

UNIVERSITATEA DIN BUCUREȘTI
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

LUCRARE DE LICENȚĂ

COORDONATOR ȘTIINȚIFIC

STUDENT

Roscaneanu George

BUCUREȘTI
data

Cuprins

Introducere.....	3
Definiții și scurt istoric.....	3
Comparație: Grafica vectorial VS Grafica raster.....	4
Motivația.....	5
Despre limbajul Java.....	5
Despre IDE-uri.....	6
Despre IntelliJ IDEA.....	7
Despre GitHub și Git.....	7
Despre implementarea lucrării.....	8
Continutul clasei BaseVectorizer.....	8
SquareVectorizer.....	11
TriangleVectorizer.....	12
PolygonVectorizer.....	13
Despre Scalable Vector Graphics (SVG).....	14
Benchmark Triangle Vectorizer.....	17
Benchmark Square Vectorizer.....	19
Benchmark Polygon Vectorizer.....	20
Exemple.....	22
Static Point Array.....	24
Clasa Utility.....	25
Clasa Image Panel.....	27
Clasa Triangle Vectorizer.....	28
Subclasa Job a clasei TriangleVectorizer.....	32
Clasa Square Vectorizer.....	38
Subclasa Job a clasei SquareVectorizer.....	39

Introducere

Definiții și scurt istoric

Grafica vectorială reprezintă folosirea primitivelor geometrice (puncte, linii, 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 pe axele X și Y ale spațiului de lucru și determină direcția liniilor. Fiecare dintre linii pot avea asigurate 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 fisierului 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 grafică raster pe baza de pixeli.

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

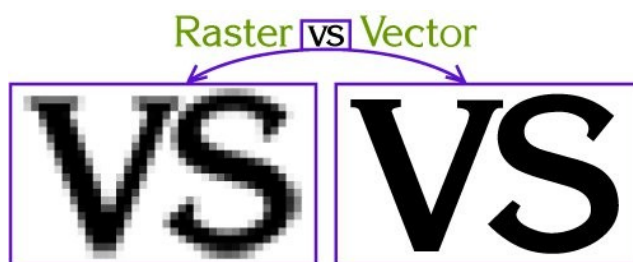
În tipografia pe calculator, fonturile moderne descriu caractere folosind curbe cubice sau cuadratice cu puncte de control. Oricum, fonturile pe baza de bitmap încă sunt folosite. Convertirea contururilor necesită umplerea lor; convertirea ei într-un bitmap nu este trivială, deoarece bitmap-urile nu au destulă rezoluție pentru a evita efectul de „aliasing”, în special caracterelor mai mici.

Una dintre aplicațiile mai moderne ale graficii vectoriale sunt spectacolele cu lumina 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.

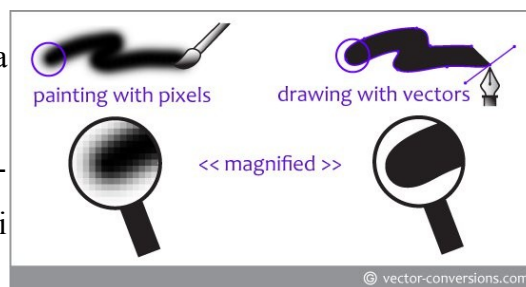
Comparație: Grafica vectorial VS Grafica raster

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.

Imaginile vector sunt formate din forme geometrice.



Se poate observa că atunci când imaginea este mărită suficient de mult, încep să apară diferențe evidente. Când cea raster a fost mărită, începe să piardă din calitate și claritate. Iar cea vectorială rămâne clar oricât de mult se va mări imaginea.



	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 grafica simplă , precum figuri geometrice și tipografie* 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	<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 vectoriale sunt create din linii și puncte , deci nu sunt la dispoziție efecte precum blurare sau umbre
Raster	<ul style="list-style-type: none">* Detalii bogate : o poză care are valoarea dpi mai mare va conține mai multe detalii subtile* Editare precisă : oricare dintre pixelii poate fi modificat independent ; o persoană poate să editeze poza respectivă cu ușurință	<ul style="list-style-type: none">* Blurată atunci când este mărită : din cauza că există un număr finit de pixeli, calculatorul va încerca să umple spațiul dintre pixelii cu interpolări între ele* Fișiere mai mari : cu cât dpi 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

Motivația

Într-o zi căutam un cadou pentru o prietena prin zona Pietii Unirii, iar unul dintre lucrurile care mi-a atras atenția au fost tricourile cu texte sau desene printate pe ele. Trecând prin toate modele pe care le aveau nu am găsit ceva destul de potrivit. Am întrebat vanzatoarea dacă poate să facă tricouri personalizate, dar cu singura condiție ca pozele să fie în format vectorial. Eram puțin uimit de ce driver-ul imprimantei nu este în stare să folosească imagini raster obișnuite dar am decis să nu comentez.

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

Primul lucru pe care l-am încercat a fost <http://vectormagic.com>. Acesta ofera 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 ofera decât posibilitatea de a vedea cum arata 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 nivele 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 orice salvarea afișează mai întâi o fereastră cu cronometru care nu sta deschisă până se duce timpul (aprox 12 secunde).

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 preturi relativ mari, am decis să creez câteva implementări pentru vectorizarea unei imagini, și să îl public codul sursă pe GitHub.

Despre limbajul Java

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

Java este un limbaj de programare concurent, bazat pe ierarhia de clase, orientat obiect și în mod special proiectat pentru a avea cât mai puține dependențe de implementare posibile. Autorii lui și-au propus să îi lase pe dezvoltatorii de aplicații să “scrie odată, să ruleze oriunde” (“write once, run everywhere”, WORA). Aceasta înseamnă că un cod care rulează pe o platformă nu mai necesită recompilare pentru a rula pe o altă. 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.

Originalele compilatoare Java, mașini 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 facilitati low-level , exista JNI (Java Native Interface) , un framework de programare care permite codului scris în Java și care ruleaza intro Mașina Virtuala Java sa apelez 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 ca 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 diverse platforme , codul scris în nativ este foarte greu de debug-uit . Luând în considerare aceste dezavantaje am decis ca programul sa nu conțină cod scris în nativ.

Pentru scrierea codului sursa , 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 automate a codului (Intelligent code completion).

Unele IDE-uri conțin un compilator, un interpretor, sau ambele, cum ar fi cazul 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 refactoriza codul deja integrat.

Într-un reportaj Infoworld din 2010 , IntelliJ a primit cele 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 programului IntelliJ IDEA. Alte medii de programare bazate pe IntelliJ sunt AppCode , PhpStorm , PyCharm , RubyMine , WebStorm și MPS .

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

Ambele ediții pot avea mai multe capacități comune precum : Smart Code Completion , On-The-Fly Code Analysis , Advanced Refactoring , unelte de Version Control având interfața 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 viteza de procesare mare , păstrarea integrității datelor și suport pentru posibilitatea de avea „workflow”-uri distribuite și non-liniare . Git a fost original creat de Linus Torvalds pentru programarea kernel-ului Linux în 2005, și de atunci a devenit unul dintre cele mai adoptate sisteme de version control.

Cea 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. Astfel 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 comandă. Deși a fost puternic influențat de BitKeeper, Torvalds intenționa să evite tehnicile conventionale , astfel ajungând la un design unic.

Git permite unui developer sa verifice ce modificari au fost făcute asupra unui proiect , sau adauga noi modificari la un repozitor , crearea unei noi „ramuri” de proiect pentru a crea un feature nou și apoi reunirea acestei ramuri la ramura principala pentru ca întreaga proiectul original sa conțină acest nou feature. Deasemenea dacă au fost făcute greșeli , se poate face reîntoarce codul la un „commit” precedent.

Am decis sa îl folosesc deoarece doresc sa îl învăț mai mult și sa abuzez de comparatia ușoară a codului cu commit-urile precedente în caz în care am făcut greșeli.

Exista optiunea de a avea repozitorul stocat doar local pe disk, dar nu vroiam sa risc pierderea proiectului așa ca am decis sa folosesc GitHub pentru a stoca proiectul.

GitHub este o serviciu online de hostare a repozitoarelor Git , astfel având toate capabilitatile Git-ului dar și adaugat câteva la acestea. Spre deosebire de Git care ofera strict doar un program care se folosește prin linii de comanda, GitHub over o interfata grafica pentru Web , desktop și telefoane smartpone. Deasemenea ofera administratorului posibilitatea de schimba nivelul de acces al celorlaltor colegi care lucrează la proiect , și unelte de colabaroare precum bug tracking , feature request , managerierea task-urilor și crearea wiki-uri.

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

GitHub este de obicei folosit doar pentru cod sursa, dar ofera în plus câteva formate și formate în plus precum vizualizarea fisierelor grafice 3D sau fisiere native PSD de PhotoShop și compararea lor cu celelalte versiuni ale acestora.

Despre implementarea lucrarii

După cum am menționat anterior, proiectul meu este realizat in Java. În proiectul meu am creat trei clase de vectorizare care extind o clasa BaseVectorizer.

Clasa BaseVectorizer a fost creata pentru usura tastarea codului în restul aplicației , astfel reducand codul care este „copy-pasted” și lasand posibilitatea de a integra cu ușurința vectorizer-ele în codul pentru GUI.

Continutul clasei BaseVectorizer

```
protected BufferedImage originalImage;  
protected BufferedImage destImage;
```

În originalImage este păstrată referinta către poza originala în format raster , iar în destImage este pusa imaginea după ce a fost transformata în format vectorial și apoi folosită să fie afisata în interata GUI. Motivul de ce am ales obiectele să fie de tip BufferedImage este din cauza ca exista functia getRGB(int x,int y) deja implementata și ajuta la luarea informațiilor. Din păcate nu am avut încredere deplina în viteza de procesare a functiei getRGB așa ca am creat câteva array-uri ajutătoare.


