Programació d'efectes en imatges en Processing

Francesc Martí Pérez

PID_00239710

Temps mínim previst de lectura i comprensió: **5 hores**

UOC
Universitat Oberta de Catalunya

Índex

Introducció

En aquest mòdul tornarem a treballar amb algunes de les transformacions estudiades en els mòduls anteriors, però fent servir el llenguatge de programació i entorn integrat de desenvolupament Processing. L'anàlisi d'aquestes operacions des d'aquest nou punt de vista proporcionarà a l'alumne una excel·lent oportunitat per reforçar i aprofundir els coneixements fins ara adquirits.

En aquest material, pressuposarem que l'alumne ja està familiaritzat amb el llenguatge de programació Processing i ens centrarem directament en l'estudi dels conceptes de l'assignatura en aquest entorn. Per a una exhaustiva introducció a Processing, l'alumne té a la seva disposició el llibre *Processing*, editat per l'UOC, a l'àrea de materials de l'assignatura.

Començarem aquest mòdul introduint com carregar, modificar, visualitzar i guardar imatges en Processing. En particular, veurem en detall com accedir als píxels d'una imatge i interactuar-hi. Per acabar aquest apartat introductori, també examinarem breument alguns dels filtres que ja vénen implementats a Processing i que ens permetran realitzar algunes de les transformacions que cal estudiar en unes poques línies de codi.

Un cop introduïts aquests conceptes bàsics de Processing, ja estarem en disposició d'abordar la programació de les transformacions vistes en els quatre primers mòduls de l'assignatura en aquest nou entorn. És a dir, les transformacions estudiades en els mòduls «Histogrames i transformacions puntuals», «Transformacions espacials lineals», «Transformacions espacials no lineals» i «Transformacions geomètriques».

Al llarg dels quatre apartats següents, anirem descrivint els algoritmes que ens permetran implementar moltes de les transformacions fins ara vistes. Estudiarem en detall aquests programes, compararem els resultats obtinguts amb els resultats obtinguts prèviament amb Photoshop i arribarem a la conclusió que el resultat final no depèn del programari que es fa servir, sinó de l'algoritme utilitzat. Aquest fet sorprèn alguns alumnes, que pressuposen que el resultat d'un programari de pagament ha de ser a la força millor que el resultat d'un programari gratuït. Photoshop, en aplicar totes les transformacions estudiades, no fa servir fórmules secretes o procediments ocults. Fa servir les operacions matemàtiques que hem analitzat durant el curs. Implementant a Processing el mateix algoritme que a Photoshop, amb les mateixes sumes, multiplicacions o tècniques de comparació del nivell de gris dels píxels, obtenim el mateix resultat.

Totes les imatges que s'utilitzen o anomenen en aquest mòdul s'adjunten amb els materials del curs. No cal dir que s'aconsella encaridament als alumnes reproduir i estudiar en el seu equip tots els exemples aquí presentats. Introduir canvis en el codi o estudiar els resultats sobre altres imatges acostumen a ser bones tàctiques per a arribar a una perfecta comprensió dels conceptes que s'hi desenvolupen.

Objectius

Els principals objectius d'aquest mòdul són:

1. Introduir la programació de les transformacions puntuals, espacials lineals, espacials no lineals i geomètriques.

2. Explorar les possibilitats del llenguatge Processing pel que fa a aquestes programacions.

3. Relacionar, mitjançant experiments dirigits, els conceptes introduïts amb la transformació d'imatges.

Aquests objectius estan relacionats amb les següents competències de l'assignatura:

A. Capacitat de modificar una imatge digital segons uns requisits previs.

B. Capacitat de canviar la resolució, relació d'aspecte i forma d'una imatge.

C. Capacitat de discriminar les opcions factibles de les que no ho són en un estudi d'especificacions d'un projecte, sistema o tasca.

G. Capacitat d'inserir contingut visual en una aplicació a Processing.

I amb les següents competències generals del grau:

11. Capturar, emmagatzemar i modificar informació d'àudio, imatge i vídeo digitals, tot aplicant principis i mètodes de realització i composició del llenguatge audiovisual.

12. Capacitat per integrar i gestionar continguts digitals en aplicacions multimodals d'acord amb criteris estètics, tècnics i funcionals.

13. Capacitat per utilitzar de forma apropiada els llenguatges de programació i les eines de desenvolupament per a l'anàlisi, el disseny i la implementació d'aplicacions.

23. Capacitat d'analitzar un problema en el nivell d'abstracció adequat a cada situació i aplicar les habilitats i els coneixements adquirits per a abordar-lo i resoldre'l.

1. Imatges en Processing

1.1. Càrrega i visualització d'imatges en Processing

Començarem aquest mòdul amb un senzill exemple, que ens servirà d'excusa per presentar la classe PImage de Processing. Aquesta classe pertany al nucli de Processing –per tant, no és necessari instal·lar cap biblioteca addicional per utilitzar-la– i és la que ens permet treballar amb imatges.

La classe PImage inclou diversos *camps* i *mètodes* per carregar, crear, tractar o guardar imatges. En aquestes pàgines n'estudiarem només alguns, els relacionats amb les transformacions d'imatges que ens concerneixen.

Enllaç d'interès

Remetem l'alumne a la documentació Javadoc de Processing per una visió completa la classe PImage i les seves possibilitats: http://processing.github.io/processing-javadocs/core/.

