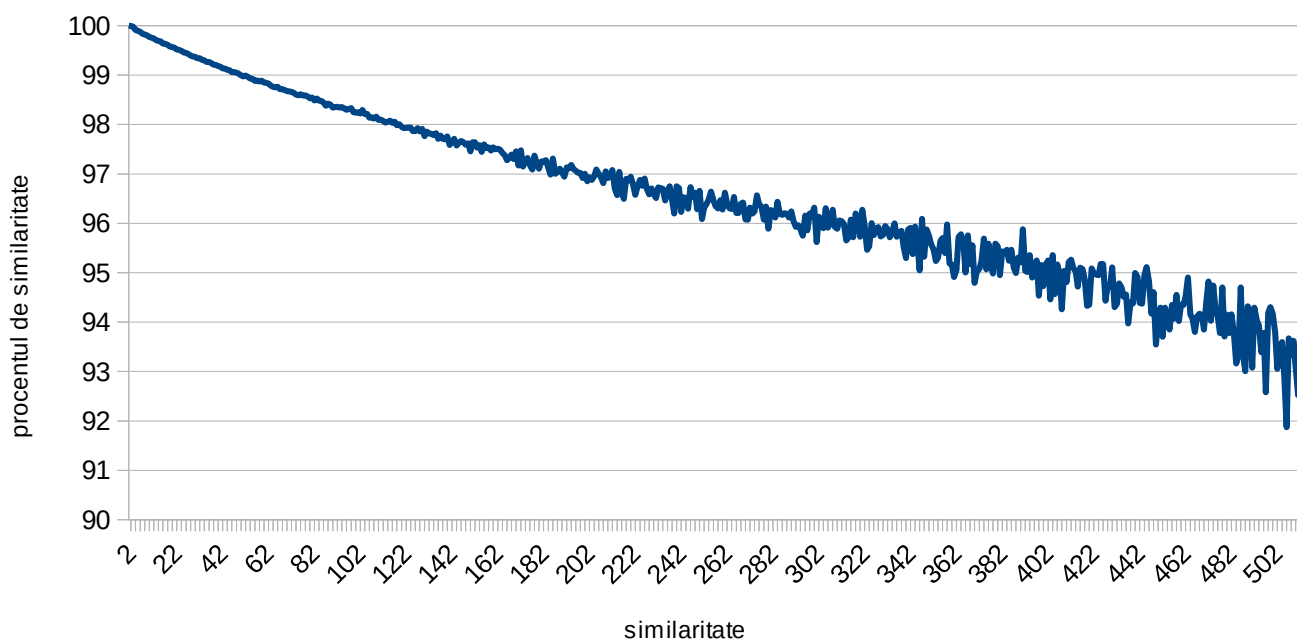


Figura aceasta măsoară dimensiunea fișierelor SVG și SVGZ în bytes.

La prima vedere, nu pare a fi prea mare diferență. Dar aceasta este o scară logaritmică. În majoritatea cazurilor se vede cum SVGZ este aproximativ de 10 ori mai mic decât SVG.



Acestă figură reprezintă procentul de similaritate cu poza originală. Se vede imediat cât de mult variază la diferite valori de *threshold*, dar până la urmă scade puțin câte puțin.

Vectorizarea folosind dreptunghiuri poate să reproducă 100% poza originală, dar cu dezavantajul că este irosit foarte mult spațiu.

