
**Treball de Fi de Grau**

**GRAU D'ENGINYERIA INFORMÀTICA**

**Facultat de Matemàtiques**

**Universitat de Barcelona**

---

# Implentació de l'algorisme Poisson image editing a la plataforma Android

---

**Marc Sánchez Matas**

Director: Lluís Garrido Ostermann

Realitzat a: Departament de Matemàtica Aplicada i Anàlisi. UB

Barcelona, Gener de 2015

# Abstract

This project's aim is to develop an Android application that implements a Poisson image editing technique, more specifically, the seamless cloning part.

The implemented algorithm's goal is to paste an area of an image onto another trying to disguise the patch inserted and making it believable to the human eye that the patch belongs to the target image.

In order to make this possible some local changes have to be made through the patch to minimize the slope between the two images but simultaneously not losing any information contained in the patch. Such information is the gradient, and it is preserved through all the process. Thus, we call this process a gradient-guided interpolation. This algorithm is defined on the paper *Poisson Image Editing* written by Perez et Al. written in 2003.

Along this project we will see similar image processing techniques and discuss the pros and cons of their usage.

Afterwards, we will focus on the Poisson equations for this problem and the several ways to solve it. We will be using a Gauss-Seidel iterative solver with successive over-relaxation.

Then, the design and implementation of all the Android application will be discussed.

Finally the results obtained by developing this project will be evaluated, ranging from the efficiency or the effectiveness of the algorithm to the plausibility of this program as a published app.

# Index

# 1. Introducció

El *Poisson blending* o *Poisson image editing*, és una algorisme que pretén minimitzar l'error visualment perceptible que resulta d'enganxar un fragment d'una imatge en una altra. La implementació d'aquesta minimització és el nucli d'aquest projecte. Podem observar aquest problema que comentem a la figura 1. L'altra gran part és l'aplicació Android que la conté.

Figura 1: Resultat visualment perceptible d'enganxar un fragment d'una imate en una altra

L'algorisme implementat és el que es pot trobar a l'article *Poisson image editing* de Patrick Pérez, Michel Gangnet, i Andrew Blake presentat a SIGGRAPH 2003.

## 1.1. Objectius

L'objectiu principal d'aquest projecte és implementar l'algorisme de *Poisson image editing* en forma d'aplicació Android, per tal de fer tal cosa, haurem d'assolir diversos objectius: primer de tot entendre el problema i veure les maneres de solucionar-lo. I buscarem algorismes similars. També ens caldrà aprendre a desenvolupar aplicacions Android i a fer ús de l'entorn de desenvolupament Android Studio.

Tot i que la idea principal és implementar el *seamless blending*, també pretenem implementar la variant del *mixed seamless blending*.

L'objectiu final és analitzar tant els resultats obtinguts visualment per part de l'algorisme com l'eficiència d'aquest.

## 1.2. Motivació

Des de que ara farà un parell d'anys que em vaig iniciar en el mon del processament d'imatge amb l'assignatura de visió artificial, em va sorprendre el munt de possibilitats que hi havia i em va fer decidir a cursar l'assignatura de processament d'imatge. La qual vaig trobar molt motivadora i em va aportar bons coneixements, que m'han sigut molt útils en la realització d'aquest projecte. És la principal raó que em va decantar cap a la tria d'aquest projecte un cop el vaig veure a la llista de propostes.

La proposta de projecte inicial era realitzar un plug-in per al programa d'edició d'imatges i de codi obert GIMP. Però durant els inicis del desenvolupament vaig trobar diversos inconvenients que no m'acabaven de motivar. Per una banda, ja existia un plug-in que ja implementava entre moltes altres aquesta funcionalitat, i per l'altra la mida enorme de l'API del GIMP i la lentitud que comportava cada compilació i execució. Vaig exposar els meus problemes al Lluís, i em va suggerir implementar el mateix algorisme però en una aplicació Android, cosa que em va semblar adient i vaig iniciar el desenvolupament.

## 1.3. Estructuració del projecte

Primer de tot farem una petita introducció passant per conceptes previs a l'explicació, i veurem com certs algorismes s'aproximen al mateix problema.

A continuació farem un estudi en profunditat sobre el problema a solucionar i què pretén l'algorisme. I veurem les diverses maneres de com trobar solució a les equacions obtingudes.

Continuarem amb una petita introducció a Android, i veurem els detalls de disseny de l'aplicació.

En aquesta part primer veurem l'aplicació des del punt de vista de implementació de la interfície, i després ja passarem a veure la part de la implementació de l'algorisme amb detall

A continuació veurem els resultats obtinguts per part del *Poisson image editing*, tant des d'un punt de vista visual com en termes d'eficiència.

I Finalment traurem conclusions sobre el projecte en general.

## 1.4. Definicions prèvies

Per entendre en detall l'explicació d'aquest algorisme, hi haurà un seguit de conceptes als que faré referència amb recurrència que caldrà que definim prèviament.

Gradient

El gradient serà de vital importància per a la resolució del nostre problema, ja que és on resideix gran part de la informació visual que percebem.

El *gradient domain image processing* és una estratègia de la que farem ús en la implementació del *Poisson image editing*, que considera les diferències entre els píxels adjacents, és a dir, en comptes de simplement considerar els valors dels píxels, tenim en compte la derivada d'aquesta. A continuació, podem observar la fórmula del gradient.

$$\nabla f = \frac{\partial f}{\partial x}\hat{x} + \frac{\partial f}{\partial y}\hat{y}$$

Matemàticament, el gradient d'una funció bidimensional ens donaria com a resultat dues components: el gradient en direcció horitzontal i en direcció vertical, és a dir que obtindríem les diferencies d'intensitat.

Les imatges, però, signifiquen una discretització d'aquestes funcions, assumim que un mostreig periòdic i equidistant obté els valors sense trencar la validesa del procés.

En el món del processament d'imatge ens aproximem a aquest ideal de gradient realitzant una convolució de una imatge amb diversos filtres, que depenent de l'ús que en vulguem fer, seran més o menys adients.

Sempre que veiem l'operador ∇, estarem parlant del gradient.

Tal com hem dit, l'algorisme de *Poisson image editing* es centra en la importància del gradient per a guiar-nos cap a una solució visualment més acceptable.

Matriu de convolució

Les imatges amb les que treballem són un conjunt de píxels agrupats bidimensionalment en coordenades cartesianes. Sobre aquests conjunts de píxels podem realitzar diverses operacions per extreure'n diverses característiques.

Una matriu de convolució és aquella que aplicarem sobre una determinada imatge per extreure'n un determinat conjunt de característiques. S'acostumen a aplicar centrats sobre cada píxel de la imatge.

## 1.5. Estat de l'art

El processament d'imatges és un camp molt ampli dins de la informàtica en el que tractem una o diverses senyals per a transformar-les. Aquesta senyal generalment serà bidimensional i discretitzada.

El processament d'imatges és una tecnologia en constant evolució, en la que és molt difícil estar al dia, tot i així parlarem del nostre algorisme, el *Poisson image editing* i d'altres algorismes que enfoquen el problema o situacions similars des d'una altra perspectiva.

El nostre objectiu és fusionar part d'una imatge en una altra fent que aquesta unió sigui poc o gens perceptible a l'ull humà.

### *Mean-Value Seamless cloning*

És un algorisme presentat el 2009 a SIGGRAPH, molt similar al Poisson image editing, però que en comptes de solucionar un sistema d'equacions poc dens, els píxels interiors s'interpolen amb un pes ponderat dels píxels més exteriors.

En certes ocasions, aquest algorisme proporciona millor solució que el Poisson image editing, tot i que en certs casos també es produeix una contaminació del resultat. És més ràpid i més paralelitzable que la implementació del Poisson image editing. Aquesta implementació no invalida la utilitat del nostre algorisme, més aviat, poden conviure perfectament i ser intercanviables depenent de la situació.

### *Piràmide Laplaciana*

Consisteix en descomposar les dues imatges repetidament en versions més reduïdes d'aquestes, fins arribar a un baix nivell, i allà realitzar la composició de la imatge, després cal anar reconstruint successivament fins arribar a la imatge original.

No presenta uns resultats excessivament bons, i la majoria d'algorismes donen millors resultats.

### *Covariant cloning*

Creat per Todor Georgiev d'Adobe. És el *healing brush* del programa d'edició d'imatge Photoshop. Planteja el problema de forma similar al Poisson image editing però no fa ús de la laplaciana. Funciona millor en casos en que hi ha una gran diferència de lluminositat.

Aquesta implementació es basa en que nosaltres no veiem el contingut dels píxels directament, sinó una interpretació del conjunt de píxels. Afirma que l'ús de la laplaciana és una generalització d'aquest algorisme.

# 2. Poisson Image Editing

## 2.1. Introducció

Aquest algorisme utilitzarà tres imatges diferents: una imatge destí  on enganxarem una zona de una imatge origen, i aquesta zona vindrà definida per una màscara.

Pel que fa a la nomenclatura que veurem a les fórmules, tindrem:

Com podem veure a la figura 2, la imatge inicial (que denominarem S), és on tindrem un seguit de valors coneguts(f*) i tot un seguit de incògnites que haurem de calcular(f).

Les incògnites a calcular pertanyen a la regió definida com **Ω,** que inicialment contindrà els valors d'una regió de la imatge origen, definida per la màscara i tot això situat sobre la imatge destí. Tot aquest conjunt de valors, constitueix la imatge de punt de partida S.

El valor dels píxels adjacents al límit de **Ω** serà denominat **δΩ**, que no pertany a **Ω** i que per tant els seus valors finals són coneguts.

La zona V conté el camp vectorial relatiu a la zona enganxada g.

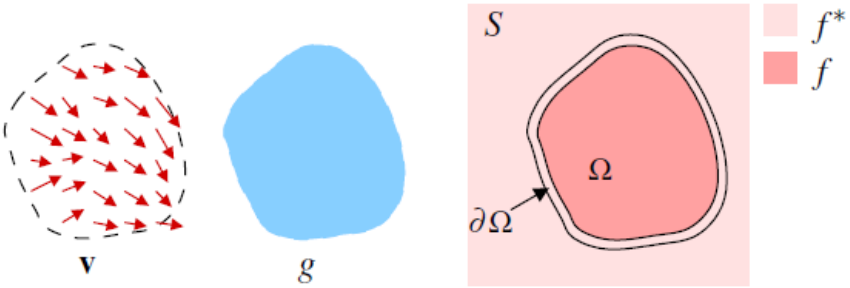


Figura 2:Parts de la imatge

L'objectiu del Poisson blending és anivellar la intensitat de la part de la image origen amb la intensitat del la imatge destí, però que no es produeixi una pèrdua d'informació, per això és de tanta importància l'ús del gradient. Com podem veure a la figura 3, el nostre interès és transferir una regió de la imatge origen a la imatge destí. I a la figura 4, podem veure com el que realment ens interessa és aquesta diferència de nivell o intensitat entre píxels.