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

public void calculateColorArrays()
```

În aceste array-uri sunt pastrate valorile culorilor roșu , verde, albastru și combinația lor. Motiv de ce primele 3 array-uri sunt de tip char este deoarece char în Java ocupa un singur byte iar valorile sunt de la 0 la 255 în comparație cu tipul byte care este de la -128 la 127. Primele trei array-uri sunt folosite pentru a calcula media culorilor dintr-o zonă. În descrierea vectorizatoarelor vor fi mai multe detalii despre acestea.

Metoda calculateColorArrays() este chemată la imediat după ce s-a setat o 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. Ea decide cât de agresivă să fie atunci când sunt comparate două culori . Dacă distanța manhattan a culorilor dintre doi pixeli este mai mare decât threshold atunci acei doi pixeli nu pot aparține aceleiași figuri geometrice. Astfel dacă threshold are valoare mai mică atunci există o șansă mai mare ca pixelii să aparțină aceleiași figuri geometrice astfel crescând mărimea fisierului dar și claritatea ei în comparație cu poza originală.

Într-un alt capitol voi prezenta și diferențele din punct de vedere vizual și al mărimumi fisierului creat.

```
protected JPanel destImagePanel;
```

Clasa JPanel a fost creată deoarece nu am reușit să găsesc un element GUI care să afișeze o imagine. Acest obiect îl folosesc de asemenea în logica codului , de exemplu dacă destImagePanel == null atunci nu este necesar să execut anumite bucăți de cod.

```
protected Thread lastJob;
```

Clasa Thread a fost creată deoarece din câte am înțeles clasa Thread nu conține o metodă de încredere de a opri subit o linie de execuție. Cea mai apropiată este metoda stop() , care din păcate este deprecată deoarece era considerat instabil și periculos pentru mașina virtuală Java.

Această clasă acum conține un boolean canceled care inițial este false. Atunci când se apelează setCanceled(true) , canceled va primi valoarea true. Acum depinde de ce se află în implementarea metodei run() , ea acum trebuie să verifice în mai multe puncte dacă canceled a devenit true și să aștepte execuția într-un mod curat (dacă sunt necesare închiderea unor streamuri sau thread-uri secundare).

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

Folosit atunci când un vectorizator dorește crearea unui nou jobThread sau oprirea celui curent.

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

Pentru a permite userului sa încerce rapid diferite valori pentru threshold , codul trebuie sa poate crea și opri cât mai rapid thread-urile dar în același timp sa elimine posibile probleme de mulți-threading precum data racing. Metoda startJob() are grija ca mai întâi dacă exista un jobThread care deja ruleaza sa îl oprească cu metoda setCanceled() și sa aștepte terminarea lui. Abia după ce s-a terminat ultimul thread se poate crea unul nou în siguranță. Metoda cancelLastJob() executa la fel în afara de crearea unui nou thread.

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

Atunci când se dorește exportarea rezultatului curent în format SVG sau SVGZ, se deschide un fisier , se obtine OutputStream-ul lui , se alege dacă va fi exportat comprimat sau nu , și se apeleaza functia. Dacă s-a ales optiunea de comprimare a fisierului , folosesc un obiect de tip GZIPOutputStream pentru comprimare. Ce este interesant cu acest obiect este ca metoda de comprimare este compatibila cu InkScape pentru a deschide fisierul nou creat. Dupa aceea obiectul este infasurat cu un BufferedOutputStream , din câte am înțeles acesta poate ajuta la procese incete de scriere .

```
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 ajutatoare pentru accesarea valorilor culorilor pentru anumiti 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 chemata atunci cand setOriginalImage(BufferedImage image) primește ca parametru un image care este diferit de fostul obiect originalImage.

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

Metoda aceasta verifica dacă imaginea setata este null atunci sa oprească jobThread-ul curent și sa reseteze valorile auxiliare la valorile lor default. Altfel dacă noul image este diferit fata de cel vechi atunci se va executa procesul de inițializare.

```
protected abstract void constructStringSVG();
```

De fiecare data când se termina un proces de vectorizare se construiește un obiect de tipul StringBuilder care sa conțină în format SVG informațiile despre figurile geometrice. Am ales StringBuilder pentru ca este recomandad pentru construirea obiectelor String de dimensiuni mari.

Am initializat obiectul StringBuilder cu un buffer de aproximativ 2 MB pentru a ajutat la performanța. Dacă se încearcă crearea unui string și mai mare de 2 MB , obiectul va reloca un

buffer mai mare.

SquareVectorizer

Acest vectorizator creeaza o lista de dreptunghiuri care acoperă întreaga suprafața a pozei originale. Fiecare dintre dreptunghiuri are o culoare care este calculata făcând media culorilor pixelilor care apartin dreptunghiului respectiv.

Pentru a ajuta codarea acestei clase , am creat o clasa ajutătoare SquareFragment care reține marginile left , right , top , down și culoarea dreptunghiului. Toate campurile sunt publice pentru a fi folosite mai rapid. Pentru a spori siguranța, classa SquareFragment conține o funcție isValid() care return true dacă marginile sunt valide.

Procesul de divizare în dreptunghiuri este recursiv, încercând prima data un dreptunghi care ocupa toată suprafața imaginii , și dacă aceste nu e considerat bun el va fi fragmentat în 4 dreptunghiuri și procesul continua de acolo.

Am decis ca procesul de fragmentare să fie randomizat și în același timp limitat. Aleg aleator doua valori midX și midY ambele aflanduse între o patrice și trei patrimi din dreptunghiul original. Astfel lasand posibilitatea la utilizator sa reincearca vectorizarea pana când este mulțumit. Din cauza naturii aleatoarea, este foarte posibil ca mărimea fisierului generat după o imagine sursa și un threshold anume sa difere considerabil.

În primele revizii ale programului midX și midY erau pur și simplu alese să fie în mijlocul dreptunghiului, dar acestea generau imagini vectoriale care avea aspect de „caiet de matematica”; aratand tabelat și stintific .

Valorile alese aleator pentru midX și midY dau un aspect mai plăcut imaginii vectoriale generate.

Procesul de validare al unei dreptunghi consta în verificarea fiecărei culori și insumarea tuturor culorilor pentru a genera media culorilor. Dacă distanța manhattan dintre cea mai apropiata culoare de negru și cea mai apropiata culoare de alb este mai mare decât valoarea threshold înseamna ca procesul de validare se poate anula în momentul acela deoarece acest dreptunghi ales este prea mare și conține culori prea diferite. Dacă un dreptunghi este respins acesta este taiat în patru și se va încerca validarea acestora. Dacă un dreptunghi este valid , media culorilor din acel dreptunghi este stocata în câmpul color de la obiectul SquareFragment folosit la validare , iar apoi este adaugat unei liste care a fost trimisa ca argument.

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

Când fiecare din thread-uri își termina executia , 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 făcut rost de ArrayList-ul cu toate dreptunghiurile , se construiește string-ul care va conține lista în format SVG. Am decis sa preconstruiesc acest string deoarece pe ecran este afisat mărimea fisierului când este salvat în formatele SVG și SVGZ.

Odată ce s-a obținut lista finală, se creează un nou BufferedImage în care este desenat o aproximatie a imaginii vectoriale și mărimea fisierului în format SVG și SCGZ. Utilizatorul poate sa observe rezultatele și sa exporteze fișierul.

TriangleVectorizer

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

Pentru a ajuta codarea acestei clase , am creat o clasă ajutoare Triangle care reține coordonatele varfurilor reținute sub forma de float, culoarea într-o valoare integer și câteva valori ajutoare precum xMin , yMin , xMax și yMax care reprezintă bounding-box-ul tringhiului având lungimea și latimea paralele cu axele X și Y ale imaginii. Deasemenea în fiecare obiect de tipul Triangle se mai afla 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 serializabilă și valorile și obiectul ajutoare sunt notate ca fiind transient ceea ce înseamnă ca ele nu vor fi scrise la serializare.

Procesul de divizare în triunghiuri este recursiv. În faza inițială imaginea care este dreptunghiulară este despartită în două triunghiuri pe una dintre diagonalele ei. Fiecare triunghi este verificat dacă satisfac valoarea curentă a threshold-ului. Dacă da, aceasta este adăugată la o listă de obiecte Triangle. Altfel aceasta este fragmentată 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 tringhiului. Linia care va fragmenta tringhiului în două este trasa de la colțul opus al laturii mari la un punct interpolat între cele două puncte ale laturii mari folosind valoarea r. În testele inițiale am observat ca dacă las valoarea r să fie selectat aleator între 0 și 1 produce rezultate neplăcute vizual. Din cauza naturii aleatoarea, este foarte posibil ca mărimea fisierului generat după o imagine sursă și un threshold anume sa difere considerabil. Deoarece r este ales aleator, las posibilitatea userului sa reîncepe vectorizarea pana când este mulțumit de rezultat. Din cauza naturii aleatoarea, este foarte posibil ca mărimea fisierului generat după o imagine sursă și un threshold anume sa difere considerabil.