Així doncs, comencem amb un primer exemple que ens mostrarà com carregar i visualitzar una imatge a la finestra de l'aplicació de Processing. En aquest i els propers exemples, l'alumne ha d'advertir com tots els programes giren entorn dels objectes PImage, i fer atenció als camps i mètodes d'aquesta classe necessaris per implementar les transformacions examinades en mòduls precedents.

```
/**
* Exemple 1: Càrrega i visualització d'una imatge
* Francesc Martí, martifrancesc@uoc.edu, 15-04-2016
*/

// Declarem un objecte de tipus PImage
PImage img;

void setup() {
// Assignem a la finestra de treball les mateixes mides que la imatge
size(696,696);

// Carreguem la imatge
img = loadImage("geranidolor.jpg");

// La funció draw() s'executarà només una vegada
noLoop();
}

void draw() {
   //La funció image() permet visualitzar la imatge a la finestra de l'aplicació
image(img,0,0);
}
```

Com hem comentat, aquest codi permet carregar a Processing una imatge emmagatzemada en el disc dur de l'ordinador i visualitzar-la per pantalla i, en ser el primer exemple, estudiarem en detall el seu codi.

1) Declaració de l'objecte. Amb el següent codi es declara l'objecte `img` de la classe PImage.

```
PImage img;
```

Com ja hem comentat, la declaració d'un objecte de tipus PImage és imprescindible per treballar amb imatges a Processing, i la declaració és igual a com declararíem una variable de tipus `float` o `int`, per exemple.

2) Inicialització de l'objecte. Com ja sabem, sempre que es declara un objecte cal inicialitzar-lo, construir-lo. En aquest cas, aquesta feina es realitza cridant la funció `loadImage()`. Aquest mètode permet carregar una imatge en un objecte PImage i té com a paràmetre el nom de la imatge a carregar (més endavant ja veurem com es pot crear també una nova imatge »buida»).

```
img = loadImage("geranidolor.jpg");
```

Cal comentar que, per defecte, si no indiquem la ruta de la imatge, Processing buscarà aquesta imatge en una carpeta de nom «data», dintre de la carpeta de l'*sketch*. I, en el cas que la ruta o el nom de la imatge sigui incorrecte, ens apareixerà un missatge d'error advertint el problema.

Processing accepta els formats d'imatge GIF, JPG, TGA i PNG. També és capaç de llegir les imatges en format TIFF, però –i això és important– només si s'han generat prèviament a Processing.

Per acabar l'anàlisi d'aquest primer exemple, només ens falta parlar de la funció `image()`. Aquesta funció ens permet visualitzar la imatge a la finestra de l'aplicació i, en aquest exemple, amb les coordenades (0,0) indiquem a Processing que situï la cantonada superior esquerra de la imatge a les coordenades (0,0) de la finestra de l'aplicació.

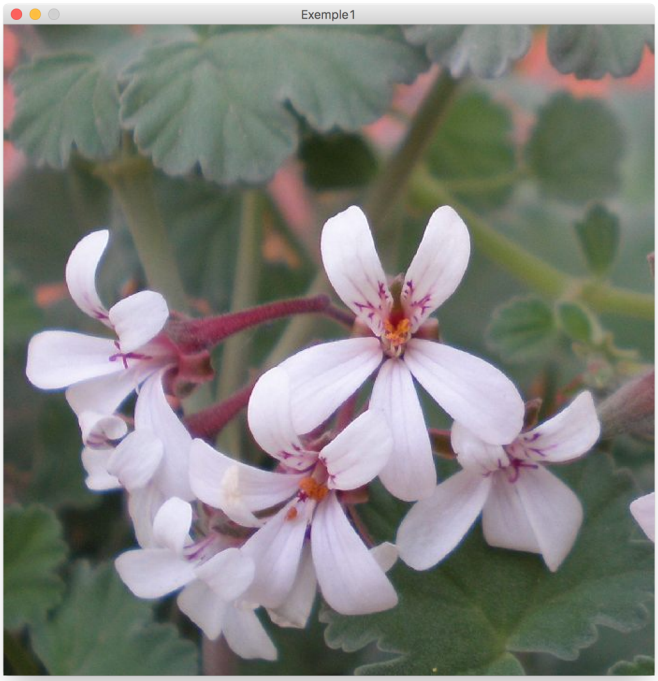
```
image(img,0,0);
```

Altres paràmetres de la funció image()

La funció `image()` també accepta dos paràmetres més, que permeten canviar la resolució de la impressió de la imatge a la finestra de l'aplicació. Per exemple, `image(img,0,0,200,200)` mostraria la imatge amb una resolució de 200 × 200 píxels. Fem notar, però, que les dimensions de tota la finestra de l'aplicació continuarien sent 696 × 296 píxels.

Així doncs, si executem l'*sketch* anterior, el resultat és el que es mostra a la figura 1.

Figura 1. Finestra de l'aplicació de l'exemple 1

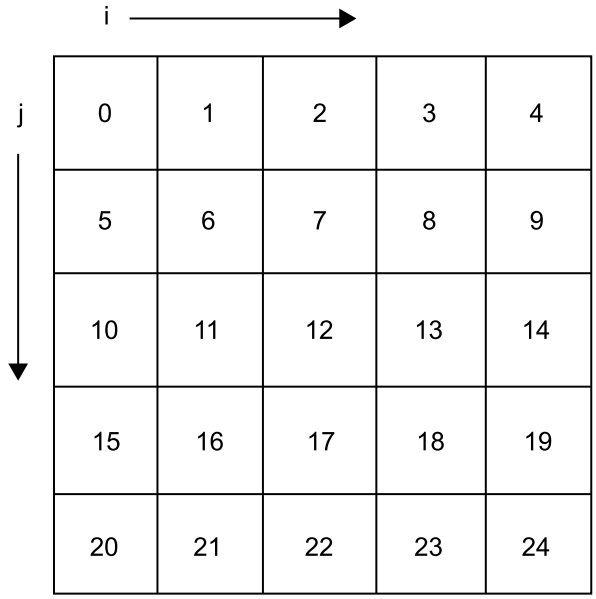


1.2. Imatges i píxels

Com ja vam veure al mòdul "Histogrames i transformacions puntuals", podem interpretar una imatge digital com una quadrícula, on la intersecció de cada fila i cada columna és un píxel de la imatge. Això funciona exactament igual a Processing. Cada vegada que definim un objecte de tipus PImage, automàticament creem una quadrícula de píxels.