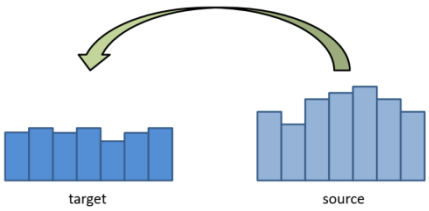


Figura 3:imatge orígen i destí íntegres

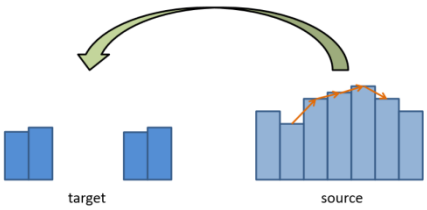


Figura 4: només ens interessa el gradient de l'origen

Tal com estem dient, ens trobem davant d'un problema de minimització, hem de rebaixar el nivell de la zona imatge origen que enganxem, al nivell de la zona destí i alhora mantenir les diferències de gradient.

Tot això ho podem observar a la figura 5

$$\min_f \iint_\Omega |\nabla f - \mathbf{v}|^2 \text{ with } f|_{\partial\Omega} = f^*|_{\partial\Omega},$$

Figura 5: eqüacions que defineixen aquesta minimització

A l'esquerra d'aquesta equació de la figura 6, tenim l'equació diferencial parcial a satisfer, que ens defineix la minimització.

Recordem que: **f** és la incògnita a calcular, en el context del Poisson image editing, **v** és el gradient de la zona a enganxar **G**. Observem doncs que la part de l'esquerra ens imposa trobar un gradient similar al de **G**. I a la part dreta tenim la restricció: volem que a la vora de f, els valors siguin els mateixos que a $\boldsymbol{\delta\Omega}$. Aquesta restricció s'anomena condició de frontera de Dirichlet.

Tot això és vist des del marc teòric, i a continuació veurem com discretitzar aquesta equació.

## 2.2. Discrete Poisson Solver

Ara traslladant-nos a una graella bidimensional, caldrà que definim alguns conceptes abans de prosseguir amb l'explicació:

- Np serà el conjunt de 4 píxels connexes al píxel p

- <p,q> serà un parell de píxels tal que q ∈ Np

- $v_{pq}$ és el vector de projecció sobre p i q

Segons l'article original, la discretització del nostre problema la podem visualitzar a la següent equació

$$\min_{f|_\Omega} \sum_{\langle p,q\rangle \cap \Omega \neq \emptyset} (f_p - f_q - v_{pq})^2, \text{ with } f_p = f_p^*, \text{for all } p \in \partial\Omega,$$

Ens diu que: per cada punt de **Ω** hem de minimitzar la diferència entre un punt, els seus 4 veïns connexes i el v. On mantindrem aquesta condició de frontera de Dirichlet de que per cada punt d'***δΩ*** el valor s'ha de mantenir igual.

En la versió discretitzada, aquest procés de considerar els 4 píxels connexes és l'equivalent a convolucionar un filtre laplacià sobre cada píxel de **Ω.** A les figures 6 i 7 podem veure aquesta idea: p és el píxel en el que ens centrem en aquell moment i q seran els seus veïns. Que si volem fer el sumatori d'això és equivalent a fer 4 cops p, menys un cop cada un dels veïns.

| | | |
|---|---|---|
| 0 | −1 | 0 |
| −1 | 4 | −1 |
| 0 | −1 | 0 |

Figura 6: "filtre" aplicat sobre p

| | | |
|---|---|---|
| 0 | q | 0 |
| q | p | q |
| 0 | q | 0 |

Figura 7: p i els seus veïns q

I $v_{pq}$ serà igual a $g_p$ - $g_q$, és a dir la intensitat de color per cada color que tinguem en cada píxel. Sent p el píxel central i q els veïns.

$$\text{for all } p \in \Omega, \quad |N_p| f_p - \sum_{q \in N_p \cap \Omega} f_q = \sum_{q \in N_p \cap \partial\Omega} f_q^* + \sum_{q \in N_p} v_{pq}.$$

La solució d'aquest problema es veu satisfet per aquesta equació:

Que aïllant farà que obtinguem l'equació següent:

$$|N_p| f_p = \sum_{q \in N_p \cap \Omega} f_q + \sum_{q \in N_p \cap \partial\Omega} f_q^* + \sum_{q \in N_p} v_{pq}.$$

On fp és el valor a calcular, Np és conegut i generalment igual a 4, fq és desconegut, ja que com que pertanyen a **Ω** són tot el conjunt d'incògnites que haurem d'anar trobant.

Si desenvolupem aquestes equacions segons la nostra nomenclatura obtenim aquesta equació, sent **d** destí i **o** l'origen.

4*d(i,j) - d(i-1, j) - d(i+1, j) - d(i, j-1) - d(i, j+1) = 4*o(i,j) - o(i-1, j) - o(i+1, j) - o(i, j-1) - o(i, j+1)

També implementarem una variant anomenada *mixed seamless cloning*, en la que considerem també el gradient de la imatge destí, cosa que farà que en determinats casos com ja veurem més endavant obtindrem resultats visualment millors. Ja que farà que parts que tenien un quantitat elevada d'informació i quedaven eliminades, podran ser visualitzades.

Són mètodes complementaris i el seu ús dependrà del tipus de imatge que vulguem obtenir. A diferència del *seamless cloning* normal, en el que només ens interessa la intensitat de p i q en la imatge origen, aquí farem una comparació (parlant en termes de valor absolut) amb també la intensitat de p i q però de la imatge destí. Podem veure aquesta formulació a la figura 8.

$$v_{pq} = \begin{cases} f_p^* - f_q^* & \text{if } |f_p^* - f_q^*| > |g_p - g_q|, \\ g_p - g_q & \text{otherwise,} \end{cases}$$

Figura 8:*Mixed Seamless Cloning*

El conjunt d'equacions que defineixen aquest problema acaba resultant en un sistema de N equacions amb N incògnites, amb unes característiques clarament definides, ja que obtindríem una matriu N*N i que té molts pocs elements no nuls, per tant, no cal alarmar-se per la mida d'aquest sistema d'equacions, ja que tenim diverses maneres de solucionar-lo.

## 2.3. Possibles solucions al problema

Tot i que un sistema de N equacions amb N incògnites pot ser solucionat de diverses maneres, els autors de l'article original, proposen una aproximació o bé utilitzant el Multigrid V-cycle o bé l'altre és el mètode Gauss-Seidel.

No entraré en gaire detall en l'explicació del Multigrid V-cycle, ja que en aquest projecte hem optat per la implementació del mètode Gauss-Seidel.

### 2.3.1. El V-cycle Multigrid

És un algorisme del tipus divideix i venceràs. Primer de tot obté una solució inicial per una graella n x n utilitzant una graella aproximada de mida n/2 x n/2, i així recursivament, fins a reduir-la a la mida adequada.

A continuació interpreta l'error com un una suma de funcions sinusoïdals de les diferents freqüències. L'error que haguem solucionat en una graella, resoldrà part de l'error en altres graelles, i així successivament fins arribar a la solució final

### 2.3.2. Mètode de Jacobi

Seria possible també una aproximació a aquest problema utilitzant el mètode de Jacobi, però Gauss-Seidel ens garanteix una convergència almenys igual de ràpida, i generalment més ràpida. Per tant l'ús del mètode de Jacobi no s'ha tingut en consideració.

### 2.3.3. Gauss-Seidel

És un mètode iteratiu per solucionar sistemes quadrats de N equacions lineals on partim d'una aproximació inicial i anirem iterant fins que considerem que hem arribat a la convergència.

El mètode Gauss Seidel pretén solucionar un sistema d'equacions de la forma:

**Ax=b,** on:

**A** és una matriu NxN poc densa, **b** és una matriu columna de Nx1 elements que conté valors coneguts i **x** és el conjunt d'incògnites a solucionar.

La matriu **A** la podem reformular com a $A=L_*+U$, on $L_*$ és una matriu triangular inferior incloent els elements de la diagonal, i U és una matriu triangular estrictament superior, on els elements de la diagonal són igual a 0. Això ens porta a les equacions següents:

$L_*\mathbf{x} = \mathbf{b} - U\mathbf{x}$ , que equival a $\mathbf{x}^{(k+1)} = L_*^{-1}(\mathbf{b} - U\mathbf{x}^{(k)}).$

I que finalment en podem extreure l'equació amb que fem les iteracions, tal com podem veure a la següent equació:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right), \quad i, j = 1, 2, \ldots, n.$$

Sent **x** el nou valor a obtenir, $a_{ii}$ és Np ,i **i** és el número que oscil·la entre 0 i N, i la **k** és la iteració en la que ens trobem.

Bàsicament, tindrem en compte totes les solucions parcials obtingudes, s'hagin actualitzat o no en aquella iteració.

Aquesta és l'equació genèrica del mètode iteratiu de Gauss-Seidel, més endavant quan veiem l'aplicació en la nostra implementació ja veurem el seu funcionament més detalladament.

#### 2.33.1. Gauss-Seidel amb *Successive Over-relaxation*

És una variant del mètode Gauss-Seidel, que ens garanteix una convergència més ràpida cap a la solució. Aquesta estratègia d'over-relaxation pot ser utilitzada amb qualsevol mètode iteratiu, ja que utilitza la solució anterior. A continuació podem veure aquesta eqüació:

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right), \quad i = 1, 2, \ldots, n.$$

Bàsicament, consisteix en donar una certa importància a la solució de la iteració anterior, donant-li un cert pes. Ajustant aquest valor, es traduirà en una més ràpida convergència del mètode iteratiu. A aquest pes l'anomenem W, i el seu valor oscil·larà entre 1 i 2.

Si la W és igual a 1, simplement, no tindrem en consideració la solució parcial anterior i estarem aplicant un Gauss-Seidel tradicional. La nova manera de calcular la següent iteració la podríem simplificar d'aquesta manera. La simplificació d'aquesta fórmula la podem observar a la figura 17.

$$x_{n+1}^{\mathrm{SOR}} = (1-\omega)x_n^{\mathrm{SOR}} + \omega f(x_n^{\mathrm{SOR}}).$$

La tria del paràmetre W no és pas trivial i té molt de pes en la velocitat de convergència del mètode. La w òptima per al nostre problema, la podrem calcular tal com es mostra a la figura 18:

$$\omega_{opt} = \frac{2}{1+\sqrt{1-\rho_J^2}}$$

Figura 9: Càlcul de la nostra W òptima

On $\rho_J = \cos\frac{\pi}{n+1}$, sent n el nombre d'elements de la matriu en una dimensió, és a dir, el nombre d'incògnites. Com hem dit aquesta manera de calcular la nostra W òptima només és vàlida per solucionar sistemes similars al nostre de la forma Ax=b.