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

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

Procesul de validare al unei triunghi constă în verificarea fiecărei culori și însumarea tuturor culorilor pentru a genera media culorilor. Dacă distanța manhattan dintre cea mai apropiată culoare de negru și cea mai apropiată culoare de alb este mai mare decât valoarea threshold înseamnă ca

procesul de validare se poate anula în momentul acela deoarece acest triunghi ales este prea mare și conține culori prea diferite. Dacă un triunghi este respins acesta este taiat în doua 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, care în acest moment va fi pur și simplu oprită procesul de fragmentare și verificare pentru el. Dacă un triunghi este valid, media culorilor din acel triunghi este 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 originat când sau creat primele două triunghiuri ale imaginii. Deoarece sunt satisfăcut de performanța 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 executia, rezultatele depuse în LinkedList-ul respectiv sunt acum depuse de către thread-ul principal într-un ArrayList de Triangles. Am ales ArrayList pentru performanța sporită la citire.

După ce s-a făcut rost de ArrayList-ul cu toate triunghiurile, se construiește string-ul care va conține lista în format SVG. Am decis să preconstruiesc acest string deoarece pe ecran este afișat mărimea fișierului când este salvat în formatele SVG și SVGZ.

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 culorilor pixelilor care aparțin poligonului respectiv.

Pentru a ajuta codarea acestei clase, am creat o clasă ajutătoare ColoredPolygon care reține coordonatele varfurilor stocate într-un array 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 află 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 și va fi detaliată mai târziu.

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

Procesul de divizare în poligoane constă în căutarea zonelor de culori care sunt similare și continue. Mulțimea de poligoane descoperită nu este cea mai eficientă deoarece am ales să folosesc metoda Greedy pentru descoperirea lor. Folosind metoda Greedy am scăzut dificultatea codului de al 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 randomizat sau aleator, o imagine va da aceleași rezultate de fiecare dată pentru un threshold anume.

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

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

Atunci când se găsește un pixel nevizitat , se cauta toți pixelii conectați de acesta și care diferenta culorilor între pixelul original și pixelii cei noi nu depășește threshold-ul curent. Toți pixelii conectați sunt apoi considerati vizitati. Fiecare poligon descoperit este adaugat la un ArrayList.

Spre deosebire de celelalte procese de vectorizare, acesta deseneaza pe parcurs din când în când imaginea vectoriala partial construita. A fost mai mult o alegere estetica , și pentru a oferi userului senzația ca programul lucrează la ceva.

După ce s-a făcut rost de ArrayList-ul cu toate poligoanele , se construiește string-ul care va conține lista în format SVG. Am decis sa preconstruiesc acest string deoarece pe ecran este afisat mărimea fisierului când este salvat în formatele SVG și SVGZ.

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. Specificatia 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 compresate. Deasemenea ele pot fi editate chiar și cu un editor text, dar în majoritatea cazurilor se folosește software specializat.

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

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

Începând cu 2001 , specificatia SVG a fost updatata la versiunea 1.1 . SVG Mobile Recommendation a introdus doua profile simplificate de SVG 1.1 : SVG Basic și SVG Tiny , menit pentru aparatele cu putere computationala redusă și display mai slab. O versiunea îmbunătățită numita SVG Tiny 1.2 a devenit mai târziu recomandarea autonoma.

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

Deși specificatia SVG se concentreaza în mod primar pe limbajul de grafica vectoriala , design-ul sau include și câteva capabilitati de baza pentru definirea aranjarii obiectelor în într-o pagina precum format-ul PDF de la Adobe. SVG-ul are informațiile necesare pentru aranja fiecare

glyph și imagine într-o poziția aleasă pe o pagina de printat. Spre deosebire XHTML are ca scop principal sa comunice content-ul sau , nu prezentarea lui , deci HTML specifica ce obiecte sunt afisate , nu unde sunt aranjate.

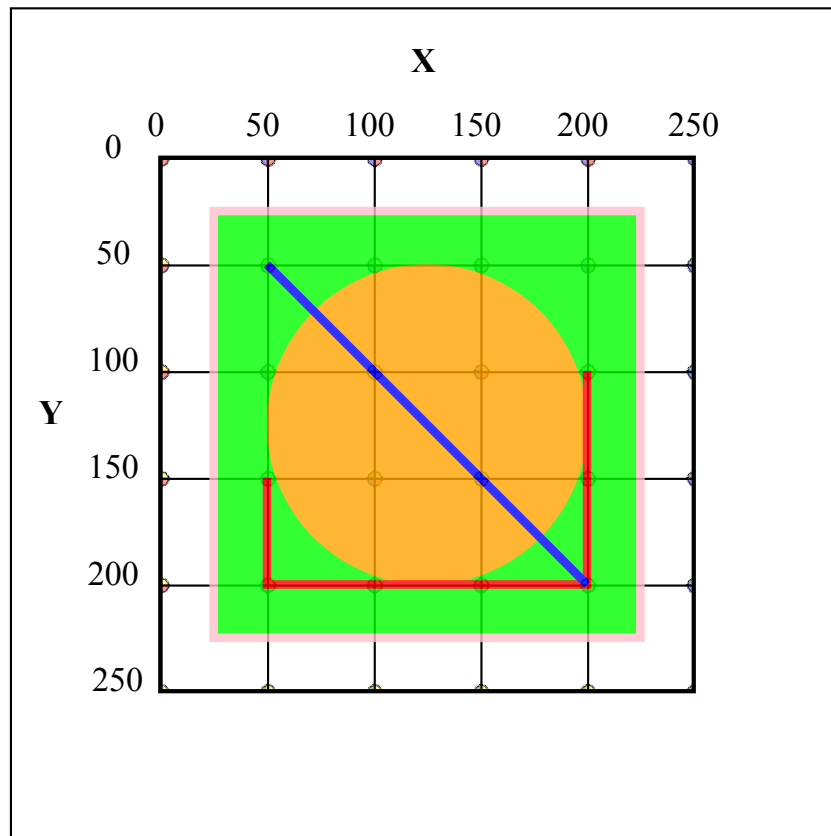
Imaginile SVG pot fi dinamice și interactive. Modificarile pe baza de timp asupra obiectelor pot fi descrise cu SMIL (Synchronized Multimedia Integration Language) sau JavaScript. Deși W3C recomanda SMIL , Google Chrome l-a deprecia în August 2015. Evenimente precum onmouseover și onclick pot fi folosite pentru a produce modificări și animații în SVG.

Deoarece SVG este scris în format XML , sunt foarte multe bucăți de text repetate, deci poate fi foarte ușor comprimat. Atunci când o imagine SVG a fost comprimată folosind algoritmul gzip , acesta este acum referit ca tipul SVGZ. De obicei o imagine SVGZ este de obicei 20-50% din imaginea originală.

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 litera mare , coordonatele sunt absolute , dacă se folosește litera mică , coordonatele sunt relative față de cele precedente.
- * Basic shapes
- * Text : Support pentru caractere Unicode. Este posibil să fie scris textul în ambele sensuri orizontale , dar și vertical pentru chineză. Textul poate de asemenea să fie scris de-a lungul unui path și să folosească multe efecte vizuale.
- * Painting : figurile pot fi umplute și/sau conturate folosind diverse culori , gradienti sau paterne.
- * Color : culorile sunt definite fie prin literale precum „black” , „red” , prin hexadecimale #63ff01 , #02f , prin decimale cu rgb(123,23,12) sau prin procente rgb(100%,0%,50%)
- * Gradients and patterns
- * Clipping, masking and compositing : conturul de alte elemente grafice , precum texte , paths , basic shapes pot fi folosite pentru a defini aceste lucruri
- * Filter effects : SVG are la dispoziție mai multe filtre precum : blend, gaussian blur , merge , tile etc
- * Interactivity : orice parte dintr-o imagine SVG poate avea event listeners
- * Linking
- * Scripting
- * Animation : conținutul SVG poate fi animat folosind elemente precum <animate> , <animateMotion> și <animateColor>
- * Fonts
- * Metadata