Tècnicament parlant, els objectes PImage guarden els píxels d'aquesta quadrícula en un *array de píxels*, anomenat `pixels[]`, amb la relació que mostrem en la figura 2.

Figura 2. Relació entre la quadrícula de píxels i l'*array* pixels[].



Per exemple, el píxel de la quadrícula situat a la columna 0 i la fila 2 es guarda en la posició 10 de l'*array* de píxels, i el píxel situat a la columna 1 i fila 3, en la 16.

En general, en una imatge de dimensions $m \times n$, el píxel situat a la columna i i la fila j, es guarda en l'*array* de píxels en la posició

$$i + j \cdot m$$

Tornant a la figura 2, com que aquesta quadrícula correspon a una imatge de dimensions 5 × 5, per exemple, el píxel situat a la columna 4 i fila 3 es guarda en la posició $4 + 3 \cdot 5 = 19$ de l'*array* de píxels.

Posem ara en pràctica aquest concepte desenvolupant una aplicació que ens permeti accedir als píxels d'una imatge i consultar el seu valor RGB. Seleccionarem els píxels fent clic amb el ratolí sobre la imatge.

```
/**
* Exemple 2: Imatges i Píxels (I)
* Francesc Martí, martifrancesc@uoc.edu, 15-04-2016
*
*/

// Declarem un objecte de tipus PImage
PImage img;

void setup() {
// Assignem a la finestra de treball les mateixes mides que la imatge
size(696, 696);
```

```
// Carreguem la imatge
img = loadImage("geranidolor.jpg");

// Sempre hem de cridar aquesta funció abans d'accedir a l'array de píxels
img.loadPixels();
}

void draw() {
//La funció image() permet visualitzar la imatge
image(img, 0, 0);
}

void mousePressed() {
// La variable loc serveix per "localitzar" el píxel seleccionat dins
// l'array de pixels
int loc = mouseX + mouseY * img.width;

// Extraiem el color del píxel
color c = img.pixels[loc];

// Aquestes funcions permeten consultar les components R, G i B d'un color
float r = red(c);
float g = green(c);
float b = blue(c);

// Finalment, imprimim el resultat a la finestra de la Console
println("El valor RGB del píxel (" + mouseX + ", " + mouseY + ") és (" + r + ", " + g + ", " + b + ")");
}
```

Analitzem ara els nous conceptes introduïts en aquest codi, començant per la crida a la funció

```
img.loadPixels();
```

Com ja hem comentat, en crear un objecte de tipus PImage, automàticament creem una quadrícula de píxels. Per poder treballar amb aquests píxels, sempre hem de cridar abans el mètode `loadPixels()`. És una forma de dir a Processing: «Prepara'm els píxels de la imatge, que necessito treballar-hi».

Com es pot veure, la major part del codi d'aquest programa es troba dins la funció `mousePressed()`, ja que volem consultar els nivells RGB d'un píxel cada vegada que fem clic amb el ratolí sobre la imatge.

El primer que fem és localitzar a l'*array* de punts el punt sobre el qual hem clicat `(mouseX, mouseY)` amb la fórmula que hem vist anteriorment.

```
int loc = mouseX + mouseY * img.width;
```

Com és fàcilment deduïble, el camp `img.width` ens proporciona l'amplada de la imatge i, si volguéssim consultar l'alçada, hauríem d'escriure `img.height`. Sí que és cert que en aquest cas ja sabíem quina és l'amplada i l'alçada de la imatge, però sempre és aconsellable consultar les dimensions d'una imatge: d'aquesta manera evitem errors i podem fer servir altres imatges amb altres mides, sense haver d'estar editant aquesta part del codi contínuament.

Un cop que sabem «on és» el punt que ens interessa, consultem el seu color i extraiem els seus nivells R, G i B.

```
// Extraiem el color del píxel
color c = img.pixels[loc];

// Aquestes funcions permeten consultar les components R, G i B d'un color
float r = red(c);
float g = green(c);
float b = blue(c);
```

Per raons pedagògiques, en aquest exemple fem servir les funcions `red()`, `green()` i `blue()` ja que és molt senzill entendre què fan. De totes maneres, el següent codi seria equivalent i més ràpid d'execució:

```
// Aquestes funcions permeten consultar les components R, G i B d'un color
float r = img.pixels[loc] >> 16 & 0xFF;
float g = img.pixels[loc] >> 8 & 0xFF;
float b = img.pixels[loc] & 0xFF;
```

Sense entrar en detalls, Processing guarda cada component R, G i B en 1 byte (8 bits) i en una posició determinada. Amb els operadors >> (*right shift*) i & (*and*) és possible anar directament al byte que guarda el valor de cada component i consultar el seu valor, operació més ràpida que fer servir les funcions `red()`, `green()` i `blue()`.

Finalment, un cop extrets els valors que cercàvem, només els hem d'imprimir a la finestra de la consola amb la funció `println()`.

Un altre mètode per extreure el color d'un píxel

També és possible extreure el color d'un píxel (*x*,*y*) fent servir la funció `get(x,y)`. Encara que és un mètode més senzill d'implementar, és sensiblement més lent que el d'accedir a l'*array* `pixels[]`, per la qual cosa, en general, es desaconsella la seva utilització.

Per finalitzar aquest subapartat, veurem com –a més de poder llegir la informació dels píxels d'una imatge– també els podem modificar i crear una nova imatge amb ells. Aprofitarem aquest exemple per veure també com podem guardar la nova imatge creada al nostre disc dur.