# 3. Disseny i implementació

## 3.1. Android

És un sistema operatiu basat en el *kernel* de Linux, que va ser dissenyat principalment per a dispositius mòbils amb pantalla tàctil. El seu codi font és obert. L'estructura d'Android es basa en aplicacions que s'executen en un entorn Java sobre una màquina virtual amb compilació en temps d'execució, cosa que permet no haver de recompilar per cada dispositiu.

El Software Development Kit(SDK), ens proporciona tot un seguit d'eines que ens permetran el desenvolupament d'aplicacions. Tot i que generalment el desenvolupament d'aplicacions es fa en Java, també és possible crear llibreries natives en C i C++ utilitzant l'ANDK.

El sistema operatiu està basat en un seguit de capes aïllades, tal com veiem a la figura 10.

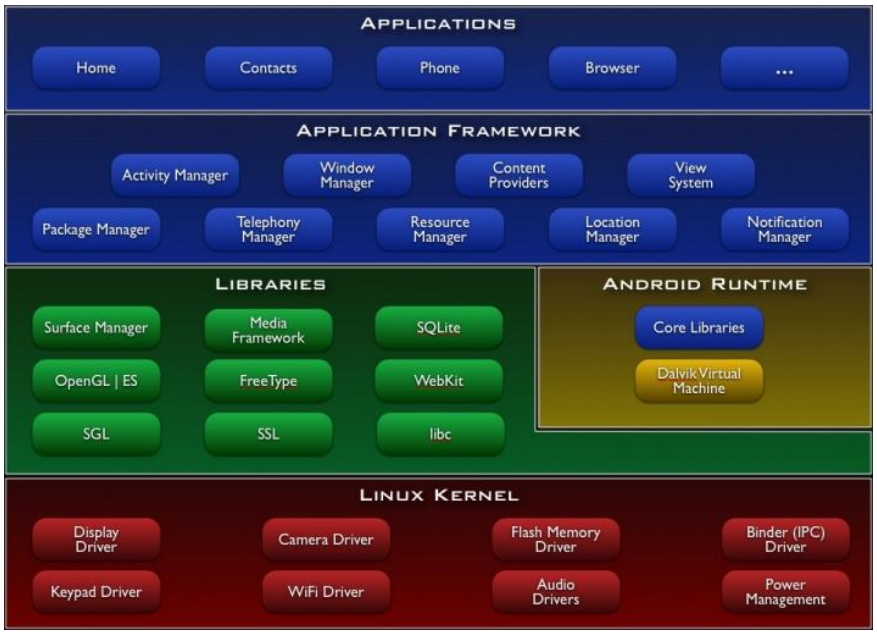


Figura 10: Sistema de capes d'Android

Les aplicacions Java que creem, estaran situades a la capa superior. Cada aplicació té un user id únic, cosa que fa que les aplicacions no puguin accedir als fitxers d'altres aplicacions.

La versió de Java utilitzada és lleugerament diferent al Java SE, ja que la part de gestionar la window o els graphics és diferent. Les llibreries swing o awt tampoc estan presents. I la part de RMI, també ha estat suprimida.

Les aplicacions Android estan constituïdes per activities, que són components amb els que els usuaris poden interactuar. Les aplicacions generalment tenen diverses activities per les que l'usuari anirà navegant. Totes les activities d'una aplicació hauran d'estar definides al manifest, que també ens dirà quina és la inicial. Totes les activities tenen un cicle de vida segons la seva situació en cada determinat moment, i hi haurà certs mètodes que s'aniran cridant. Aquests mètodes els podem sobreescriure d'acord els nostres interessos. Aquest conjunt de mètodes de l'activity els podem veure a la figura 11.

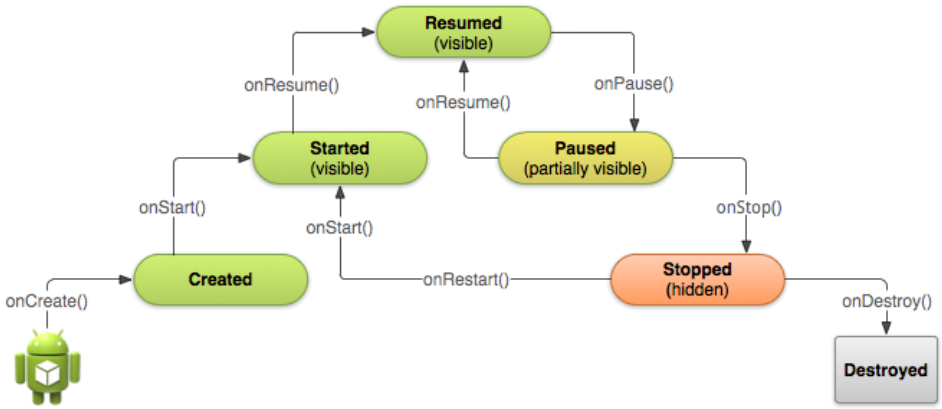


Figura 11: Cicle de vida de l'Activity

Les activities ens poden mostrar a la pantalla qualsevol element fill de la classe View, aquests elements els podem o bé crear programàticament o bé dins d'un fitxer XML.

Generalment utilitzarem aquests fitxers xml, ja que ens proporcionen una estructuració més còmoda i amb més possibilitats. Aquests elements dins del fitxer XML tindran una id, que podrem utilitzar des del fitxer Java per obtindre'n la referència.

Per anar navegant entre les diferents activities haurem d'utilitzar els intents, que executaran l'activity a la que vulguem anar. Podem enviar valors a les activity destí sempre que aquests no siguin massa grans.

La gestió dels events la podem realitzar de dues maneres: O bé afegint listeners sobre els components des de el fitxer Java, o bé quan definim els components al fitxer XML.

Sovint necessitarem tot un seguit de permisos per realitzar determinat tipus d'accions, que podrien cometre accions no desitjades. A l'usuari un cop vagi a instal·lar una aplicació se li requerirà acceptar aquests permisos per prosseguir. Aquests permisos poden ser tals com tenir accés a internet, fer ús de la càmera, emmagatzemar fitxers a memòria o realitzar trucades. Tot aquest seguit de permisos vindran especificats al manifest.

## 3.2. Eines i entorn utilitzat

Aquesta aplicació ha estat implementada al nivell d'API 16 d'Android, és a dir la versió 4.1 d'Android, coneguda també com a Jelly Bean.

Tot i que pels mètodes que utilitzem, seria possible executar aquesta aplicació en versions anteriors (testejat fins a l'API 10), s'havia de triar una plataforma mínima. Aquesta ha sigut triada per diverses raons:

Tot i que posar un nivell d'API elevat, limita la quantitat de dispositius antics que poden utilitzar una aplicació, hem de pensar que aquesta aplicació fa una considerable quantitat de càlculs, i que en dispositius més antics, que acostumen a tenir APIs més antigues, aquesta experiència podria ser més lenta i no tant agradable per l'usuari.

Podríem haver triat versions més actuals d'Android, però així garantitzem l'ús a un nombre més elevat de dispositius.

Actualment, al Gener del 2015, el percentatge de dispositius que poden fer ús de la nostra aplicació és del 85'1%. Aquest percentatge de dispositius Android capaços d'executar l'aplicació només podrà augmentar, ja que dispositius amb versions més antigues aniran desapareixent, i els nous sistemes seran compatibles amb aquesta versió. La distribució actual de versions d'Android és la que podem veure a la figura 12.

| Version | Codename | API | Distribution |
|---|---|---|---|
| 2.2 | Froyo | 8 | 0.4% |
| 2.3.3 - 2.3.7 | Gingerbread | 10 | 7.8% |
| 4.0.3 - 4.0.4 | Ice Cream Sandwich | 15 | 6.7% |
| 4.1.x | Jelly Bean | 16 | 19.2% |
| 4.2.x | | 17 | 20.3% |
| 4.3 | | 18 | 6.5% |
| 4.4 | KitKat | 19 | 39.1% |

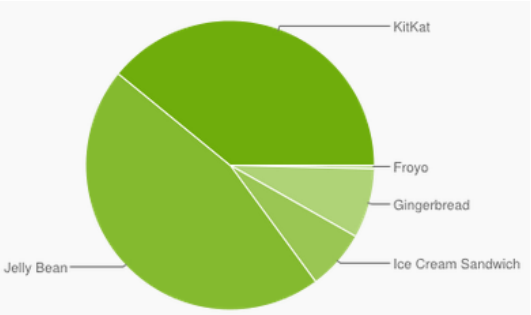


Figura 12: Distribució de versions d'Android actualment

La aplicació ha estat desenvolupada en l'entorn de desenvolupament Android Studio de Google, l'ús del qual és gratuït. També inclou el *virtual device manager*, que ens permet emular dispositius, tot i així, l'aplicació ha estat dissenyada i testejada en un dispositiu Nexus 5. Per a realitzar aquesta execució i debugament directament sobre el dispositiu ha estat necessari instal·lar els drivers usb específics.

## 3.3. Disseny

Primer de tot cal esmentar que hem triat que l'orientació de l'aplicació sigui sempre la mateixa, en forma landscape. Primer de tot perquè generalment les fotografies tenen aquesta orientació, i per temes de visualització és més còmode. I d'altra banda perquè al visualitzar les imatges les adaptarem a la mida de la pantalla.

El color de fons de tota l'aplicació és el mateix, #006699, el qual resulta visualment acceptable i no dificulta la lectura del text.

Part de les icones per els botons de l'aplicació han estat realitzades per mi mateix, i d'altres han estat obtingudes sota llicencia Creative Commons.

La icona de l'aplicació és la que veiem a la figura 13, és un petit joc de paraules amb la idea de que implementem un blending i la paraula *blender* en anglès és batedora.


Figura 13: Icona de l'aplicació

Totes les imatges de que fem ús, han estat obtingudes sota llicencia Creative Commons a www.flickr.com.

## 3.4. Casos d'ús

A la següent figura 14 podem veure el diagrama de casos d'ús de l'aplicació, cobreix totes les necessitats requerides per l'algorisme i a més d'altres per millorar l'experiència de l'usuari. Tots aquests casos d'ús els anirem veient més endavant.