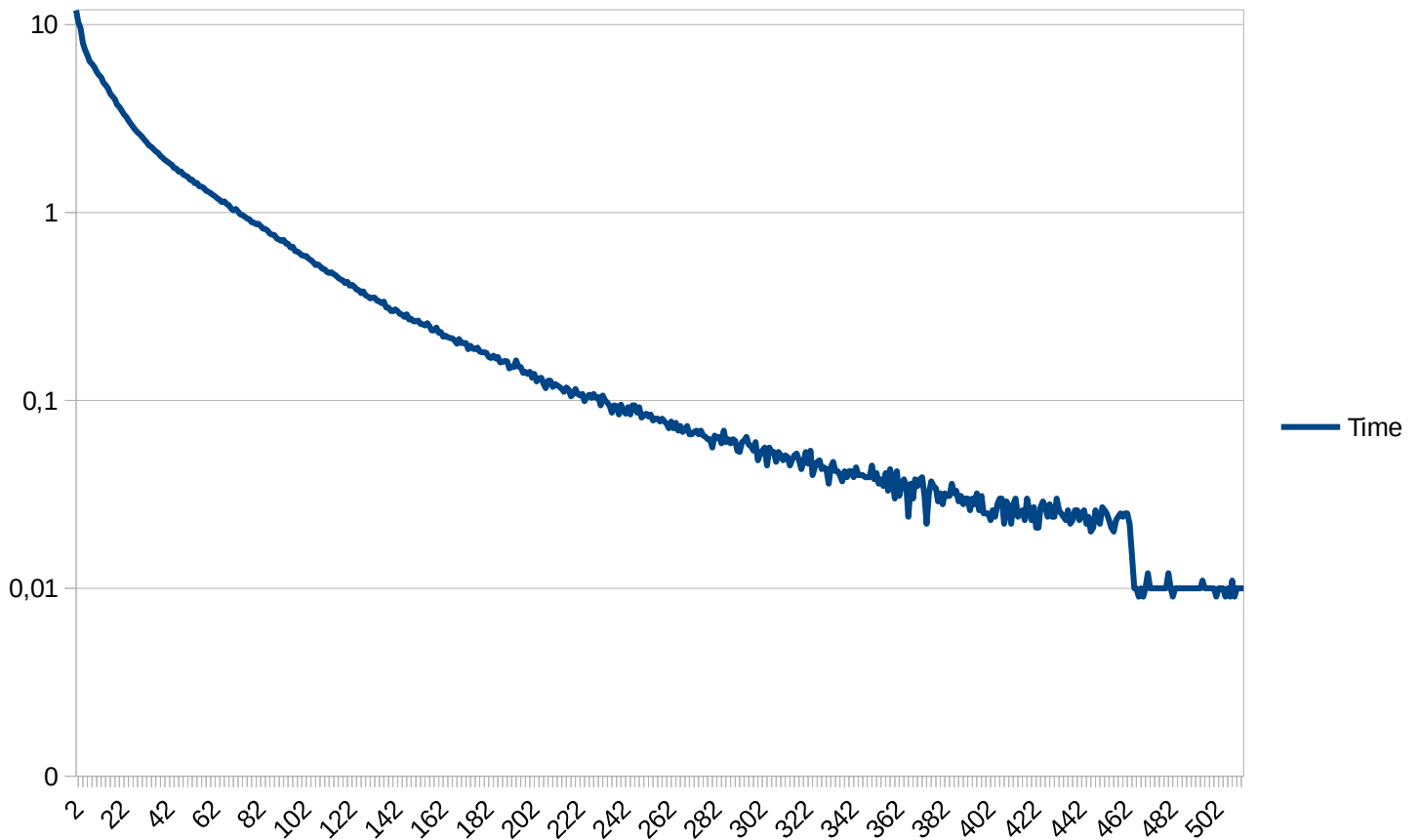
Exemplu :



```
<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 afara de grila și texte. Acesta poate fi deasemenea scris într-un notepad , salvat cu formatul SVG , și apoi deschis cu browser precum Firefox sau Chrome.

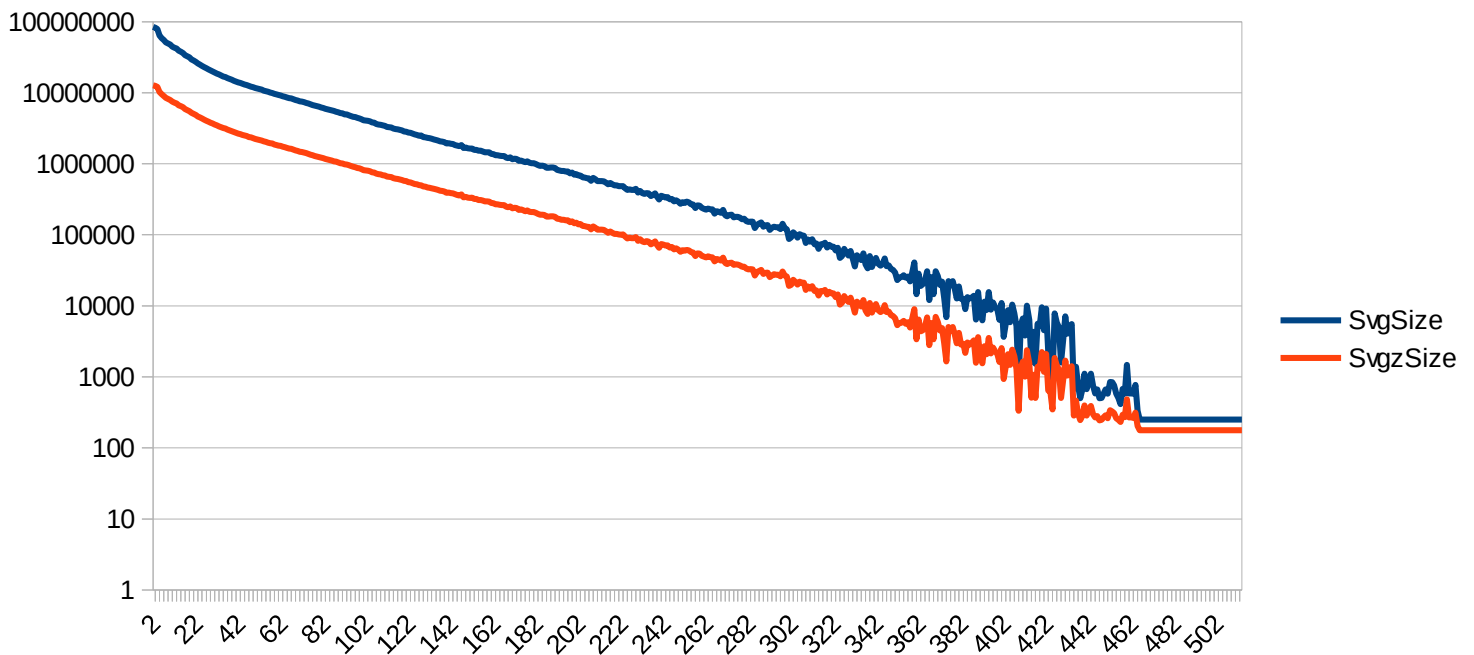
Benchmark Triangle Vectorizer



Tabelul acesta măsoară în secunde timpul de vectorizare a imaginii.

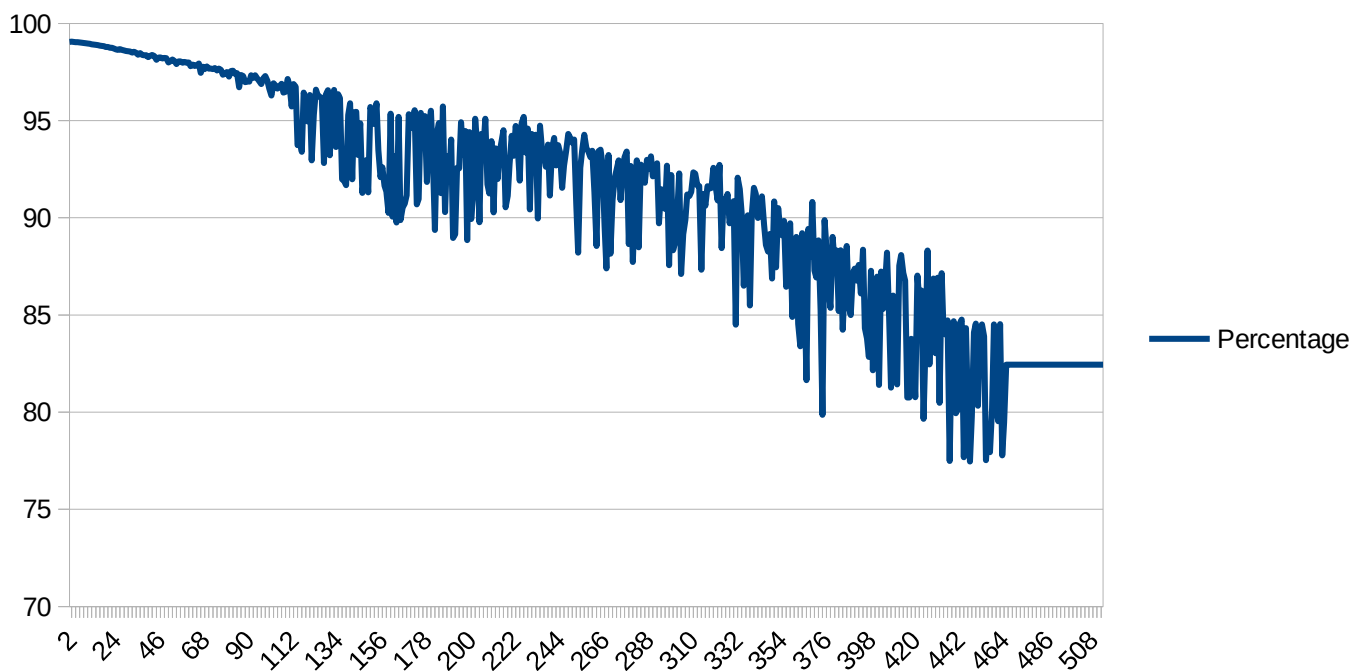
Având în vedere faptul ca triangularizarea folosește doar doua thread-uri , sunt mulțumit de performanța pe care o ofera.

Se poate observa imediat fenomenul bizar de după 462 , când timpul de execuție este foarte mic. Treshold-ul este de mare încât se vectorizarea returneaza doar doua trinughiuri mari care ocupa toată imaginea.



Tabelul aceste masoare valorile fisierelor SVG și SVGZ în bytes.

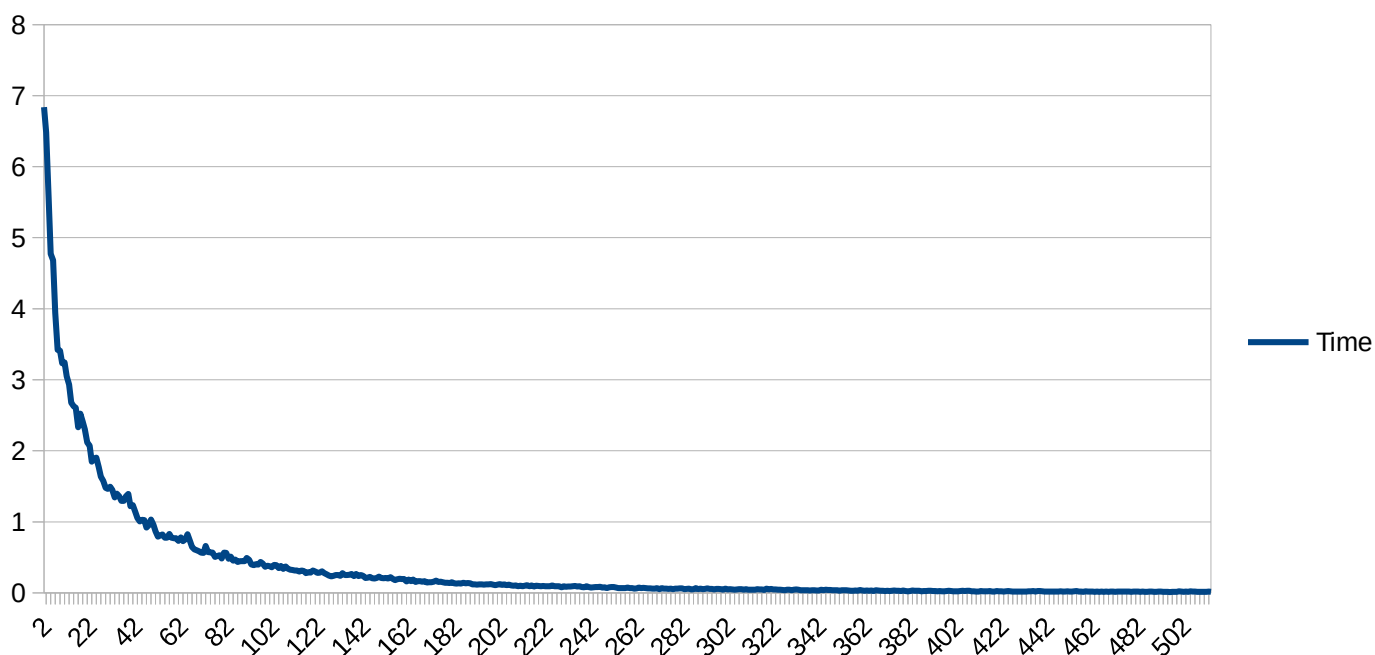
La prima vedere, nu pare a fi prea mare diferența. Dar aceasta este o scara logaritmica. În majoritatea cazurilor se vede cum SVGZ este aproximativ de 10 ori mai mic decât SVG.



Acest tabel reprezintă procentul de similaritate cu poza originala. Se vede instant cât de mult variaza la diferite valori de threshold , dar eventual scade câte puțin câte puțin.

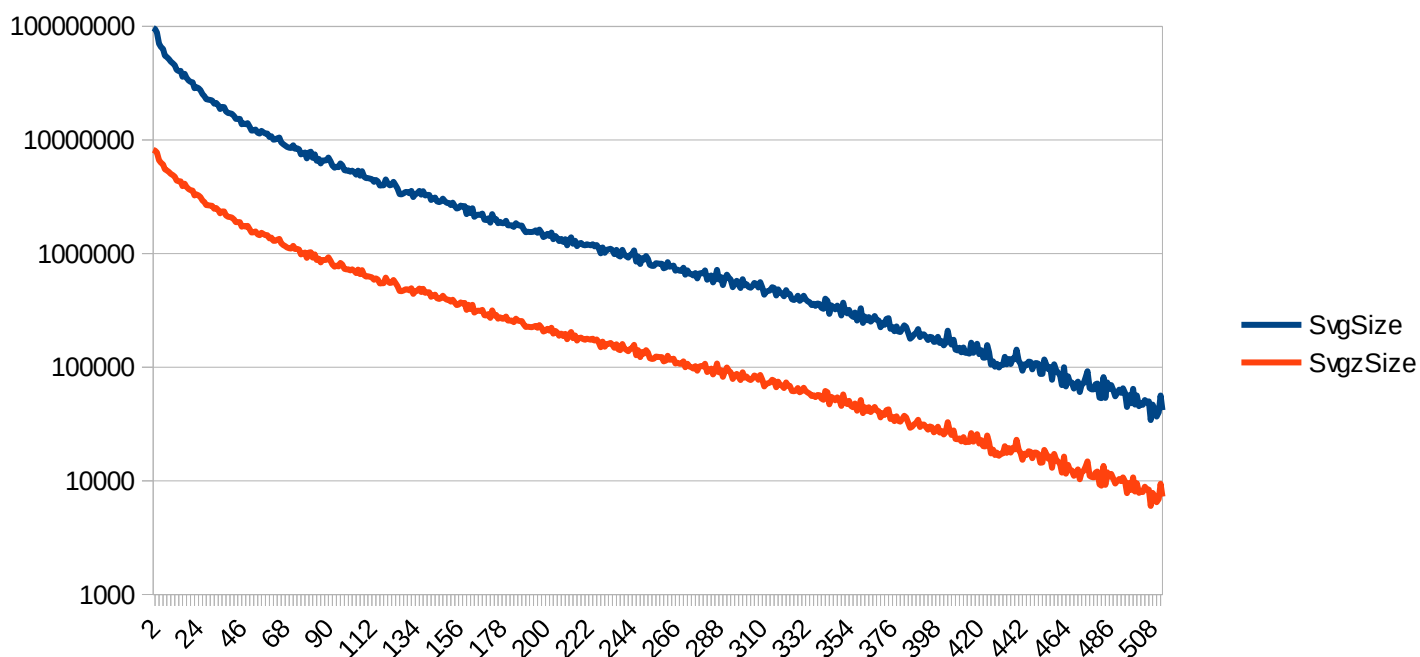