Així doncs, en el següent exemple filtrarem la imatge «geranidolor.jpg», respectant a cada píxel el valor del seu component vermell (R), però assignant el valor 0 als components verd (G) i blau (B).

```
/**
* Exemple 3: Imatges i Píxels (II)
* Francesc Martí, martifrancesc@uoc.edu, 15-04-2016
*
*/

// Declarem dos objectes de tipus PImage, un per la imatge original
// i un altre per la imatge filtrada
PImage imgOriginal;
PImage imgFilter;

void setup() {
// Carreguem la imatge
imgOriginal = loadImage("geranidolor.jpg");

// Creem una nova imatge amb les mateixes dimensions que la imatge original
imgFilter = createImage(imgOriginal.width, imgOriginal.height, RGB);

// Assignem a la finestra de treball les mateixes mides que la imatge
surface.setSize(imgOriginal.width, imgOriginal.height);

// La funció draw() s'executarà només una vegada
noLoop();
}

void draw() {
// Com sempre, hem de cridar aquestes funcions abans d'accedir a l'array
// de píxels de les imatges
imgOriginal.loadPixels();
imgFilter.loadPixels();

int loc= 0;

// Recorrem tots els píxels de la imatge
while (loc < imgOriginal.pixels.length) {
// Aquestes funcions permeten consultar les components R, G i B d'un color
float r = imgOriginal.pixels[loc] >> 16 & 0xFF;
float g = imgOriginal.pixels[loc] >> 8 & 0xFF;
float b = imgOriginal.pixels[loc] & 0xFF;

// A la imatge filtrada només importem el valor del nivell R (vermell)
// i els altres dos els deixem amb nivell 0
imgFilter.pixels[loc] = color(r, 0, 0);

loc++;
}
```

```
// Si fem modificacions sobre els píxels, sempre hem d'actualitzar l'array
// de píxels amb la funció updatePixels()
imgFilter.updatePixels();

// La funció image() permet visualitzar la imatge filtrada
image(imgFilter, 0, 0);

// Guardem el resultat a la carpeta "data" del projecte
imgFilter.save(dataPath("geranidolor2.jpg"));
}
```

Com podem veure, en aquest exemple creem dos objectes de tipus PImage, un per carregar la imatge original i un altre per guardar les modificacions i el resultat final en el disc dur.

```
PImage imgOriginal;
PImage imgFilter;
```

Ja hem estudiat com inicialitzar un objecte cridant la funció `loadImage()` i passant-li el nom de la imatge que volem carregar. Ara, per crear una nova imatge «buida» hem de cridar la funció `createImage()` i passar-li les mides que volem que tingui i el format. De fet, tècnicament, la imatge no està buida. Per defecte, Processing crea una nova imatge amb tots els seus píxels en color negre (0, 0, 0).

En el nostre cas, volem crear una imatge amb les mateixes mides que la imatge original i en format RGB. Per tant, el codi ha de ser:

```
imgFilter = createImage(imgOriginal.width, imgOriginal.height, RGB);
```

Seguidament, anem omplint els píxels d'aquesta nova imatge amb el valor de vermell de la imatge original i amb 0 els altres dos components. Això ho fem amb

```
imgFilter.pixels[loc] = color(r, 0, 0);
```

Un punt molt important del codi és

```
imgFilter.updatePixels();
```

Com ja hem vist, amb la funció `loadPixels()` preparem els píxels d'una imatge per llegir-los. Però si els modifiquem, en acabar, hem de cridar la funció `updatePixels()`. En aquest programa hem modificat només els valors dels píxels de `imgFilter`, que per defecte tots valien (0, 0, 0), així que ara hem d'actualitzar els seus valors amb la crida `imgFilter.updatePixels()`.

Finalment, si, a més de mostrar el resultat a la finestra de l'aplicació, volem guardar la imatge filtrada en el disc dur, haurem de fer servir el mètode `save()`, assignant un nom i una ruta d'emmagatzematge a la nova imatge, com es pot veure en aquest codi:

```
imgFilter.save(dataPath("geranidolor2.jpg"));
```

Aquí, l'extensió indicarà el format de la imatge de sortida. Si volguéssim guardar el resultat en format TIFF hauríem d'afegir «.tif» al nom de la imatge. És a dir,

```
imgFilter.save(dataPath("geranidolor2"));
```

Anàlogament, hauríem d'afegir les terminacions «.tga», «.jpg» o «.png» per guardar la imatge en format TARGA, JPEG o PNG, respectivament. Si no especifiquem cap format, Processing guarda la imatge en format TIFF.

També convé comentar que sempre és recomanable indicar una ruta absoluta on guardar la imatge a la funció `save()`. En aquest exemple això ho fem amb la funció `dataPath("")`, que és una forma d'indicar la carpeta «data» del projecte. Si no ho fem així, és possible que en alguns casos rebem el missatge d'error «Pimage.save() requires an absolue path. Use createImage(), or pass savePath() to save()».

Sobre la impressió de la imatge a la finestra de l'aplicació, fem notar que, en aquest exemple, per definir les mides de la finestra, a l'inici hem fet servir el codi

```
surface.setSize(imgOriginal.width, imgOriginal.height);
```

Aquest codi ens permet definir les mides de la finestra de l'aplicació sense saber prèviament les mides de la imatge i, a més, no l'hem de canviar encara que carreguéssim imatges de diferents mides. Hem de fer notar que la funció que hem fet servir prèviament, `size()`, no admet variables (a partir de la versió 3 de Processing), per la qual cosa el codi

```
size(imgOriginal.width, imgOriginal.height);
```

no seria correcte, i rebríem el corresponent missatge d'error en intentar executar-ho.

Acabem, doncs, mostrant el resultat d'aquest programa. A la següent figura podem veure la imatge que genera el codi anterior: una imatge monocromàtica composta per diverses intensitats de color vermell.