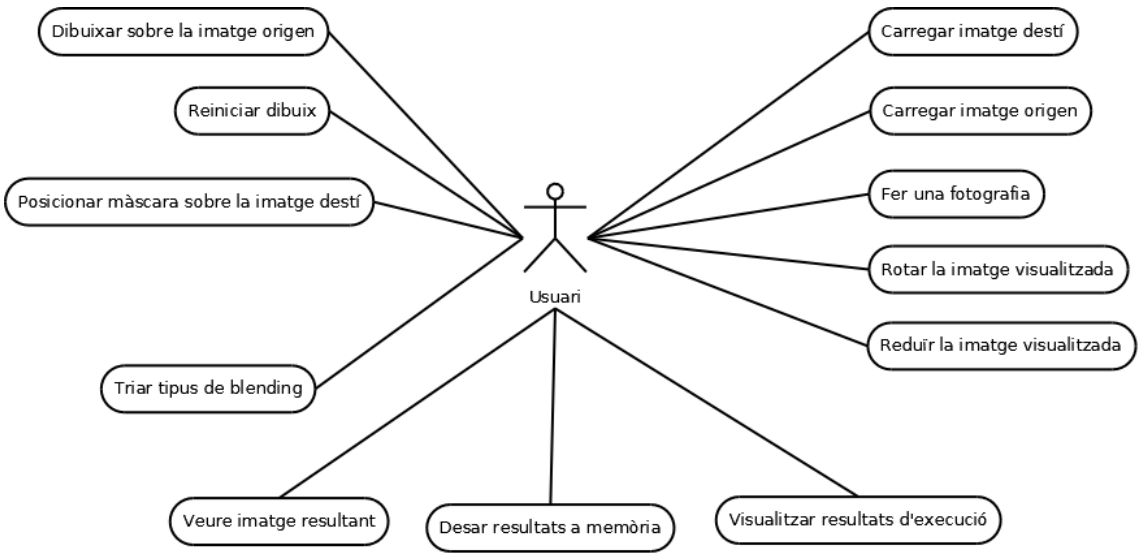


Figura 14: Diagrama de casos d'ús de l'apliació

## 3.5. Implementació de l'aplicació

La distribució i nomenclatura de classes a la que farem referència sovint és la que veiem a la figura 15.

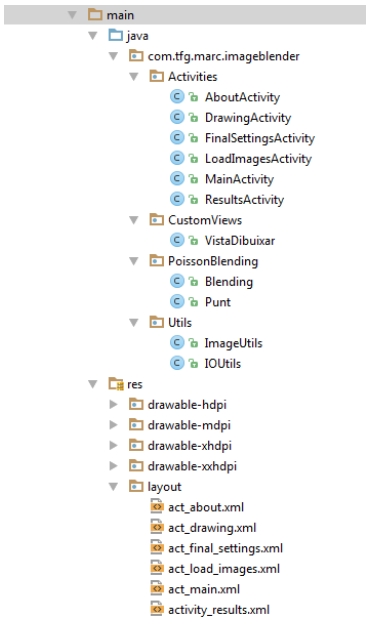


Figura 15: distribució de classes i layouts de l'aplicació

El projecte es composa d'altres fitxers, però que no seran rellevants per a la següent explicació.

Ara tractarem per separat el contingut de cada activity:

### *Activity inicial:*

És l'activity que veurem just iniciar l'aplicació, conté el nom de l'aplicació i dos botons que ens portaran a dues altres activities. Veiem el contingut d'aquesta a la figura 16.

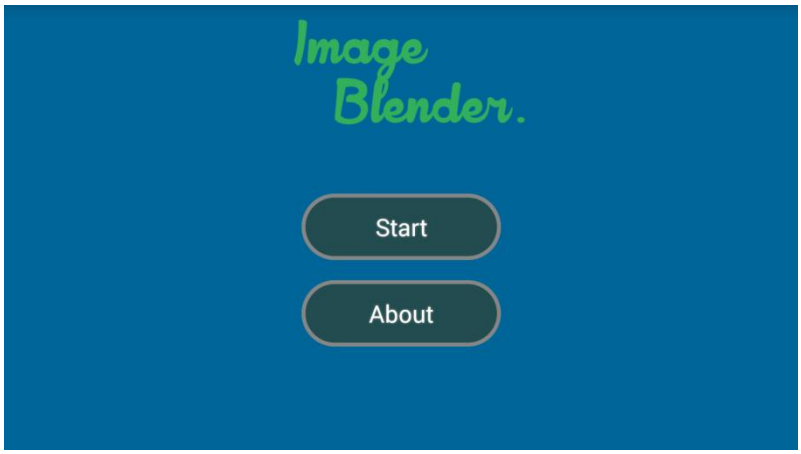


Figura 16: Pantalla d'inici

### *Activity About*:

Aquí veiem una petita part d'informació relativa al projecte. Es tracta simplement d'un camp de text. La podem veure a la figura 16

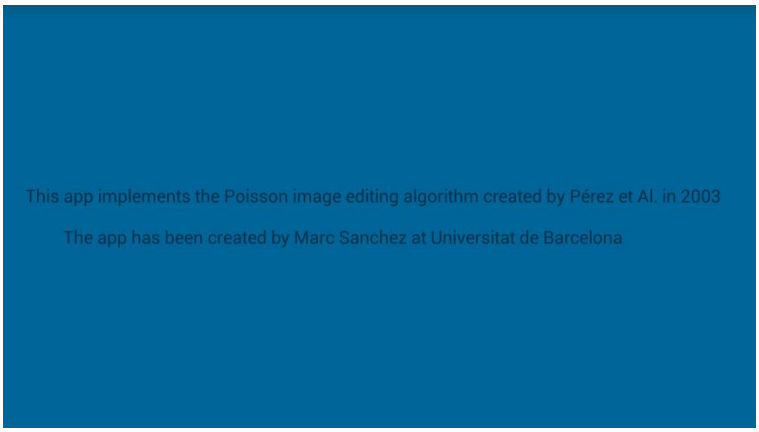


Figura 16: Activity About

### *Activity carregar imatges:*

Està localitzada al fitxer LoadImagesActivity.java i la layout que conté la vista és act_load_images.xml.

Al iniciar l'activity, veurem la pantalla relativament buida, i a la part superior esquerra instruccions sobre què hem de fer. Les icones dels botons són bastant auto-explicatives, d'esquerra a dreta tenim: triar imatge, reduir mida, rotar, i prosseguir. A la figura 17 podem veure el contingut de la pantalla.


Figura 17: Inici de l'Activity de carregar imatges

Tots els altres botons estaran deshabilitats fins que no carreguem una imatge, així que quan polsem aquest, s'obrirà un menú contextual  tal com veiem a la figura 18 que ens donarà dues opcions: seleccionar una imatge de la galeria, o bé fer una foto amb la càmera del dispositiu.

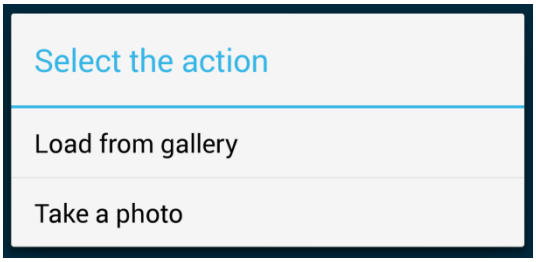


Figura 18: diàleg per carregar imatge

Si triem la opció de carregar des de la galeria, podrem navegar per les carpetes d'aquesta fins que seleccionem una imatge, i un cop fet això es mostrarà a la layout sota del text i els botons.

En canvi, si triem la opció de fer una foto, s'obrirà la càmera del dispositiu i podrem fer una fotografia, que podrem desar-la o descartar-la. Si la desem, es guardarà automàticament dins de la carpeta per a les fotografies, i com a nom, seguint les convencions establertes, la data i hora actuals. La carregarem i visualitzarem automàticament

Abans de mostrar la imatge per pantalla però, es realitza una petita comprovació, si la imatge és més grans que la mida de la pantalla farem un re-escalat a una mida inferior, mantenint la relació d'aspecte d'aquesta tal com podem veure a la figura 19.

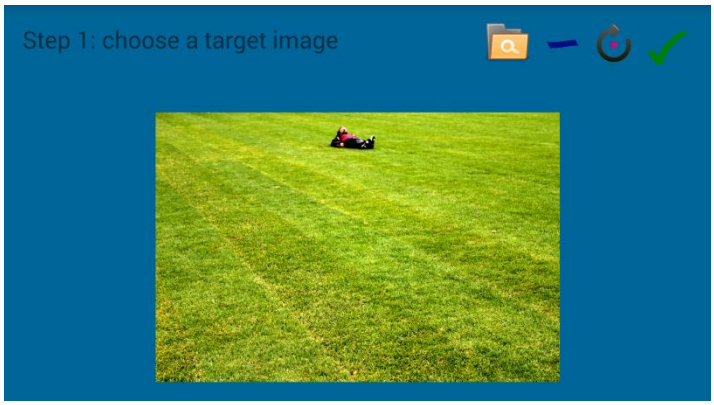


Figura 19: Imatge carregada

Ara ja podem modificar la imatge segons les nostres necessitats: podem reduir-la a un 95% de la seva mida actual, operació que pot ser repetida tants cops com vulguem.

També tenim la opció de rotar-la 90 graus en sentit horari, aquí tornarem a fer la comprovació de les mides de la imatge i la pantalla del dispositiu, i aplicaríem un re-escalat en cas de que fos necessari. Aquesta rotació la fem aplicant una matriu de transformació.

Un cop estiguem satisfets amb la imatge destí pitjarem el botó d'ok, i hi haurà un parell de canvis a la pantalla: el text canviarà i deixarem de veure la imatge destí que hem carregat, tal com podem veure a la figura 20.


Figura 20: el text informatiu ha estat modificat

Ara carregarem la imatge d'origen, utilitzant les mateixes opcions que hem utilitzat amb la imatge destí, un cop carregada també la visualitzarem.

Al prémer el botó d'ok desarem les dues imatges carregades a la memòria interna del dispositiu, ja que el fet de passar-les a les següents activitats mitjançant els *extras* de l'intent, podria causar un error de l'aplicació si la mida de les imatges és massa gran. Un cop desades a la memòria interna, s'iniciarà l'activity on dibuixarem la màscara.

### *Activity dibuixar màscara*

El fitxer que conté l'activity és DrawingActivity.java i la vista no està continguda en un XML com en els altres casos, sinó que hem creat una vista pròpia, està en el fitxer VistaDibuixar.java aquest fet l'explicarem a continuació. A la figura 21 veiem el que apareixerà en pantalla a l'iniciar l'activity.

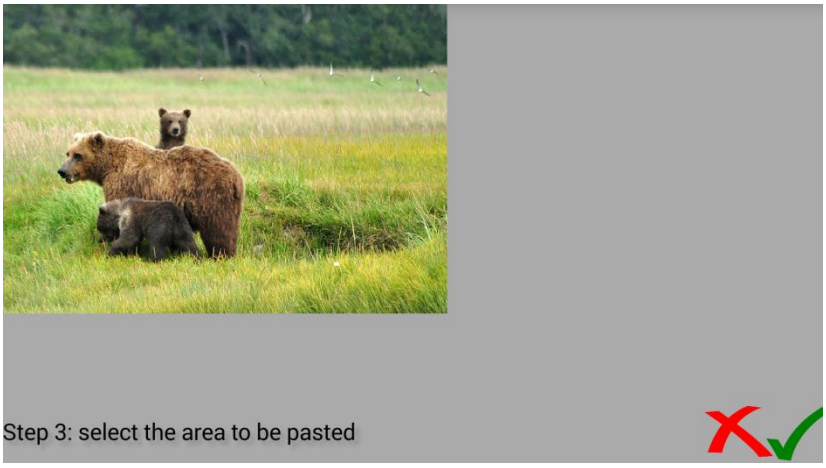


Figura 21: Activity per dibuixar la màscara

Aquí haurem de seleccionar una zona de la imatge origen que més endavant enganxarem a la imatge destí per realitzar el blending.