Chiar dacă valoarea de treshold este 0 , vectorizarea folosind triugnhiuri nu poate sa reproduca 100% poza originala.

Benchmark Square Vectorizer



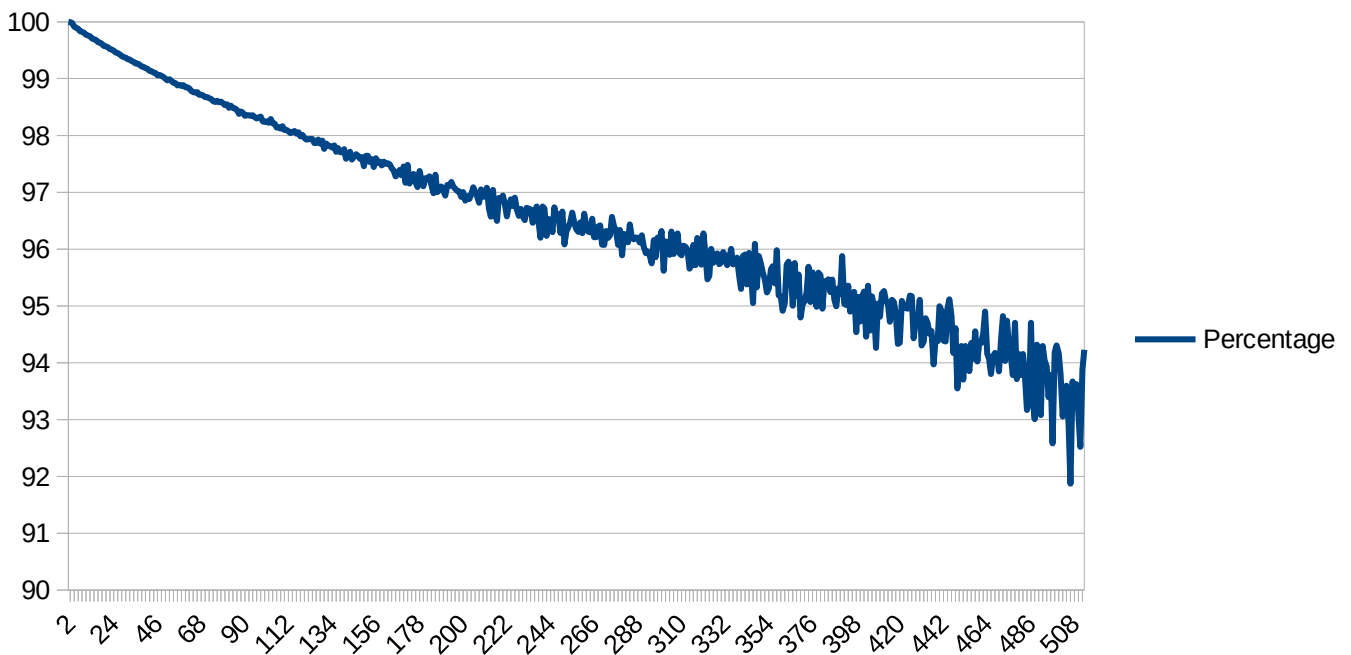
Tabelul acesta măsoară în secunde timpul de vectorizare a imaginii.

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



Tabelul aceste masoare valorile fisierelor SVG și SVGZ în bytes.

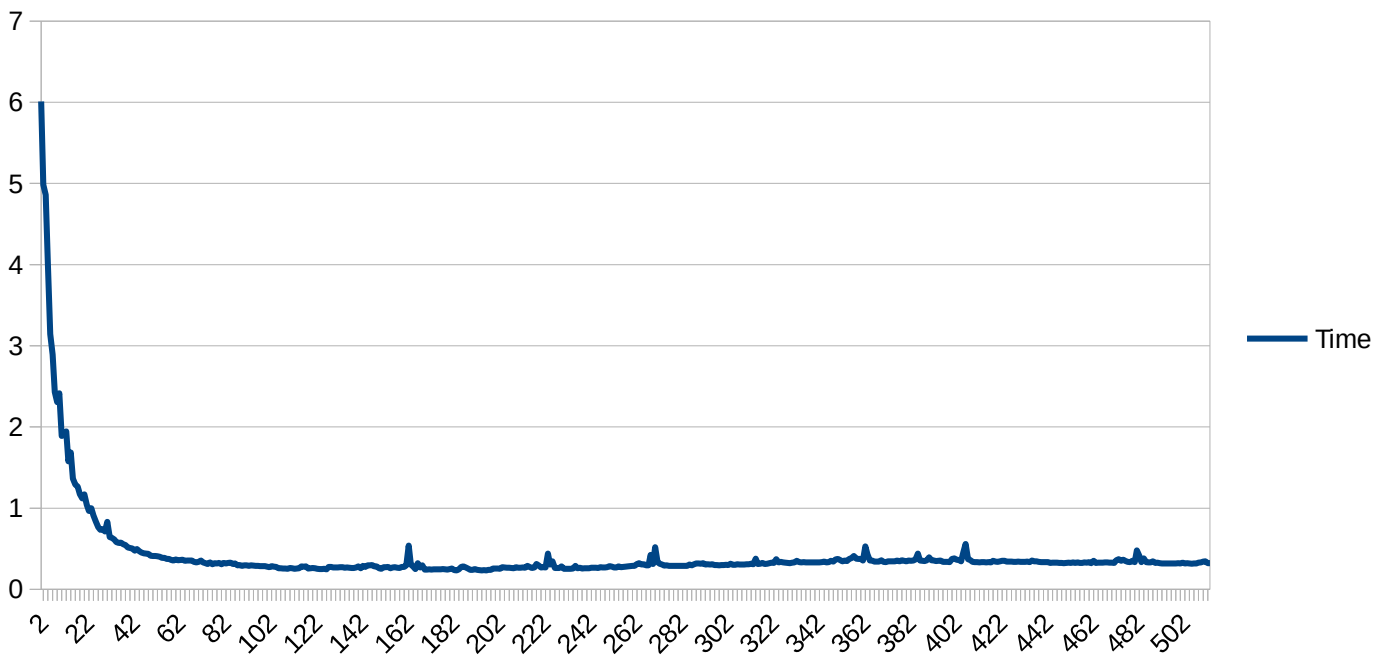
La prima vedere, nu pare a fi prea mare diferența. Dar aceasta este o scara logaritmica. În majoritatea cazurilor se vede cum SVGZ este aproximativ de 10 ori mai mic decât SVG.



Acest tabel reprezintă procentul de similaritate cu poza originala. Se vede instant cât de mult variaza la diferite valori de threshold , dar eventual scade câte puțin câte puțin.

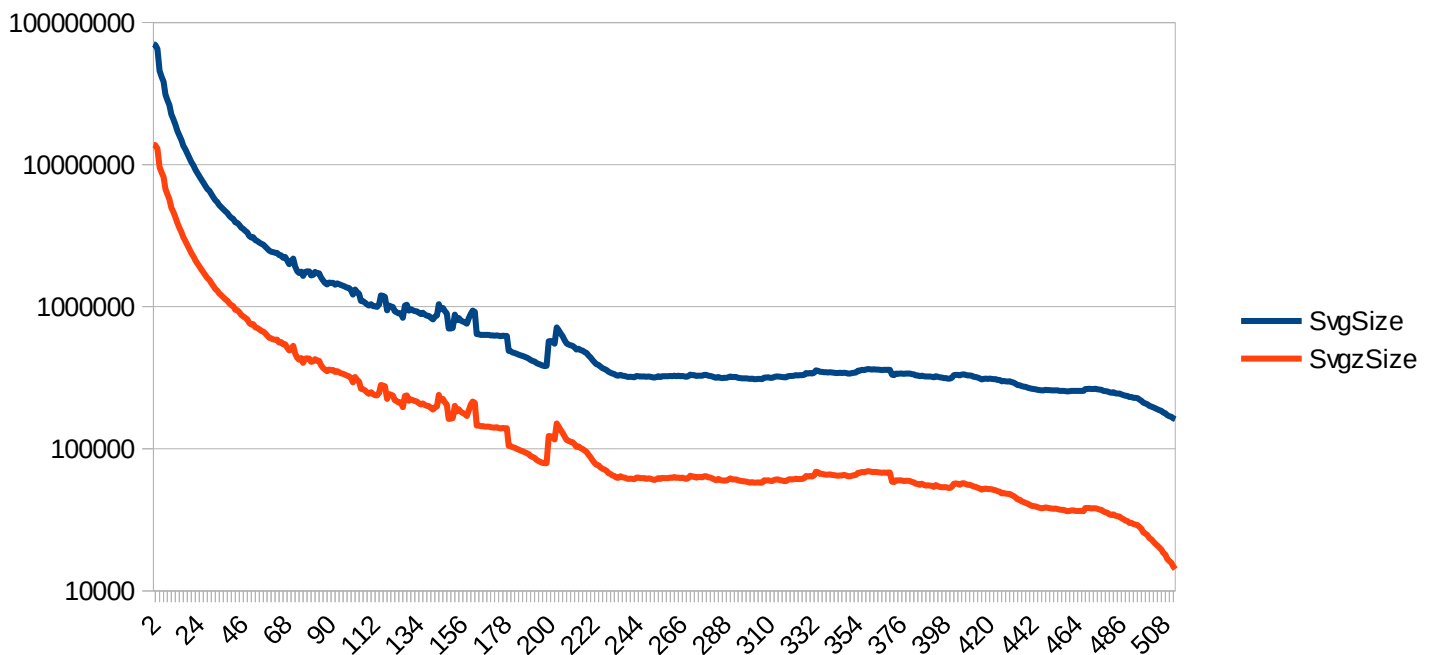
Vectorizarea folosind dreptunghiuri poate sa reproducă 100% poza originala , dar cu dezavantajul ca este irosit foarte mult spațiu.

Benchmark Polygon Vectorizer



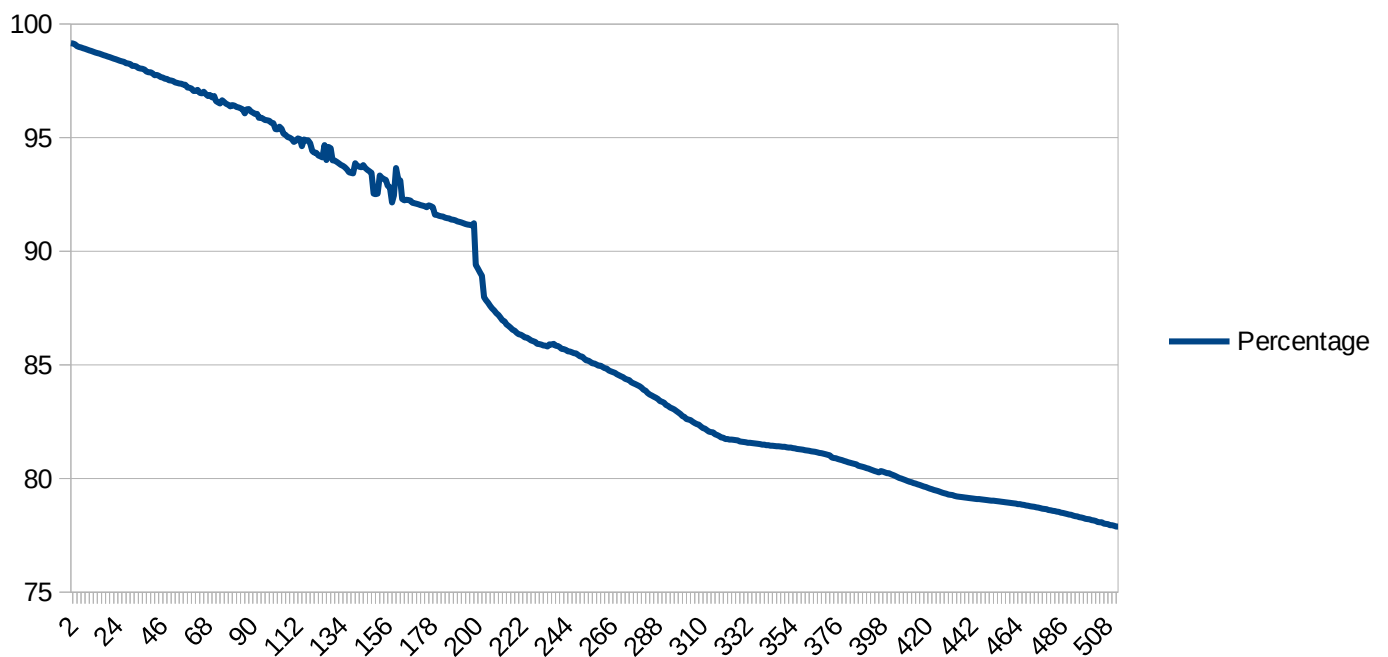
Tabelul acesta măsoară în secunde timpul de vectorizare a imaginii.

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



Acest tabel reprezintă procentul de similaritate cu poza originala. Se vede instant cât de mult variaza la diferite valori de threshold , dar eventual scade câte puțin câte puțin.