Teste pentru Polygon Vectorizer

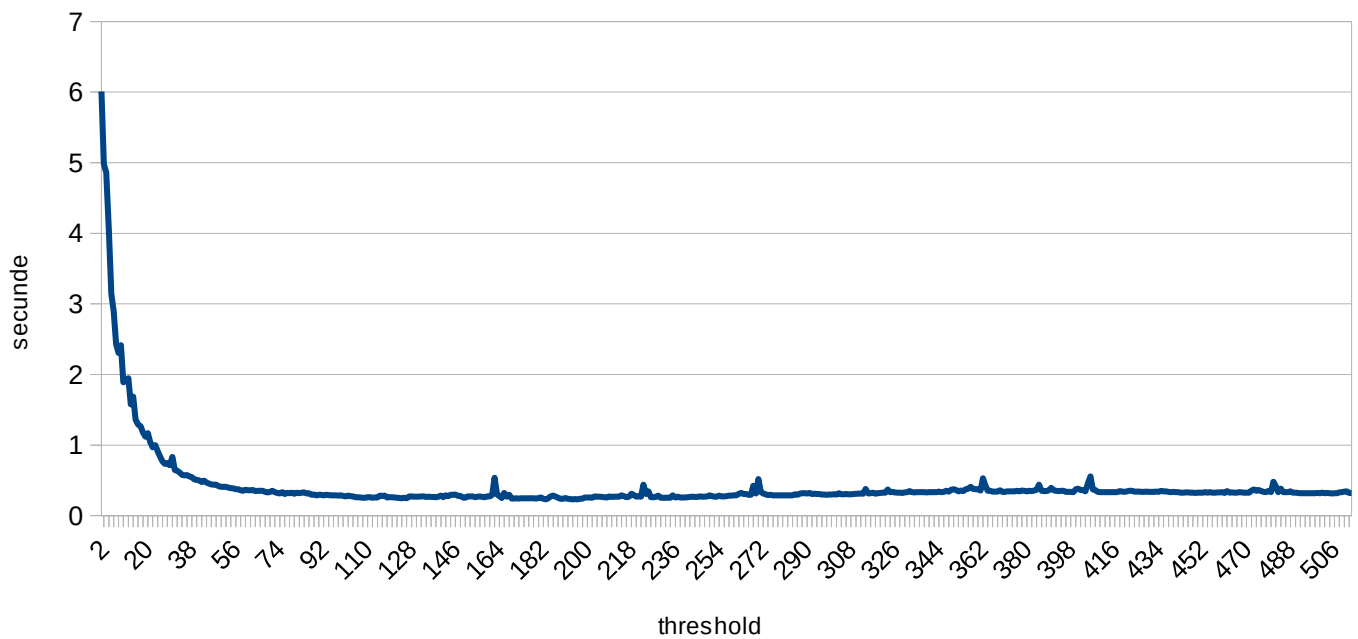


Figura această măsoară în secunde timpul de vectorizare a imaginii.

Având în vedere faptul ca vectorizarea cu poligoane folosește doar un singur thread , sunt mulțumit de performanța pe care o oferă.