La idea utilitzada per seleccionar la zona és pintant sobre el Bitmap. Per aconseguir això tenim com a vista, una classe pròpia que hereta de la classe View. Això ens permet sobreescriure tot un seguit de mètodes que ens permetran dibuixar sobre el Bitmap.

Primer de tot carregarem la imatge origen que teníem desada a la memòria interna de l'aplicació, i la desem en un Bitmap. Per tal de poder dibuixar ens cal definir una pintura amb tot un conjunt de característiques tals com el color, el tipus de traç, l'amplada. En el nostre cas, l'amplada del traç serà variable depenent de la mida del Bitmap, concretament una desena part de l'arrel quadrada de l'amplada * alçada.

Necessitarem també un Canvas on poder-hi dibuixar, en crearem un amb el nostre Bitmap i el dibuixarem. Pel fet d'heretar de la classe View, tot el que veiem és un

Canvas, però no ens interessa desar cap dibuix fora de la zona del Bitmap, per això creem un Canvas específicament pel Bitmap.

Sobreescrivint el mètode onDraw de la classe View, que es va cridant periòdicament, aconseguim que es dibuixi a la pantalla tot el que volem, ja sigui, el Bitmap, els traços que anirem fent o bé el text i els botons que veurem més endavant.

Sobreescrivint el mètode OnTouchEvent, som capaços de captar els events sobre la pantalla, en el nostre cas, mirem si premem o deixem de prémer la pantalla i si realitzem un moviment mentre estem prement. I actuarem en conseqüència en els casos corresponents, ja sigui iniciant un traç, continuant-lo o acabant-lo. A la següent figura 22, podem veure com hem dibuixat sobre la imatge.


Figura 22: Màscara dibuixada sobre la imatge

A la part inferior de la pantalla tenim un text i dos imatges que actuaran com a botons, que seran dibuixats dins del mètode onDraw.

El botó de l'esquerra servirà per descartar els canvis realitzats sobre el Bitmap fins al moment, i poder-ho tornar a intentar, en canvi, el botó de la dreta ens permetrà que desem la imatge modificada a memòria i prosseguim a la següent activity.

El fet que dibuixem sobre la màscara i no marquem diversos punts i llavors obtinguem la zona continguda entre aquests, ens permet seleccionar zones separades de la imatge i no estem tancats a actuar sobre un únic bloc enganxat.

### *Activity posicionar*

Ara estarem a l'activity continguda al fitxer FinalSettingsActivity.java, i la layout corresponent està continguda al fitxer act_final_settings.xml

Com es veu a la figura 23, observem l'habitual text informatiu, la imatge destí, una check box, i un botó.

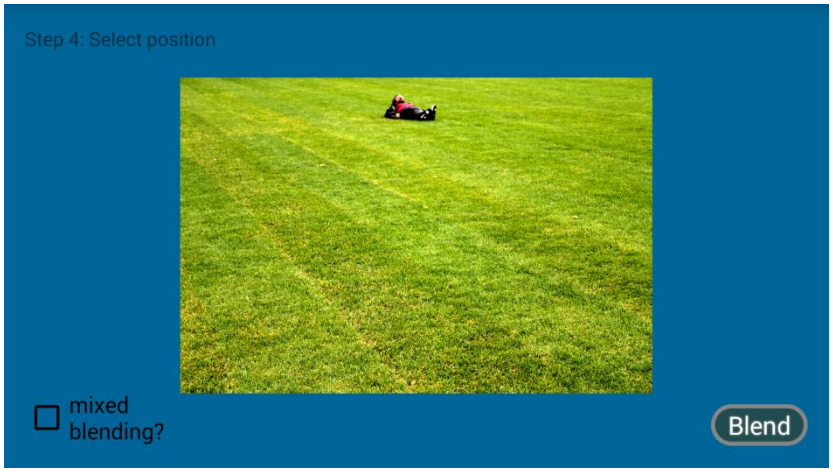


Figura 23: Inici de l'activity on posicionem la màscara

Al iniciar l'activity, construirem la màscara que utilitzarà l'algorisme posteriorment. Primer de tot carregarem la imatge origen, i la màscara modificada, i així podrem separar la zona que ens interessa. La nova màscara no tindrà la mida de la imatge origen, sinó que la farem de la mida de la imatge destí per així estalviar recursos quan l'algorisme hagi d'iterar sobre la màscara, ja que és la part més lenta del programa.

Al ImageView li afegirem un listener per capturar l'event en cas de que el toquem. Haurem de realitzar una conversió de les coordenades de l'event, per a transformar les coordenades de pantalla a coordenades de Bitmap i així poder-hi interactuar.

Amb aquestes coordenades podrem posicionar la part de la imatge origen sobre la imatge destí com podem veure a la figura 24, aquest posicionament només té un interès visual de cara l'usuari, el que ens interessa són les coordenades que posteriorment utilitzarem. Es posicionarà la imatge segons l'àrea rectangular que resulta d'haver creat la submàscara, sent el punt clicat el centre d'aquest.

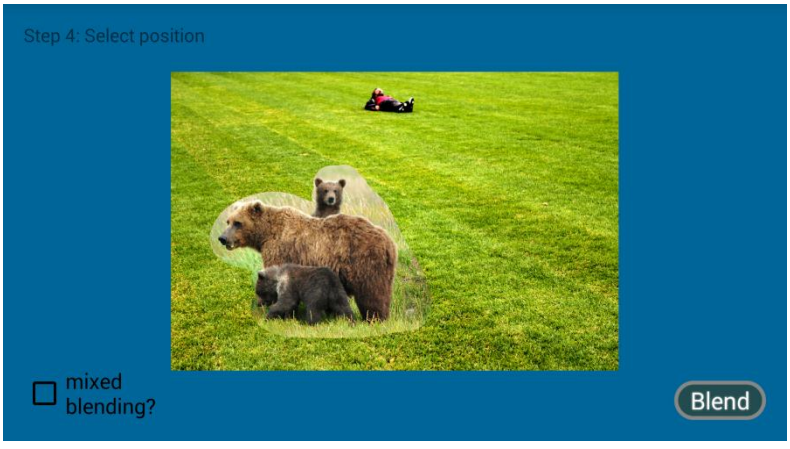


Figura 24: Imatge posicionada

A la part inferior esquerra de la pantalla trobarem una checkbox, que ens permetrà decidir quin tipus de blending: si la checkbox no està activada, només tindrem en

compte el gradient de la imatge origen, en canvi, si l'activem, tindrem en compte tant el gradient de la imatge origen com el de la imatge destí.

Un cop fet això ja podrem iniciar el procés clicant al botó, però ho farem d'una manera per que s'executi en un thread a part i no bloquejem la interfície. Farem ús d'una classe privada dins de l'activity, que hereta de AsyncTask. Mentre es realitza el blending veurem un Dialog com el de la figura 25 que ens indicarà que s'està duent a terme el blending.

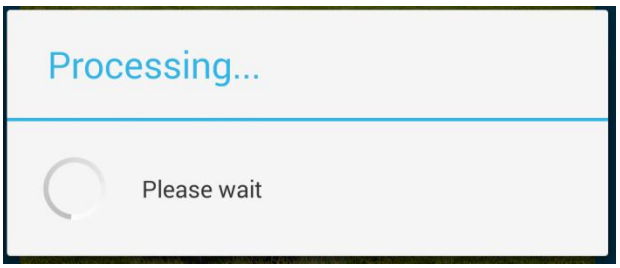


Figura 25: Diàleg d'espera

Haurem de sobreescriure uns determinats mètode d'aquesta classe que és on farem la crida al mètode del Blending. Aquests mètodes són onPreExecute, doInBackground i onPostExecute, que seran cridats en aquest orde un cop cridem al mètode execute de la classe pare.

I un cop s'hagi fet el blending, passarem a la següent activity.

Cal mencionar, que en aquesta activity tenim un Slider invisible, que ha estat utilitzat durant el testeig per veure la convergència de l'algorisme variant el paràmetre W amb aquest Slider. Per poder fer ús d'aquest Slider per seleccionar el paràmetre desitjat, només cal canviar un parell de línies de codi.

### *Activity Resultats*

Aquesta Activity està continguda al fitxer ResultsActivity.java i la layout corresponent està al fitxer act_results.xml

Aquí com veiem a la figura 26 visualitzarem la imatge resultant i tindrem dos botons possibles: guardar la imatge i veure estadístiques.

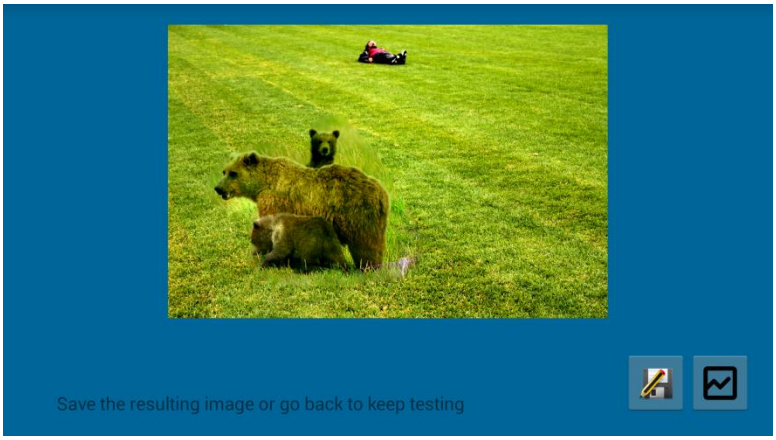


Figura 26: Resultat del blending

La opció de guardar ens obrirà un Dialog (Figura 27) amb un camp de text on introduirem el nom del fitxer destí, i la imatge es guardarà en format png dins de la carpeta Pictures.

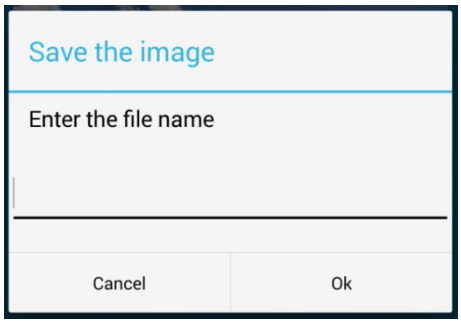


Figura 27: diàleg de guardar on podem introduïr el nom del fitxer

Prement el botó d'estadístiques podrem visualitzar el temps d'execució(Figura 28).