Figura 3. Finestra de l'aplicació de l'exemple 3

1.3. El mètode filter() de PImage

Com hem comentat abans, la classe PImage inclou diversos *camps* i *mètodes*, dels quals només n'estudiarem alguns. En el grup dels que analitzarem es troba la funció `filter()`, una funció que permet filtrar una imatge amb diferents mètodes.

La seva sintaxi és molt senzilla. Si `img` és un objecte de tipus PImage, podem filtrar aquesta imatge amb el codi

```
img.filter(filterName, parametre);
```

Els mètodes del filtre disponibles inclouen THRESHOLD, GRAY,OPAQUE, INVERT, POSTERIZE, BLUR, ERODE i DILATE, i alguns d'aquests necessiten el pas d'un paràmetre. Per exemple, el filtre GRAY no necessita paràmetres

```
img.filter(GRAY);
```

en canvi, el filtre THRESHOLD, sí.

```
img.filter(THRESHOLD, 0.3);
```

Molt resumidament, els diferents mètodes disponibles a la funció `filter()` són els següents:

- **THRESHOLD**: és la transformació de binarització, on el valor del llindar va de 0.0 (0) a 1.0 (255). Aquest mètode necessita paràmetre.

- **GRAY**: converteix una imatge en color a escala de grisos. No necessita paràmetre.
- **OPAQUE**: fa completament opaca una imatge. No necessita paràmetre.
- **INVERT**: és la transformació en negatiu. No necessita paràmetre.
- **POSTERIZE**: limita el nombre de colors de cada canal. El valor del paràmetre estableix aquest límit, i pot variar entre 2 i 255.
- **BLUR**: aplica un filtre de tipus *Guassian blur*, amb el radi especificat en el segon paràmetre. Si no s'inclou el segon paràmetre, el valor del radi del filtre és 1. Com més gran sigui el radi, més acusat serà el filtrat.
- **ERODE**: aplica la transformació espacial no lineal erosió. No necessita paràmetre.
- **DILATE**: aplica la transformació espacial no lineal dilatació. No necessita paràmetre.

Per entendre millor el funcionament d'aquests filtres, veiem un exemple senzill amb el filtre GRAY.

```
/**
* Exemple 4: Filtre GRAY
* Francesc Martí, martifrancesc@uoc.edu, 15-04-2016
*
*/

// Declarem un objecte de tipus PImage
PImage img;

void setup() {
// Carreguem la imatge
img = loadImage("geranidolor.jpg");

// Assignem a la finestra de treball les mateixes mides que la imatge
surface.setSize(img.width, img.height);

// La funció draw() s'executarà només una vegada
noLoop();
}

void draw() {
// Apliquem el filtre GRAY a la imatge
img.filter(GRAY);

//La funció image() permet visualitzar la imatge filtrada
image(img, 0, 0);

// Guardem el resultat a la carpeta "data" del projecte
img.save(dataPath("geranidolor2.jpg"));
```

```
}
```

Executant aquest codi –com esperàvem– obtenim una imatge en escala de grisos.

Figura 4. Finestra de l'aplicació de l'exemple 4

Al darrer programa d'aquest subapartat, veiem un exemple molt similar, però ara amb un filtre que sí que necessita paràmetre: un filtre de tipus POSTERIZE.

```
/**
* Exemple 5: Filtre POSTERIZE
* Francesc Martí, martifrancesc@uoc.edu, 15-04-2016
*
*/

// Declarem un objecte de tipus PImage
PImage img;

void setup() {
// Carreguem la imatge
img = loadImage("geranidolor.jpg");

// Assignem a la finestra de treball les mateixes mides que la imatge
surface.setSize(img.width, img.height);

// La funció draw() s'executarà només una vegada
noLoop();
}
```

```
void draw() {
// Apliquem el filtre POSTERIZE a la imatge
img.filter(POSTERIZE, 6);

//La funció image() permet visualitzar la imatge filtrada
image(img, 0, 0);

// Guardem el resultat a la carpeta "data" del projecte
img.save(dataPath("geranidolor2.jpg"));
}
```

A la següent figura, podem veure quin seria el resultat d'executar aquest programa, on hem configurat un filtre de tipus POSTERIZE, amb un nombre màxim de colors per canal de 6.

Figura 5. Finestra de l'aplicació de l'exemple 5

Així doncs, en aquest subapartat hem examinat el mètode `filter()` de la classe PImage, i hem vist com és de fàcil aplicar un d'aquests filtres a una imatge. Tornarem a fer servir aquests filtres a l'apartat 4, en particular els filtres ERODE i DILATE, però abans, en els propers dos apartats, de forma anàloga a com hem fet en l'exemple 3, veurem com aplicar transformacions puntuals i transformacions espacials lineals a una imatge, accedint als seus píxels i aplicant uns algoritmes.

Si bé, amb aquest mètode, la implementació d'aquestes transformacions serà una mica més complicada, el seu estudi proporcionarà a l'alumne un sòlid coneixement del funcionament de les transformacions d'imatges, de «què succeeix» cada cop que apliquem un filtre a Photoshop, o fem servir el mètode `filter()` a Processing. A més, la comprensió per part de l'alumne d'aquests mecanismes, automàticament, anirà unida a l'adquisició de poderoses eines que li permetran desenvolupar les seves pròpies transformacions.

2. Transformacions puntuals

Les transformacions puntuals es caracteritzen per tractar la imatge com a un conjunt de píxels independents. Amb l'ajuda de la funció que defineix cada transformació, cada nou píxel de la imatge transformada s'obté a partir del píxel amb les mateixes coordenades de la imatge original.

Vegeu també

Les transformacions puntuals s'estudien en el mòdul "Histogrames i transformacions puntuals" d'aquesta assignatura.