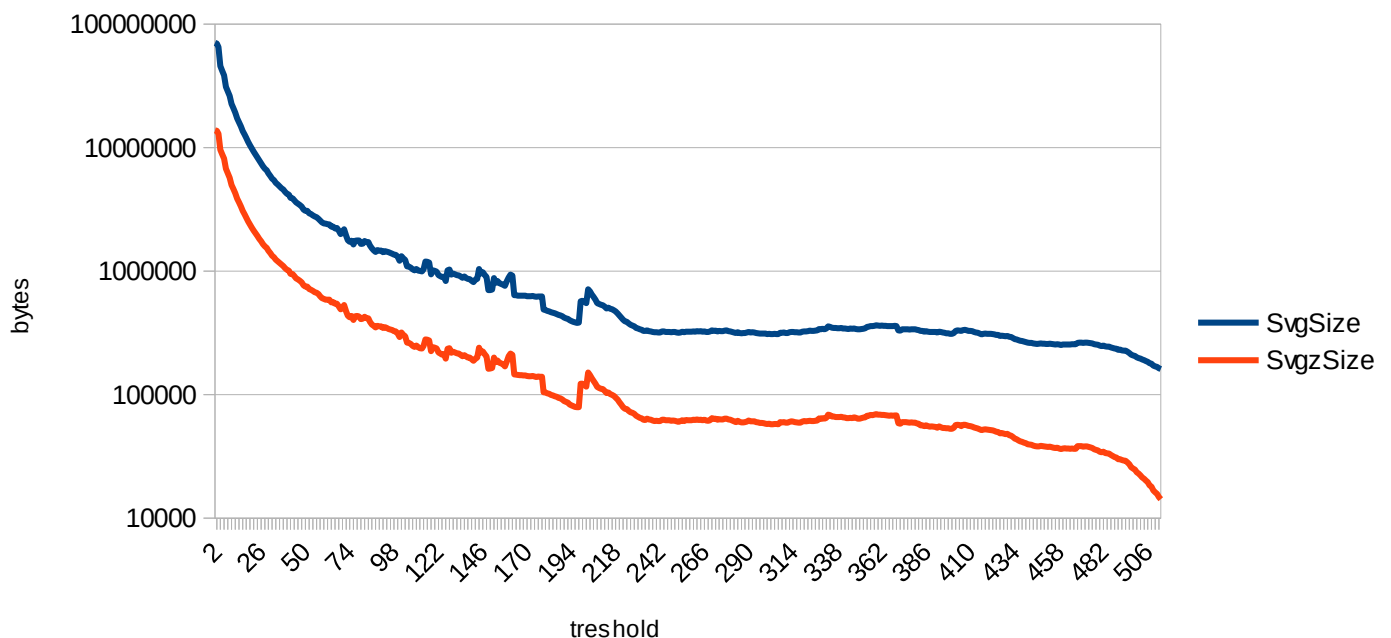
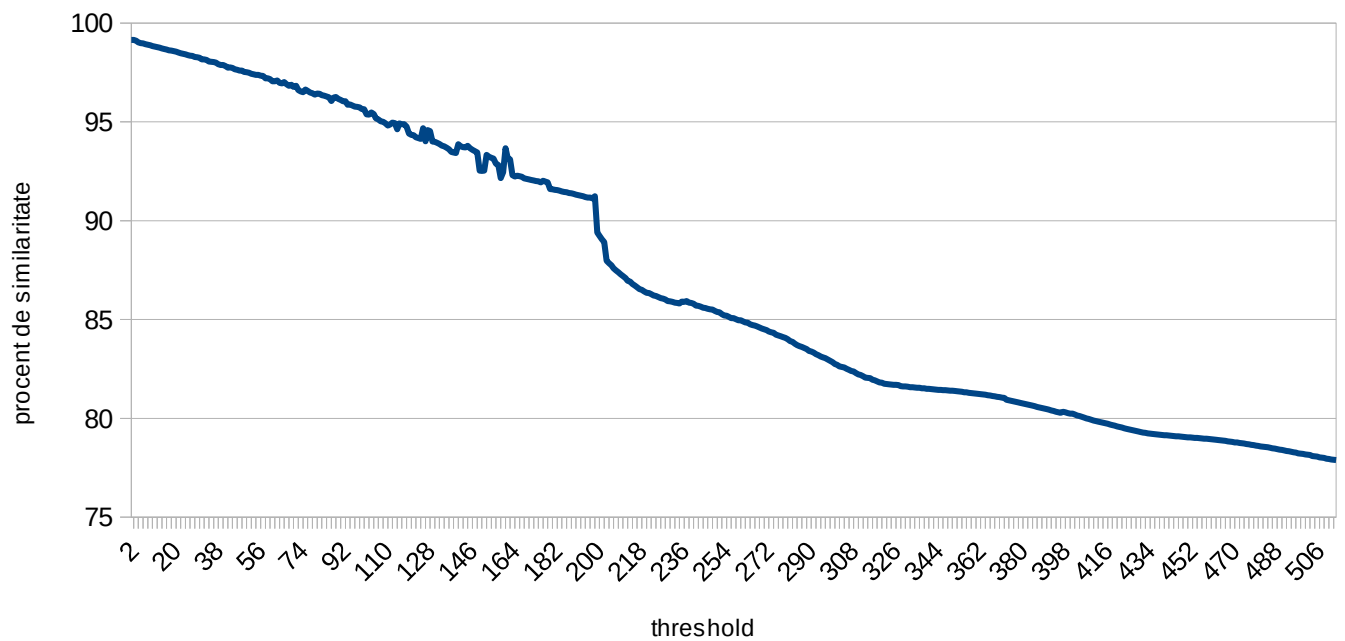


Figura aceasta măsoară dimensiunile fișierelor SVG și SVGZ în bytes.

La prima vedere, nu pare a fi prea mare diferența. Dar aceasta este o scară logaritmică. În majoritatea cazurilor se vede cum SVGZ este aproximativ de 10 ori mai mic decât SVG.



Această figură reprezintă procentul de similaritate cu poza originală. Se vede imediat cât de mult variază la diferite valori de *threshold*, dar până la urma scade puțin câte puțin.

Vectorizarea folosind poligoane poate să reproducă 100% poza originală, dar cu dezavantajul că este irosit foarte mult spațiu.

6. Exemple de poze vectorizate

Poza originala : JPG 1600 x 1200 370 KB



TriangleVectorizer: SVG: 6.12 MB ; SVGZ: 1.2 MB



RectangleVectorizer: SVG: 4.81 MB ; SVGZ: 0.65 MB



PolygonVectorizer: SVG: 4.22 MB ; SVGZ: 1.02 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. *Lincoln Laboratory Journal, Volume 19, Number 1, 2012*. 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>