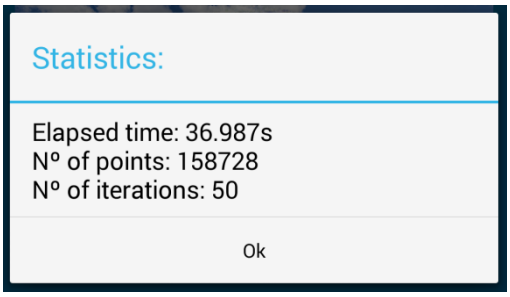


Figura 28: Estadístiques de l'execució

## 3.6. Implementació del Poisson Image editing

Un cop ja tenim la imatge destí, la imatge origen, la submàscara creada amb el desplaçament d'aquesta respecte el destí i el paràmetre W podem començar a aplicar l'algorisme. Dividirem l'explicació en tres parts: la preparació prèvia, el mètode iteratiu, i la sortida dels resultats

### *Preparació prèvia*

Haurem d'iterar sobre la submàscara que hem creat a l'activity de posicionar la màscara, i buscarem tots aquells punts que siguin de color blanc, és a dir, pertanyen a la zona a enganxar.

Aquí tindrem una llista anomenada punts, on anirem afegint un objecte propi Punt, que contindrà les coordenades x,y respecte la imatge destí, els valors vpq per cada canal de color, i el valor de cada color que anirà variant a mida que avanci l'algorisme

Mentre estem afegint aquests punts a la llista(només amb les coordenades x i y inicialitzades), a una matriu de la mida de la màscara, anirem afegint a les coordenades corresponents, l'índex d'aquests punts, això ens serà útil més endavant, perquè així, donat un element de la llista de punts, podrem saber l'índex a la llista dels tots píxels adjacents.

També calcularem el sumatori dels vpq (Figura 29) per cada píxel de la llista, ja que aquest element es mantindrà constant. Això ja ens donarà l'element de la part dreta de la equació, és a dir:

$$\sum_{q \in N_p} v_{pq}.$$

Figura 29: Sumatori dels $v_{pq}$

Aquests valors els obtindrem iterant sobre la imatge origen per cada canal de color.

Recordem, que com hem explicat abans, vpq és la resta de p menys q, on p és el punt central i q són els 4 veïns.

Això s'acaba traduint en 4*(o (x,y))-o(x-1,y)-o(x+1,y)-o(x,y-1)-o(x,y+1)

Observem aquí dues coses importants:

1. En cas de que aquest píxel s'estengués fins la vora de la imatge, aquest número 4 seria substituït pel nombre de píxels pertanyents a la imatge i el mateix succeiria amb aquests termes de la equació.
2. En el cas que el punt p pertanyi a la vora de la màscara, seguirem agafant els valors q de la imatge origen, tot i que aquelles coordenades no siguin modificades en la imatge final.

Com hem dit abans, també fem ús de un *mixed seamless cloning*, la resta de procés es manté igual, però la manera de calcular és lleugerament diferent.

Obtindrem, per cada canal de color, i cada parell de punts <p,q>, com fèiem abans el valor en la imatge origen, però ara també obtindrem aquest valor sobre la imatge destí, i en farem una comparació, utilitzarem del valor que sigui superior, parlant en termes de valor absolut, tal com hem vist a la figura 8.

### *Mètode iteratiu*

Inicialment tindrem definit un nombre inicial d'iteracions, que definirà fins quantes iteracions del mètode Gauss-Seidel amb *successive over-relaxation* realitzarem.

Com que ja sabem la quantitat de punts sobre els que treballarem, primer de tot procedim a calcular la nostra W òptima, per assegurar-nos una convergència el més ràpid possible.

Recordem que cada entrada la llista de punts, conté els valors que anirem re-calculant, i que el seu índex a la llista també ha estat inserit en una matriu.

A cadascuna de les iteracions recorrerem tota la llista de punts, i per cadascun dels valors de color per separat farem:

Recórrer els seus 4 veïns connexes i comprovar la situació de tal punt segons la màscara sumant i restant 1 a les coordenades del punt. Aquí hi ha dues possibles situacions, que tinguem un píxel intern a la màscara o bé un píxel de la vora.

Si el píxel és intern, el valor serà un els de els que estem calculant, i per tant està emmagatzemat a la llista de punts, però per conèixer la seva posició a la llista utilitzarem mirant la seva coordenada a la matriu d'índexs. Un cop obtingut aquest valor, l'anirem sumant al total de valors interns de la màscara per aquest píxel.

D'altra banda, si el píxel no pertany a l'interior de la màscara, es tractarà d'un píxel de la vora, i el seu valor corresponent el podem agafar de la imatge destí directament. I com en l'altre cas anirem sumant el total d'aquests valors.

Ara ja tenim tots els valors per calcular el nou valor d'un punt:

nou_valor = (1 - W)*valor_antic + W * (valors_interns + valors_vores + total_vpq) / Np

Aquest nou valor el desarem en el punt actual, i seguirem recorrent la llista de punts.

### *Resultats*

Un cop haguem finalitzat el mètode iteratiu, recorrerem la llista de punts, comprovarem que els valors finals no estiguin fora del rang 0-255(tot i que sempre que tinguem una entrada dins d'aquest rang, és impossible que passi), i acabarem col·locant els nous valors obtinguts a la posició corresponent sobre la imatge destí.

I la imatge ja estarà llesta per ser visualitzada.

# 4. Resultats obtinguts

Dividirem l'avaluació dels resultats obtinguts en dues parts, per una banda, analitzarem els resultats visuals obtinguts tant per el seamless blending com per el mixed seamless blending. I en l'altra part analitzarem els resultats tant en termes de convergència de l'algorisme com en temps d'execució.

## *Resultats visuals:*

Cas 1:

L'objectiu és inserir el globus de la figura 30 a la figura 31, fent ús de la màscara vista a la figura 32, la regió seleccionada consta de 79.747 punts. A la figura 33 veurem quin és el resultat de simplement enganxar el globus a la zona destí, i tot ser colors similar, el contrast és evident. En aquest cas, podem observar que els colors de fons d'una imatge a l'altre són molt similars, però l'enganxem sobre una zona ennuvolada, on-hi trobem un petit nivell de gradient que haurem de tenir en compte.

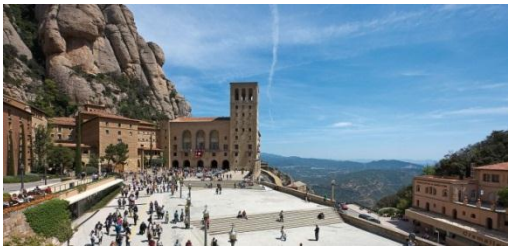

Figura 30: imatge destí

Figura 31: imatge origen

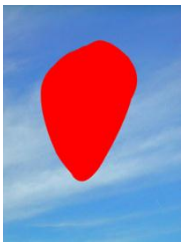

Figura32: màscara

Figura 33: Resultat de simplement enganxar la imatge origen sobre la destí

A la figura 34 veiem quin resultat hem obtingut després de 20 iteracions, i els resultats són molt bons, però fixem-nos que a la part superior dreta del globus, es veu com el núvol de la

imatge destí queda tallat. Podem provar de mirar que succeeix amb el *mixed seamless blending.*

Figura 34: Seamless blending

A la figura 35, podem observar quin és el resultat del mixed seamless blending, i el problema que comentàvem respecte a la figura 34 sembla estar solucionat i podem veure el núvol de la imatge destí. També cal dir que afortunadament el globus no rep cap transferència de gradient de la imatge destí, ja que els canvis d'il·luminació que té fan que hi hagi un gradient elevat al llarg de tota la seva superfície.

Figura 35: Mixed seamless blending

Cas 2

Volem insertar l'ull de la figura 37 a la mà de la figura 36, i farem ús de l'àrea seleccionada a la figura 38. La regió consta de 93.232 punts

Figura 36:destí

Figura 37:origen

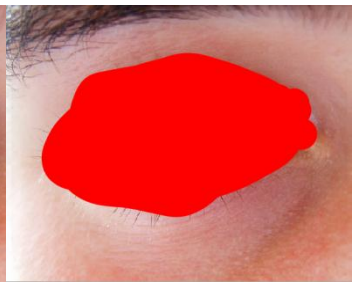

Figura 38: màscara

I tal com podem veure a la figura 39, enganxem la regió sobre la imatge destí.

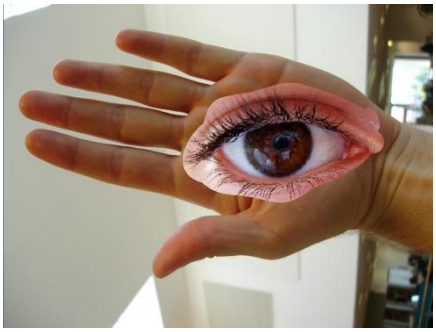

Figura 39: origen sobre destí

A la figura 40 podem veure el resultat del *seamless blending* després de 20 iteracions, un resultat prou bo en termes de color, però per exemple certes arrugues de la mà queden suprimides. El temps d'execució és de 10'06 segons.

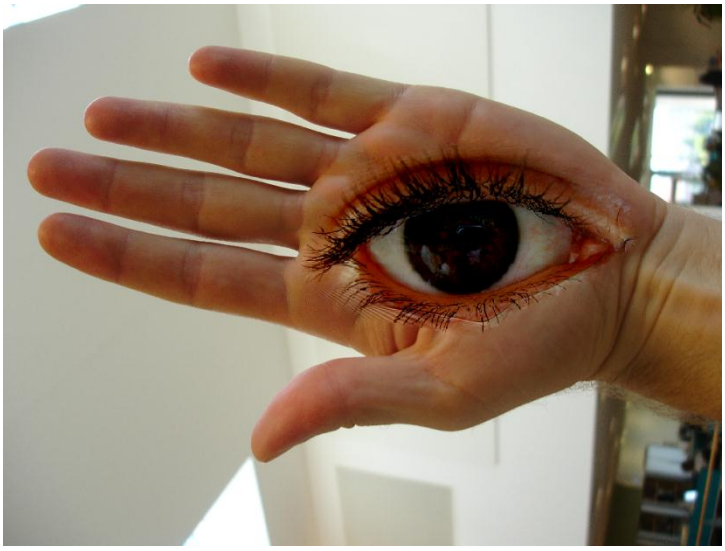

Figura 40: *seamless blending*

Si provem d'utilitzar el mixed seamless blending, per solucionar els problemses vistos a la imatge anterior, observem que aquells problemes han estat solucionats (Figura 41), però que n'ha aparegut un de nou: una de les arruges de la mà es superposa a la part blanca de l'ull, cosa que visualment no sembla acabar de quedar correcte del tot.