En aquest apartat, bàsicament, tornarem a veure quines són aquestes funcions, com programar-les i com aplicar-les sobre una imatge donada per generar la imatge transformada.

Començarem programant la transformació puntual més senzilla, la identitat, i l'agafarem com a model per generar la resta de transformacions, ja que, com veurem, el codi serà molt similar.

De manera anàloga a com vam fer al mòdul "Histogrames i transformacions puntuals", en aquest apartat els programes que desenvoluparem estan pensats per ser aplicats sobre imatges en escala de grisos. En aquest punt, cal comentar que Processing, quan carrega una imatge, per exemple, JPEG o PNG en escala de grisos, per defecte, la tracta com si fos una imatge RGB, assignant el mateix valor a cadascun dels canals. Encara que tècnicament ens trobem amb una imatge RGB, si els valors dels tres canals són sempre iguals, R = G = B, podem continuar pensant que es tracta d'una imatge en escala de grisos, ja que són imatges equivalents.

2.1. Identitat

Per a cadascuna de les transformacions d'aquest apartat, recordarem breument la seva definició, però no entrarem a estudiar-les en detall.

Vegeu també

Recordeu que l'estudi detallat de les transformacions que tractem en aquest apartat es va fer al mòdul "Histogrames i transformacions puntuals".

La transformació puntual identitat consisteix a assignar a cada píxel de la imatge transformada el mateix valor de nivell de gris que el píxel amb les mateixes coordenades de la imatge original. Per tant, per a un píxel donat X de nivell de gris l de la imatge original, el nivell de gris del píxel Y amb les mateixes coordenades de la imatge transformada vindrà determinat per la funció:

$$T(l) = l$$

Passem doncs a veure directament com es pot programar aquesta transformació a Processing, i estudiarem el seu codi en detall més endavant.

```
/**
```

```
* Exemple 6: Transformació Puntual Identitat,
* per imatges en escala de grisos
* Francesc Martí, martifrancesc@uoc.edu, 15-04-2016
*
*/

// Declarem dos objectes de tipus PImage, un per la imatge original
// i un altre per la imatge filtrada
PImage imgOriginal;
PImage imgFilter;

void setup() {
// Carreguem la imatge
imgOriginal = loadImage("4.1.02.png");

// Creem una nova imatge amb les mateixes dimensions que
// la imatge original
imgFilter = createImage(imgOriginal.width, imgOriginal.height, RGB);

// Les mides de la finestra de treball permetran visualitzar
// les dues imatges alhora
surface.setSize(2*imgOriginal.width, imgOriginal.height);

// La funció draw() s'executarà només una vegada
noLoop();
}

void draw() {
// Com sempre, hem de cridar aquestes funcions abans d'accedir a
// l'array de píxels de les imatges
imgOriginal.loadPixels();
imgFilter.loadPixels();

int loc= 0;

// Recorrem tots els píxels de la imatge
while (loc < imgOriginal.pixels.length) {
// En ser una imatge en escala de grisos, R = G = B
// Així que tenim suficient consultant un dels canals RGB
float b = imgOriginal.pixels[loc] & 0xFF;

// Algoritme - Transformació Puntual (Identitat)
imgFilter.pixels[loc] = color(b);

loc++;
}
```

```
// Si fem modificacions sobre els píxels, sempre hem d'actualitzar
// l'array amb la funció updatePixels()
imgFilter.updatePixels();

// La funció image() permet visualitzar les dues imatges
image(imgOriginal, 0, 0);
image(imgFilter, imgOriginal.width, 0);

// Guardem la imatge transformada a la carpeta "data" del projecte
imgFilter.save(dataPath("imgFilter.png"));
}
```

Comencem comentant que en els següents exemples sempre mostrarem alhora, a la finestra de l'aplicació, la imatge original i la imatge modificada. Per tant, definim les mides de la finestra de forma que puguin mostrar les dues imatges simultàniament.

```
surface.setSize(2*imgOriginal.width, imgOriginal.height);
```

Com ja hem comentat abans, en ser una imatge en escala de grisos, tenim que el nivell de gris d'un píxel ve determinat per qualsevol dels valors R, G o B, ja que R = G = B. En el nostre cas, utilitzem el canal B (blau) per consultar el valor del nivell de gris d'un píxel.

```
float b = imgOriginal.pixels[loc] & 0xFF;
```

Finalment, l'algoritme de la transformació puntual identitat ve definit per

```
imgFilter.pixels[loc] = color(b);
```

on, com es pot veure, a cada píxel de la nova imatge li assignem el mateix valor de gris que el píxel corresponent de la imatge original.

A la següent imatge podem veure quin seria el resultat d'aquesta transformació. Com esperàvem, les dues imatges, l'original i la transformada, són iguals.

Figura 6. Finestra de l'aplicació de l'exemple 6

2.2. Negatiu

La transformació puntual negatiu, o inversa, consisteix a assignar a cada píxel de la imatge transformada el valor de nivell de gris invertit que el píxel amb les mateixes coordenades de la imatge original. Per tant, si l és aquest valor de gris, la funció de la transformació puntual negatiu és:

$$T(l) = 255 - l$$

El codi de totes les transformacions puntuals és molt semblant i, bàsicament, només canvia l'algoritme que s'aplica. Per tant, el codi de la transformació puntual negatiu és igual que el codi de la transformació puntual identitat, excepte per la línia on es defineix el seu algoritme. Substituint la línia de l'algoritme de l'exemple identitat per

```
// Algoritme - Transformació Puntual (Negatiu)
imgFilter.pixels[loc] = color(255-b);
```

ja tindríem el codi de la transformació negatiu.

Com podem veure, aquí la variable `b` guarda el nivell de gris del cadascun dels píxels de la imatge original. I el píxel corresponent de la imatge transformada tindrà un nivell de gris `255-b`.