Vectorizarea folosind dreptunghiuri poate sa reproducă 100% poza originala , dar cu dezavantajul ca este irosit foarte mult spațiu.



Acest tabel reprezintă procentul de similaritate cu poza originala. Se vede instant cât de mult variaza la diferite valori de threshold , dar eventual scade câte puțin câte puțin.

Vectorizarea folosind dreptunghiuri poate sa reproducă 100% poza originala , dar cu dezavantajul ca este irosit foarte mult spațiu.

Exemple

Poza originala : JPG 1600 x 1200 370 KB



SVG: 6.12 MB ; SVGZ: 1.2 MB



SVG: 4.81 MB ; SVGZ: 0.65 MB



SVG: 4.22 MB ; SVGZ: 1.02 MB



Static Point Array

Pentru a mari viteza de procesare a perimetrelor la vectorizatorul de poligoane am creat o noua clasa care sa memoreze coordonatele punctelor perimetrelor.

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

Am ales sa folosesc doua array-uri de tipul *short* ca sa memorez coordonatele , deoarece nu ma aștept ca vreun user sa folosească poze care au înălțimea sau latimea mai mare de 32000.

Exista o variabila cu numele *count* care memoreaza lungimea curenta a datelor relevante din array-uri.

Descrierea metodelor :

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

Împinge la sfârșitul array-ului un punct nou, functia arunca excepții dacă operația nu este posibila. Operația are timp constant deoarece ea mereu introduce la finalul array-ului.