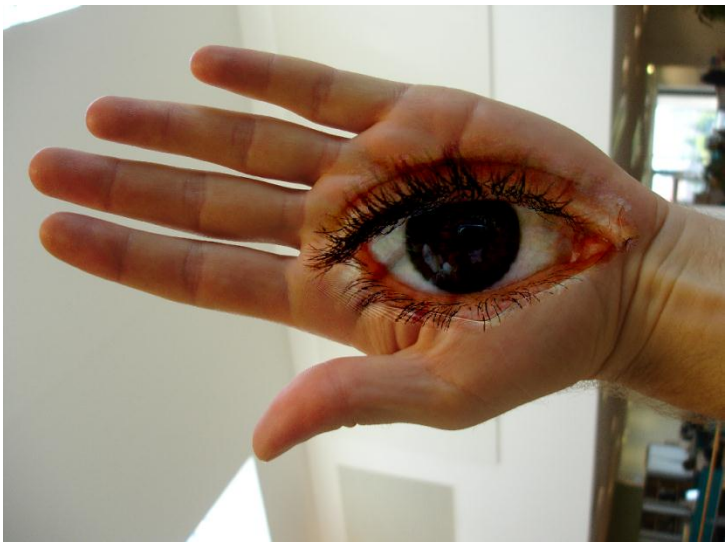

Figura 41:mixed seamless blending

### *Convergència de l'algorisme*

Aquí analitzarem la convergència sobre casos pràctics, i veurem fins a quin punt val la pena seguir iterant sobre una mateixa imatge.

Sent aquestes la imatge destí(Figura 42), origen(Figura 43) i la màscara(Figura 44) respectivament:

Figura 42: destí

Figura 43: origen

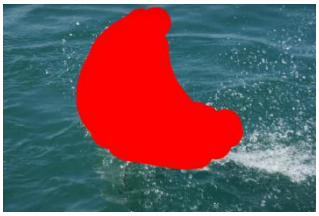

Figura 44: màscara

I aplicant una reducció, i posicionant la imatge origen sobre la imatge destí obtenim el que podem veure a la figura 45:

Figura 45: orígen sobre destí

Òbviament, les diferències són perceptibles fàcilment. Ara haurem de iterar sobre la zona seleccionada per arribar a una solució visualment millor. El Conjunt de píxels sobre el que iterem és de 80.535 i ho farem durant 100 iteracions. I realitzarem un seamless cloning normal.

Aquesta quantitat de píxels, es tradueix en un paràmetre W òptim de 1.999922

Vegem com evolucionen els valors mitjans per cada canal de color R,G i B a les figures respespectives: 46,47 i 48:

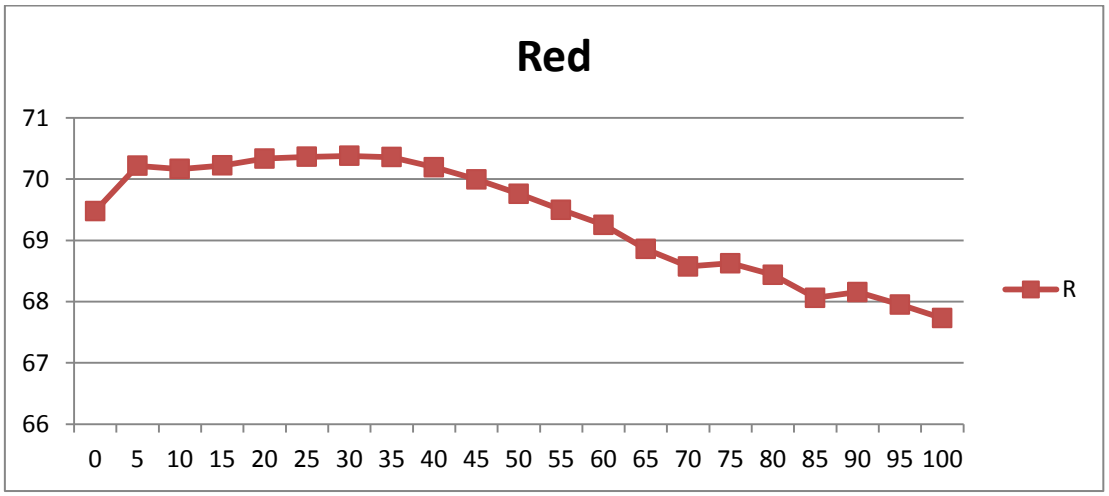


Figura 46: Gràfic sobre l'evolució de la mitjana de vermell al llarg de les iteracions

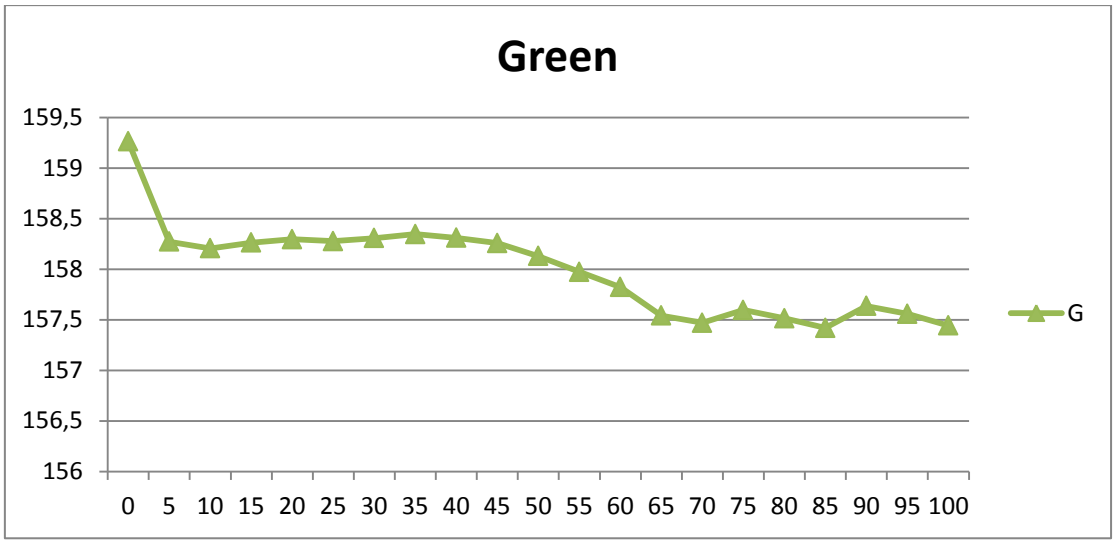


Figura 47: Gràfic sobre l'evolució de la mitjana de verd al llarg de les iteracions

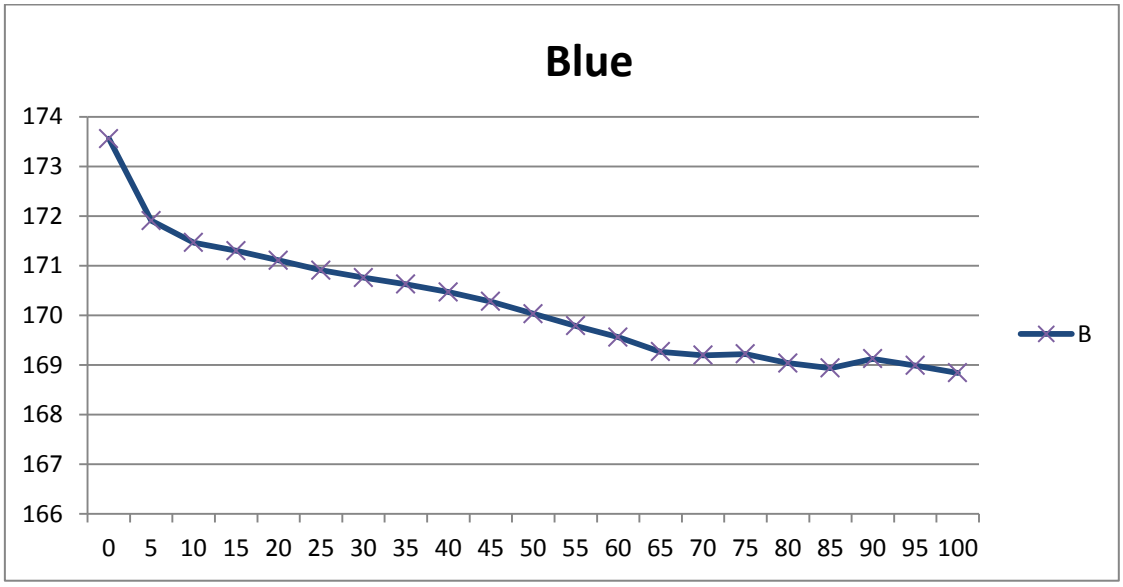


Figura 48: Gràfic sobre l'evolució de la mitjana de blau al llarg de les iteracions

Com dèiem, estem parlant dels valors mitjans de un conjunt de 80.535 píxels, per tant la variació entre cada iteració no és molt elevada. Vegem però que en les primeres 5-10 iteracions es produeix un canvi substancial, i que a partir de llavors ja ens anem acostant més lentament cap a la solució final.

També podem observar que en tots casos, en les últimes iteracions, el valor tendeix a pivotar sobre un petit rang de valors.

En aquest cas veiem que per part del color vermell inicialment percebem una lleugera pujada, i que no és fins al cap d'un seguit d'iteracions que acabem convergint cap a una intensitat inferior. Aquí cal que recordem que aquest valor és merament una mitjana, i que la propagació d'intensitat de les vores de la imatge cap a les zones internes pot trigar una mica a tenir l'efecte desitjat.

Tot i que en les gràfiques veiem que la mitjana triga més a convergir sobre una solució final, generalment, els resultats visibles no varien després de les 25-50 iteracions. Ja que com hem vist abans, en les 5-10 primeres iteracions és on realment es produeix un canvi substancial més elevat entre les mitjanes. Fem ús de les mitjanes sobre tot el conjunt, ja que prendre valors de punts aïllats no seria representatiu de la transformació que s'està duent a terme sobre el conjunt de punts.

El resultat final visible obtingut seria el que veiem a la figura 49:

Figura 49: resultat del seamless blending

### *Temps d'execució en funció del nombre de píxels*

Des del punt de vista de temps d'execució hem testejat amb diverses quantitats de píxels i exactament 100 iteracions per veure com progressava(Figura 50).