A la següent imatge, podem veure quin seria el resultat d'aplicar aquesta transformació. Com podem veure, és el mateix resultat que obteníem en aplicar aquesta transformació amb Photoshop.

Figura 7. Finestra de l'aplicació de l'exemple 7

2.3. Binarització

La transformació puntual binarització genera una imatge amb dos nivells de gris: blanc (255) i negre (0). Aquesta transformació queda definida per un llindar o *threshold* i la funció:

$$T(l)=\begin{cases}255 & l \geq \text{llindar} \\ 0 & l < \text{llindar}\end{cases} \qquad (1)$$

on, com abans, l és el nivell de gris d'un píxel X.

El codi a Processing d'aquesta transformació també serà pràcticament igual als anteriors exemples. Aquí, l'única diferència és que la funció de la transformació necessita la variable `llindar`, que definirem a l'inici del programa.

```
// Declarem dos objectes de tipus PImage, un per la imatge original
// i un altre per la imatge filtrada
PImage imgOriginal;
PImage imgFilter;

// Llindar
int llindarValue = 20;
```

Com hem estudiat en el mòdul "Histogrames i transformacions puntuals", el valor del llindar pot ser qualsevol número entre 0 i 255 (suposant que, com fem habitualment, treballem amb 256 nivells de gris). En aquest cas, en ser una imatge bastant fosca, escollim un valor de llindar baix, però suggerim als alumnes que experimentin amb altres valors.

Anàlogament als casos anteriors, substituïm la línia de l'algoritme. En aquest cas, la fórmula de la binarització seria

```
// Algoritme - Transformació Puntual (Binarització)
if (b < llindarValue) {
imgFilter.pixels[loc] = color(0);
} else {
imgFilter.pixels[loc] = color(255);
}
```

A la figura 8, podem veure quin seria el resultat de la transformació binarització amb llindar 20.

Figura 8. Finestra de l'aplicació de l'exemple 8

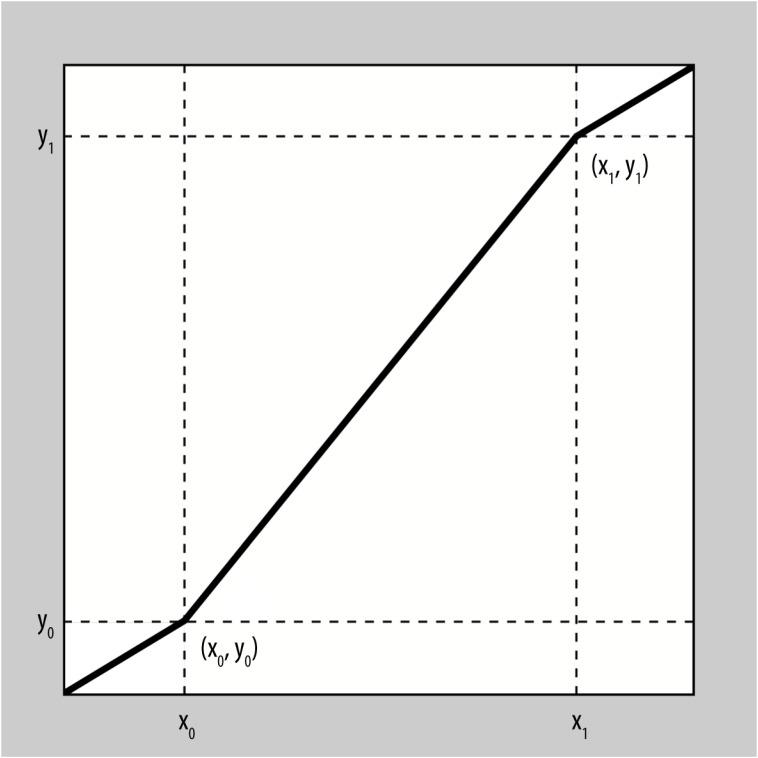
2.4. Transformacions d'aclariment i enfosquiment de la imatge

Unes de les transformacions estudiades amb més detall al mòdul "Histogrames i transformacions puntuals" han estat les transformacions d'aclariment i enfosquiment de la imatge. Com hem vist, és possible definir corbes que aclareixen una imatge i és possible definir corbes que enfosqueixen una imatge. També hem vist que és possible definir una transformació d'aclariment i enfosquiment per parts, que combina les dues transformacions, i té la finalitat de modificar el contrast d'una imatge.

En aquest subapartat estudiarem directament les transformacions d'aclariment i enfosquiment per parts, perquè ja inclouen les versions simples d'aquestes transformacions.

El codi d'aquesta transformació és molt semblant al codi de la transformació puntual binarització, però ara, la transformació ve donada per una funció per parts, com la de la figura 9, que vam estudiar al mòdul "Histogrames i transformacions puntuals".

Figura 9. Exemple de transformació per parts



Ha de quedar clar que la corba de la transformació podria tenir més o menys parts, amb altres proporcions, i que aquestes podrien ser lineals (com les de la figura 9) o fer servir altres tipus de funcions. En l'exemple d'aquest subapartat implementarem una transformació per parts amb una funció amb una gràfica semblant a la de la figura 9.

Com es pot veure, el primer que necessitem és definir els valors `x0`, `x1`, `y0` i `y1`, per definir la forma de la corba. Anàlogament a la transformació binarització, aquesta assignació de valors la fem a l'inici del codi.

```
// Amb aquests valors, definim la forma de la corba d'aclariment i enfosquiment
intx0 = 23;
intx1 = 250;
inty0 = 65;
inty1 = 240;
```

No hi ha cap fórmula matemàtica que indiqui quins són els valors que ha de tenir la corba de la transformació per parts. En ser la imatge original una imatge bastant fosca, s'han agafat uns valors que aclareixen força la imatge, però s'invita a l'alumne a experimentar amb altres valors.