```
public void deleteLast()
```

Sterge punctul final din array , functia arunca excepții dacă operația nu este posibila. Tot ce se face este să se decrementeze variabila *count*. Operația are timp constant deoarece ea mereu sterge de la finalul listei.

```
public void clearAll()
```

Se sterg toate punctele din array. Tot ce se face este să fie setata variabila *count* la 0. Operația are timp constant.

```
public boolean isEmpty()
```

Se returneaza dacă *count* este egal cu 0.

```
public int size()
```

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

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

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

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

```
public short getLastX()
```

```
public short getLastY()
```

Funcții getter specializate, citește ultima valoare din array-uri.


```

public void setX(short i, short x)

public void setY(short i, short y)

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

```

Funcții setter pentru puncte de pe anumite poziții. În momentul acesta nu se verifica dacă index-ul se afla între 0 și *count*.

```

public StaticPointArray cloneUpTo(int s)

```

Funcția folosită pentru a crea un nou obiect StaticPointArray de mărime s care copiază din array-ul original până la indexul s. Timpul de execuție este liniar.

```

public void delete(int i)

```

Funcție pentru a șterge un punct de la o anumită poziție. Din păcate timpul de execuție este liniar, din cauza la array-uri.

```

public void copyFrom(StaticPointArray spa)

```

Am creat funcția asta pentru a copia rapid valori de la StaticPointArray la altul. Timpul de copiere este liniar, și arunca excepție dacă nu este suficient spațiu pentru a copia array-urile.

Clasa Utility

Deoarece majoritatea claselor folosesc mai multe funcții la comun am decis să le mut pe toate într-o clasă utilitară.

```

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

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

```

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

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

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

Aplicand AND binar cu 255 , obțin ultimi opt biti 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 diferenței 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 eficientă 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 creată deoarece am început să scriu în mod repetat o bucată de cod și am vrut să o încapsulez.

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

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

Clasa Image Panel

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

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

```
private BufferedImage image;
```

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

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

Deoarece clasa `ImagePanel` era una „custom” care extinde clasa `JPanel` a trebuit 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 înconjurez cu un `block try/catch` pentru a prinde erori , dar aparent din documentatia metodei `drawImage` , acesta deja are grija de astfel de situații și acționeaza 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 performanță.

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

Metodele sunt rapide, scurte și ușor de citit.

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

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

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

Clasa `Triangle Vectorizer`

Acuma voi prezenta mai multe detalii despre implementarea vectorizatorului pe baza de triunghiuri.

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

Aeste nevoie de o valoare aleatoare pentru ca paternele care sunt generate sa nu arate ca un desen „tehnic” . Fiind aleator exista o șansa ca tringhiurile să se aranjeze în paterne mai organice.

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

Aceasta dar tine o referinta către ultima lista de tringhiuri care a fost generata de către un `Job`. Este posibil sa renunț la aceasta implementare dacă nu ii găesc folos , pentru ca lista de triunghiuri este totuși considerabila ca și mărime.

```
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();
}
}

```

Funcția asta o s-o descriu decât o singură dată aici deoarece ea este practic identică cu celelalte vectorizatoare.

Motivul de ce am folosit un obiect `jobLock` pentru a îmi asigura siguranța datelor este deoarece obiectul `lastJob` mai trebuie accesat și din alta parte, astfel având nevoie de un obiect comun ca să creeze sincronizarea.

După ce mi-am asigurat accesul unui singur thread la date importante, verific dacă există vreun Job în referința `lastJob`. Dacă aceasta există, este posibil ca să și ruleze, așa că setez flag-ul *canceled* pe `true`. Thread-ul de tip job nu se va opri instant, trebuie așteptată oprirea ei folosind `lastJob.join()`. Chiar și așa, funcția care rulează pe Job trebuie să verifice în mod constant valoarea flag-ului *canceled*. Voi discuta despre asta mai jos.

Acum că sunt sigur că numai există niciun Job thread rulând, care ar putea să corupă date, sau să țină referința la structuri de date nedorite, *lastJob* primește referința unui nou thread Job.

Funcția `setIsDone()` este folosită în benchmarking pentru a anunța atunci când un Job a terminat, și poate să înceapă unul nou în siguranță. Deoarece aici practic job-ul a fost înlocuit cu altul nou, îl setez pe fals pentru siguranță.

Ca ultimă asigurare înainte să pornească noul thread, resetez valoarea atomică `aproxCompletedPixelCount` 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

ca vectorizatorul a terminat ceva.

Metoda aceasta este similara cu cele din celelalte vectorizatoare deci nu este necesar sa îl mai explic din nou mai jos.

```
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 afisa userului rezultatul aproximativ (si superficial) al vectorizarii. Mai întâi se verifica dacă am referinta către obiectul ImagePanel , sau dacă vectorizarea doar este în Benchmark-ing.

Am dorit sa masor timpul de desenare în benchmark deoarece o imagine mai mare și un threshold mic începe sa ingreuneze semnificativ viteza de creare a imaginii de prezentare după vectorizare.

Nu am văzut observat niciun overhead semnificativ în crearea obiectului Graphics2D la fiecare a chemare a metodei drawTraignles() așa ca l-am lăsat așa.

După aceea interez prin toată lista de obiecte Triangle, la fiecare pas setand culoarea triunghi-ului , și umplerea figurii geometrice definite de către obiectul triunghi (t.getPath()).

Dacă vectorizarea a fost făcută cu scopul masurarii performantei sale , nu se va afisa nimic userului.

```
protected void constructStringSVG()
```

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

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

Deoarece exista o șansa foarte mare ca userul sa încerce mai multe threshold-uri pana când este mulțumit , am create un obiect StringBuilder pe care îl resetez la zero de fiecare data când încep 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 bufferului sau intern.

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

Prima linie pe care o conține SVG-ul va descrie în ce format este scris , și ce latime și înălțime trebuie sa aibe fișierul imagine.

```
DecimalFormat decimalFormat = new DecimalFormat("#.##");
```

Acesta este un obiect destul de interesant. Atunci când se crează un string folosind acest format și un număr real, el va încerca să îl reprezinte ca în acel format.

El va afișa de exemplu pentru numerele $3.14159 \Rightarrow 3.1$; $42 \Rightarrow 42$; $621.59 \Rightarrow 621.6$.

Încă un lucru interesant este că formatul pe care l-am ales pune întotdeauna un punct, nu o virgulă, atunci când vrea să scrie partea fracționară. Dacă partea întreagă și partea fracționară ar fi fost separate de o virgulă, atunci formatul SVG nu ar fi fost scris corect, deoarece el folosește virgulă pentru a separa coordonate.

Am ales să afișeze doar o singură zecimală deoarece pixelii originali din care a fost construită poza aveau numai coordonate întregi, așa că nu am nevoie de precizie mai mare de atât. Astfel mai reduc umplerea din mărimea fișierului față de cazul în care foloseam două sau mai multe zecimale.

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

Există o mică problemă cu vectorizarea prin triunghiuri. Din cauza erorilor de calcul cu float încep să apară mici spații între triunghiurile care ar trebui să fie vecine. Atunci când un utilizator deschide imaginea în format SVG într-un editor sau într-un browser, el/ea poate să observe numeroase linii albe între triunghiuri. Efectul este și mai evident dacă mărește poza folosind zoom.

Așa că aplic o proprietate comună la toate triunghiurile care o să le adaug, toate vor avea perimetrul sau desenat cu 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 una peste alta prea mult.

```
for (Triangle t : lastSavedTriangleList) {

    svgStringBuilder.append(String.format("<path d='M%s,%sL%s,%sL%s,%sZ' fill='##06X'
stroke='##06X' />\n",
        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 formatul SVG.

Am ales tag-ul path pentru a mai reduce din numărul de caractere total. Litera 'M' are rolul de a muta „pensula” fără să deseneze la coordonatele absolute definite de următoarele două numere. Litera 'L' are rolul de a trasa o linie de la ultima poziție a pensulei la următoarele coordonate absolute. Iar litera 'Z' este folosită pentru a închide figura geometrică descrisă. Atributul *fill* și *stroke* primesc ca valoare o culoare definită asemănător stilului HTML. Format-ul *%06X* definește un număr hexadecimal în care dacă numărul sau de cifre nu depășește 6 va fi umplut în stânga cu cifre de 0 până când are 6 cifre.

Am ales să scriu în format RGB hexadecimal deoarece este cea mai scurtă opțiune la dispoziție. A trebuit să am o precauție în plus în definirea culori, așa că păstrez numai biții care sigur au numai valori pentru RGB, și fără Alpha. Operația aceasta de filtrare a bitilor o fac prin

AND binar cu numărul 0xfffff.

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

La finalul fisierului SVG , se adauga sfarsiturile de tag pentru proprietatea care da la toate triunghiurile o linie grosa , și tag-ul de sfârșit al intregului format SVG.

O să presupun aici ca JIT-ul o să mai optimizeze codul atunci când sunt create atât de 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 initializat cu un buffer de 2000000 de bytes , ceea ce ar trebui să fie suficient pentru start, iar obiect-ul *gzos* am decis sa îl initializez aici.

Obiectul *gzos* funcționează folosinduse 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 genereaza un obiect String destul de mare ceea ce ar putea deranja performanța aplicației , dar se pare ca JIT-ul se descurca destul de bine. După ce string-ul a fost scris in *gzos* , apelez explicit *flush()* pentru a ma asigura ca procesul de comprimare s-a efectuat cu succes.

Continul vechi al lui *svgzStringBuilder* este resetat , și apoi conținut 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 va tine ca referinta lista de triunghiuri la care lucrează thread-ul, dacă operația de

vectorizare se termina cu succes , referinta aceasta va fi copiată mai târziu în *lastSavedTriangleList*.

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