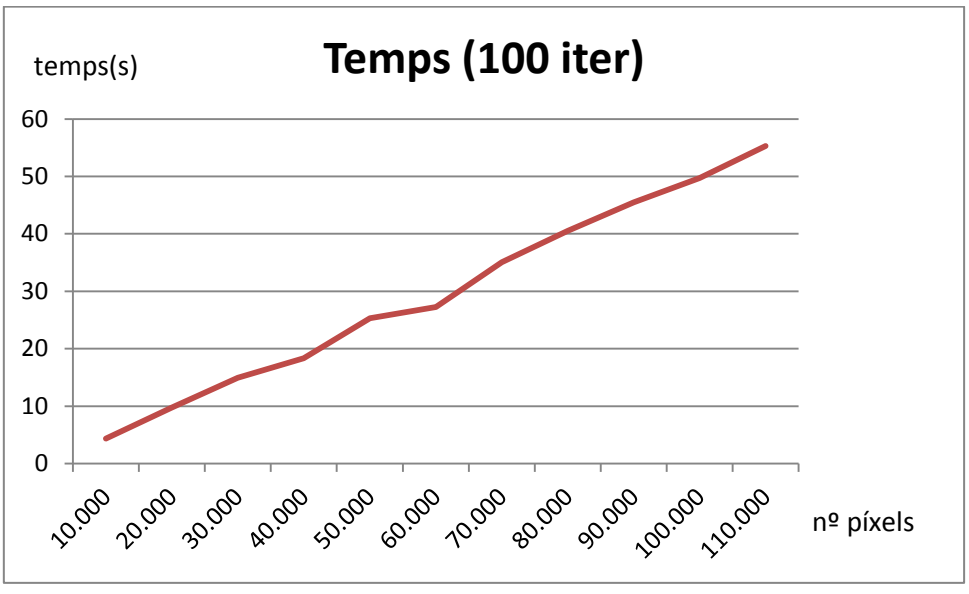


Figura 50: evolució del temps d'execució en funció del nombre de píxels tractats

Com podem veure a la figura, el temps evoluciona de forma lineal, i per tant donada una certa quantitat de píxels podem ser capaços de predir el temps que trigarà en executar-se. Hi ha petites variacions, que podem considerar negligibles, ja que la progressió global es manté.

Aquest comportament és el mateix encara que variem el nombre de iteracions.

### *Temps d'execució en funció del nombre d'iteracions*

També era interessant analitzar si donat un nombre fix de píxels, variant el nombre d'iteracions que realitzem com es comportava el temps. Com podem veure a la següent figura 51, es comporta de la forma esperada i progressa de manera lineal. Altres quantitats de píxels presenten un comportament similar.

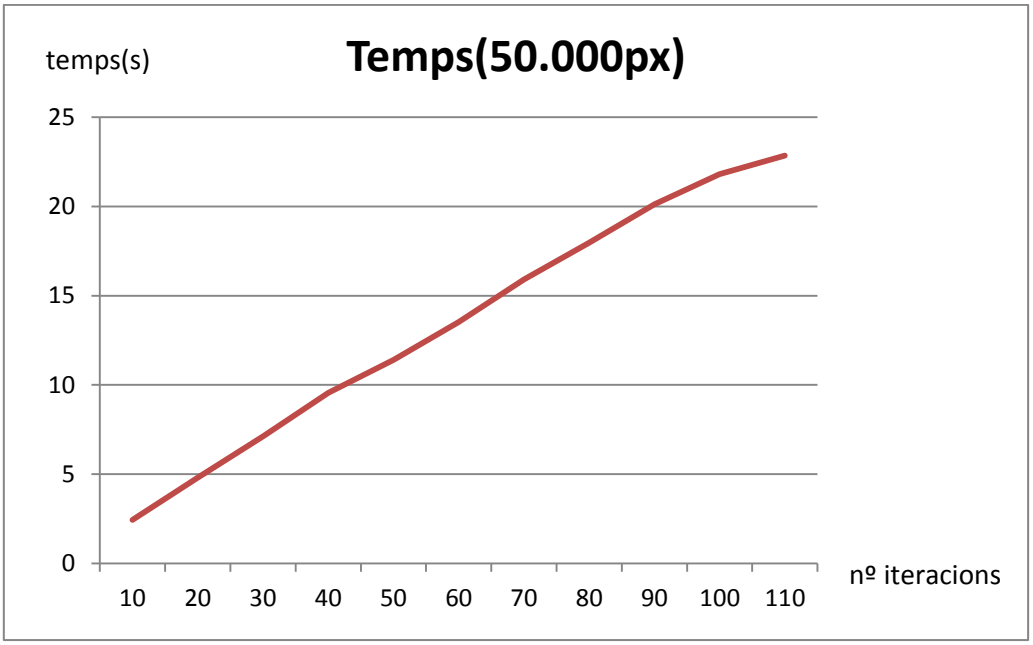


Figura 51: temps d'execució en funció del nombre d'iteracions

# 5. Conclusions

Considero que finalment, s'han assolit tots els objectius proposats des d'un bon principi, és a dir pel que fa del punt de vista de l'aplicació som capaços de carregar les imatges de manera prou còmode, amb l'afegit de poder fer-ho amb fotografies. També considero la manera de crear la màscara adient, ja que ens permet seleccionar zones no connexes, cosa que si ho haguéssim fet creant un conjunt de punts i després omplint la zona tancada no hauria estat possible. A part que degut a la mida reduïda de la mida dels dispositius, la precisió és més limitada.

Pel que fa a l'algorisme, considero els resultats visualment bons, si que és cert que sempre es transmet part de la intensitat de color cap al dibuix inserit. Però no és cap error, és tal com funciona l'algorisme aquest, ho podem veure a la pàgina 4 de l'article original. Algorismes posteriors tals com el *mean-value seamless cloning* semblen enfrontar millor aquest problema.

Un dels objectius inicials era trobar una tolerància, amb que poguéssim considerar que l'algorisme ha convergit i poder forçar-ne l'aturada, això no ha estat possible, ja que degut a que es tracta d'un problema de minimització. Ha estat testejat utilitzant píxels aïllats, i el cas era que en alguns casos, d'una iteració a la següent la variació era gairebé nul·la, però més endavant aquest valor augmentava considerablement. Un comportament similar succeeix quan treballàvem utilitzant les mitjanes. Així que tots els intents de trobar una condició d'aturada no han estat fructífers, i per això l'algorisme usa un nombre fixat d'iteracions.

També cal mencionar que tot i haver realitzat les proves d'execució amb un nombre elevat d'iteracions, el resultat visual és perceptiblement el mateix a partir de les 20 iteracions. On es redueix enormement el contrast entre les dues imatges és a les 10 primeres iteracions.

L'avantatge d'aquest algorisme és que ens permet una selecció de la zona poc exacte, ja que treballa molt bé amb les parts exteriors dels objectes.

Des del punt de vista de publicar l'aplicació a la Play Store, el mercat d'aplicacions Android de Google, seria possible, ja que he investigat una mica, i tot i que l'algorisme va ser creat per membres de Microsoft reasearch UK, no sembla haver-hi cap patent vigent sobre d'aquest. Hi ha patents sobre algorismes similars, però no sobre d'aquest. Tot i que potser consideraria realitzar diverses millores per tal d'aconseguir una experiència d'usuari òptima.

## Possibles millores

Potser si que haurà estat interessant implementar una opció per canviar la mida del pinzell a l'hora de fer el dibuix de la màscara, i que el color que pintem tingues part de transparència, per poder veure

Un dels grans inconvenients de l'algorisme és la velocitat, ja que la considero una mica lenta com per a traduir-se en una experiència per d'usuari òptima. Aquest inconvenient podria millorar-se paralelitzant, fent un thread per cada color, ja que els càlculs són independents per cada un. Més paralelització seria impossible per tal com funciona el mètode Gauss-Seidel.

Una altra opció hauria estat enviar les imatges a un servidor i realitzar els càlculs allà, i simplement retornar la imatge resultant.

## Opinió personal

Em considero molt satisfet amb els resultats que he obtingut amb aquest projecte, ha sigut un conjunt de mesos amb molta feina, primer de tot per el canvi de projecte, de fer el plug-in de GIMP a realitzar l'aplicació Android finalment.

I que seccions del projecte que semblaven prou ràpides d'implementar es van allargar massa, com per exemple la part de poder dibuixar la màscara, va ser extremadament feixuga.

Des del punt de vista d'Android he après moltíssim, ho havia tractat feia tres anys en aquesta universitat, però va ser un projecte treballat de forma modular entre quatre estudiants i els coneixements adquirits no van ser molt elevats. Així que

Tot i el volum de feina que ha comportat tot el projecte me n'emporto un bon record i un conjunt d'habilitats adquirides formidables que crec que corroboren el que après al llarg d'aquests anys i que espero seguir utilitzant en el futur.

# 6. Fonts documentals

Degut a que clarament el projecte el podríem dividir en dos camps d'interès diferents, he considerat oportú realitzar el mateix amb les fonts documentals. La major part de les fonts consultades té com a origen la web.

### *Fonts documentals sobre Android*

Referència sobre el desenvolupament en Android:
https://developer.android.com/develop/index.html

Guia d'aprenentatge Android

https://developer.android.com/training/index.html

Web per resoldre preguntes pertanyents al desenvolupament de manera col·laborativa

http://stackoverflow.com/

### *Processament d'Imatge*

Article de l'algorisme implementat:

http://www.cs.jhu.edu/~misha/Fall07/Papers/Perez03.pdf

Gauss-Seidel i *Successive over-relaxation*

http://en.wikipedia.org/wiki/Gauss%E2%80%93Seidel_method

http://en.wikipedia.org/wiki/Successive_over-relaxation

Article sobre solucionadors possibles per les equacions de Poisson:

http://www.benhumberston.com/wp-content/uploads/2012/08/CPSC_517_BenHumberston_ProjectWriteup_compressed.pdf

Apunts de l'assignatura Processament d'Imatges de la Universitat de Barcelona, d'Oriol Pujol Vila

Gradient Image processing:

http://groups.csail.mit.edu/graphics/classes/CompPhoto06/html/lecturenotes/10_Gradient.pdf

Llibre sobre  processament d'imatge:

http://szeliski.org/Book/drafts/SzeliskiBook_20100903_draft.pdf

Curs sobre processament d'imatge

http://cs.brown.edu/courses/cs129/

Mean-value cloning:

http://www.cs.huji.ac.il/~danix/mvclone/

# 7. Agraïments

Voldria agrair l'èxit d'aquest projecte a totes les persones que ho han fet possible.

Primer de tot voldria agrair-ho al meu tutor de projecte Lluís Garrido Ostermann, per tots els coneixements que m'ha sabut transmetre, imprescindibles per a l'implementació de l'algorisme. També per saber-me escoltar i aconsellar en aquells moments de bloqueig i de desesperació.

També voldria agrair-ho a tots els professors que m'han acompanyat al llarg dels estudis i dels que tant he après.

També vull agrair-ho a la meva família, que al llarg dels anys m'han donat suport a l'hora de cursar aquests estudis, i que sense ells això no hauria estat possible.

I finalment, també agrair-ho als meus amics, per comprendre aquells moments de: "Avui no puc, haig de fer projecte".

Gràcies.