```
public void run()
```

În urmatoarele paragrafe voi explica continutul metodei run.

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

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

Din prima linie se verifica dacă valoarea flag-ului *canceled* a fost deja 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.

Cele doua 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 doua fire de execuție. Una pe cea curenta , una pe un thread nou.

Initializez doua liste dublu inlantuite care au rolul de a depune rezultatele celor doua linii de execuție. Motivul de ce am ales sa folosesc LinkedList în loc de ArrayList , este deoarece aceste lista vor avea foarte multe operatii de adaugare la final. Un ArrayList va produce probleme de performanța atunci când este upuizat bufferul sau intern de la prea multe adaugari, dar LinkedList nu va avea aceeași problema.

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

Dacă dintr-un motiv vectorizarea pe *t2* a reușit sa termine mai repede , trebuie să mă asigur ca celalalt 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 pana când acesta termina , sau dacă thread-ul deja a terminat , *join()* va returna imediat.

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

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

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

Dacă cele doua linii de execuție au executat cu succes *recTriangulation* , trebuie culese informațiile adunate de către ele într-o lista comuna. Prima data ma asigur ca ArrayList-ul *triangles* va avea o zona de memorie suficient de mare pentru a putea include toate datele dintr-o data. După aceea ele sunt adaugate una câte una.

Mai fac încă o verificare a flag-ului *canceled* pentru ca tocmai s-a executat câteva linii care au o durata considerabila 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 ca s-au colectat toate triunghiurile într-o lista comuna, se seteaza lista de triunghiuri a intregului obiect TriangleVectorizer 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.

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

Aceasta metoda cauta în mod recursiv toate triunghiurile care se afla în obiectul *triangle* și le depoziteaza 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 initializez 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 comparatia 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 ca 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. Deasemenea verific dacă variabila *fail* a devenit true , deoarece este posibil ca un pixel sa fi fost prea diferit pentru 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 cu primele două segmente ale triunghiului, deci cu siguranță 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 extind marginile spre numere întregi fără să li se dea voie să iasă din segmentul $[xMin, xMax]$.

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

Se verifică 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 verifică variabila *fail* dacă a devenit adevărat și 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 incrementeaza contorul de pixeli , și se adaug valorile culorilor la sumele lor corespunzatoare. Acestea vor fi folosite mai târziu pentru a calcula adevărata medie a culorilor.

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

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

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

Se calculează distanța Manhattan bazata pe valorile culorilor RGB dintre pixelul current și media aproximativa a culorilor.

```
if(m > threshold)
```

```
fail = true;
```

Dacă valoarea m este mai mare decât threshold, înseamnă ca s-a descoperit un pixel care este prea diferit de media aproximativ calculata , deci analiza triunghiului este oprita.

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

```
if(count==0)
```

```
return;
```

```
else {
```

```
    rTotal /= count;
```

```
    gTotal /= count;
```

```
    bTotal /= count;
```

```
}
```

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

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

```
    triangle.color = 0xff000000 | (rTotal<<16) | (gTotal<<8) | bTotal;
```

```
    triangles.add(triangle);
```

```
    int x = aproxCompletedPixelCount.addAndGet((int)Math.ceil(triangle.area * 100));
```

```
    updateDetails(String.format("Progress : %.1f%%",1.f*x/area));
```

Se verifica dacă vreunul dintre pixelii verifica a eşuat testul threshold, dacă da acel triunghi nu va fi adaugat. A trebuit sa 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 lasa numai anumite triunghiuri sa treacă. Dacă aria triunghi-ului este mai mica sau egal 3 , el va avea aproximativ cel mult 3 pixeli întregi deci nu mai rost sa mai folosesc conditia stricta de threshold.

```
}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 nu a fost respins , acesta va trebui separat în doua. Pentru ca triunghiurile sa nu devină prea subtiri sau sa aibe un unghi prea obtuz, taierea 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 numar aleator r din mulțimea $[0.2,0.8]$. Folosind valoarea r apoi despart triunghiul cel mai mare în doua mai mici , având grija sa tai pe latura cea mai lunga. Punctul de despărțire este ales prin interpolarea celor doua puncte ale laturii cele mai mici cu valoarea numarului r .

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

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

Exista posibilitatea ca după taierea triunghiului , unul dintre ele o să fie foarte foarte mic. Dacă aria triunghiului a devenit mai mic decât o jumătate de pixel , nu mai are rost să fie luat în considerare.

Atunci când un triunghi este considerat suficient de mare , se continua procesul recursiv , pasand referinta către containerul de triunghiuri.

Clasa Square Vectorizer

Acuma voi prezenta mai multe detalii despre implementarea vectorizatorului pe baza de dreptunghiuri.

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

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

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

Aceasta doar tine o referinta către ultima lista de dreptunghiuri care a fost generata de către un Job. Este posibil sa renunț la aceasta implementare dacă nu îi găsesc folos , pentru ca 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 similară cu funcția *drawTriangles* din *TriangleVectorizer*. Am folosit metoda *fillRect* în loc de *fillPath* deoarece aceasta este specializată pentru a desena dreptunghiuri deci există o șansă foarte mare și fie mai rapidă. Motivul de ce am adăugat 1 la lățime și lungime este deoarece metoda scade cu 1 aceste valori înainte să deseneze.

```
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 (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&0xffffffff));
}
```

Am decis să folosesc tag-ul *rect* pentru a face mai natural de citit, și manipulat într-un editor de fișiere vectoriale precum Inkscape. Dacă aș folosi un tag mai compact precum *path* atunci dimensiunea fișierelor în format SVG ar scădea foarte mult dar s-ar pierde intuitivitatea.

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

Subclasa Job a clasei SquareVectorizer

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 copiată 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 urmatoarele 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 triunghiurile care se afla în obiectul *triangle* și le depoziteaza 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 initializez 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 aicea 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 comparatia cu tresahold-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 ca 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.Deasemenea

verific dacă variabila *fail* a devenit true , deoarece este posibil ca un pixel sa fi fost prea diferit pentru threshold.

```
flag = 0;

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

if (flag == 1) {

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

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

Procesul de calculare și respingere este unul rapid și eficient. Variabila *flag* are rolul de număra câte puncte de intersecție s-au găsit pana 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 (x1,y1)(x2,y2). În funcție de valoarea lui *flag* al doilea punct de intersecție va fi notat fie pe i0 fie pe i1.

```
if (flag == 1) {

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

Este posibil ca *flag* sa fi rămas egal cu 1, deci asta înseamnă ca s-a descoperit decât un singur punct de intersecție pana acum cu primele doua 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 extind marginile spre numere întregi fără sa li se dea voie sa 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 itereaza de la stânga la dreapta toate coordonatele x între cele doua puncte de intersecție găsite. La fiecare pas se verifica variabila *fail* dacă a devenit adevărat și să se oprească procesarea triunghiului. Deoarece acest *for* este destul de strâns , am decis sa pun și aici încă o verificare pentru flag-ul *canceled*.

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

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

```
m = Math.abs(rMed - redOrig(x, y)) +  
  
    Math.abs(gMed - greenOrig(x, y)) +  
    Math.abs(bMed - blueOrig(x, y));
```

Se calculează distanța Manhattan bazata pe valorile culorilor RGB dintre pixelul current și media aproximativa a culorilor.

```
if(m > threshold)  
  
    fail = true;
```

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

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

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

```
if(!fail || triangle.area<=3){  
  
    triangle.color = 0xff000000 | (rTotal<<16) | (gTotal<<8) | bTotal;  
    triangles.add(triangle);  
    int x = approxCompletedPixelCount.addAndGet((int)Math.ceil(triangle.area * 100));  
    updateDetails(String.format("Progress : %.1f%%",1.f*x/area));
```

Se verifica dacă vreunul dintre pixelii verifica a eşuat testul *threshold*, dacă da acel triunghi nu va fi adaugat. A trebuit sa 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 lasa numai anumite triunghiuri sa treacă. Dacă aria triunghi-ului este mai mica sau egal 3 , el va avea aproximativ cel mult 3 pixeli întregi deci nu mai rost sa mai folosesc conditia stricta de threshold.

```
} 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 nu a fost respins , acesta va trebui separat în doua. Pentru ca triunghiurile sa nu devină prea subtiri sau sa aibe un unghi prea obtuz, taierea 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 numar aleator r din mulțimea [0.2,0.8]. Folosind valoarea r apoi despart triunghiul cel mai mare în doua mai mici , având grija sa tai pe latura cea mai lunga. Punctul de despărțire este ales prin interpolarea celor doua puncte ale laturii cele mai mici cu valoarea numarului r:

```
if(t1.area>0.5)recTriangulation(t1,triangles);  
  
if(t2.area>0.5)recTriangulation(t2,triangles);
```

Exista posibilitatea ca după taierea triunghiului , unul dintre ele o să fie foarte foarte mic. Dacă aria triunghiului a devenit mai mic decât o jumătate de pixel , nu mai are rost să fie luat în considerare.

Atunci când un triunghi este considerat suficient de mare , se continua procesul recursiv , pasand referinta către containerul de triunghiuri.