Un cop definida la corba, el segon pas és definir l'algoritme de la transformació.

```
// Algoritme - Transformació Puntual (Aclariment i Enfosquiment)
if (b < x0 ) {
imgFilter.pixels[loc] = color(int(map(b, 0, x0, 0, y0)));
```

```
} else if ( b < x1 ) {
imgFilter.pixels[loc] = color(int(map(b, x0, x1, y0, y1)));
} else {
imgFilter.pixels[loc] = color(int(map(b, x1, 255, y1, 255)));
}
```

El que fem en aquest algoritme, bàsicament, és primer determinar en quina «part» es troba el nivell de gris que cal avaluar i, després, aplicar la funció que li correspon. Com es pot veure, en aquest algoritme fem servir la funció `map()`, que realitza una correspondència lineal entre dos segments de valors.

Anem a veure, doncs, tot el codi d'aquesta transformació puntual d'aclariment i enfosquiment per parts.

```
/**
* Exemple 9: Transformació Puntual Aclariment i Enfosquiment per parts,
* per imatges en escala de grisos
* Francesc Martí, martifrancesc@uoc.edu, 15-04-2016
*
*/

// Declarem dos objectes de tipus PImage, un per la imatge original
// i un altre per la imatge filtrada
PImage imgOriginal;
PImage imgFilter;

// Amb aquests valors, definim la forma de la corba d'aclariment i enfosquiment
intx0 = 23;
intx1 = 250;
inty0 = 65;
inty1 = 240;

void setup() {
// Carreguem la imatge
imgOriginal = loadImage("4.1.02.png");

// Creem una nova imatge amb les mateixes dimensions que la imatge original
imgFilter = createImage(imgOriginal.width, imgOriginal.height, RGB);

// Les mides de la finestra de treball permetran visualitzar
// les dues imatges alhora
surface.setSize(2*imgOriginal.width, imgOriginal.height);

// La funció draw() s'executarà només una vegada
noLoop();
}
```

```
void draw() {
// Com sempre, hem de cridar aquestes funcions abans d'accedir a
// l'array de píxels de les imatges
imgOriginal.loadPixels();
imgFilter.loadPixels();

int loc= 0;

// Recorrem tots els píxels de la imatge
while (loc < imgOriginal.pixels.length) {
// En ser una imatge en escala de grisos, R = G = B
// Així que tenim suficient consultant un dels canals RGB
float b = imgOriginal.pixels[loc] & 0xFF;

// Algoritme - Transformació Puntual (Aclariment i Enfosquiment)
if ( b < x0 ) {
imgFilter.pixels[loc] = color(int(map(b, 0, x0, 0, y0)));
} else if ( b < x1 ) {
imgFilter.pixels[loc] = color(int(map(b, x0, x1, y0, y1)));
} else {
imgFilter.pixels[loc] = color(int(map(b, x1, 255, y1, 255)));
}

loc++;
}

// Si fem modificacions sobre els píxels, sempre hem d'actualitzar
// l'array amb la funció updatePixels()
imgFilter.updatePixels();

// La funció image() permet visualitzar les dues imatges
image(imgOriginal, 0, 0);
image(imgFilter, imgOriginal.width, 0);

// Guardem el resultat a la carpeta "data" del projecte
imgFilter.save(dataPath("imgFilter.png"));
}
```

A la figura 10, podem veure quin seria el resultat d'aplicar una transformació puntual d'aclariment i enfosquiment per parts sobre la imatge "4.1.02.png". Com s'aprecia, la imatge s'ha aclarit sensiblement. De totes maneres, com ja s'ha comentat, s'invita a l'alumne a experimentar amb altres valors i comparar els resultats.

Figura 10. Finestra de l'aplicació de l'exemple 9

3. Transformacions espacials lineals

Les transformacions espacials són aquelles en què el valor de cada píxel de la imatge transformada depèn del píxel amb les mateixes coordenades de la imatge original i els seus veïns. Dins de les transformacions espacials, en aquest apartat parlarem de les transformacions espacials lineals.

Vegeu també

Les transformacions espacials s'estudien al mòdul "Transformacions espacials lineals" d'aquesta assignatura.

El primer canvi que haurem d'introduir en el codi és la forma en què recorrem els píxels de la imatge. Fins ara, hem fet servir el codi

```
// Recorrem tots els píxels de la imatge
while (loc < imgOriginal.pixels.length) {
```

En les transformacions puntuals, la imatge és tractada com a un conjunt de píxels independents, així que només ens hem d'assegurar que recorrem tots els píxels de la imatge, sense importar l'ordre.

En canvi, a les transformacions espacials necessitem operar sobre un píxel i els seus veïns. Si tornem a consultar la figura 2, podem veure que el píxel 9 i el píxel 10 es troben un a cada punta de la imatge. Si accedim a l'*array* de píxels de forma seqüencial ens trobarem que serà molt complicat calcular els píxels veïns d'un píxel donat.

Així que ens interessa recórrer la imatge de manera que, un cop determinat el píxel sobre el qual fer les operacions, determinar quins són els seus veïns sigui senzill. Això es fa recorrent la imatge fila per fila amb el següent codi:

```
// Recorrem tots els píxels de la imatge
for (int x = 0; x < imgOriginal.width; x++) {
for (int y = 0; y < imgOriginal.height; y++) {
```

Un cop determinat un píxel (x,y), la seva posició dins de l'*array* de píxels vindrà determinada –com ja hem vist– per la fórmula

```
int loc = x + y * imgOriginal.width;
```

Anàlogament al que hem fet a l'anterior apartat, començarem programant la transformació espacial lineal més senzilla, la identitat, i l'agafarem com a model per generar la resta de transformacions. Com veurem, el codi serà el mateix a tots els subapartats, i només haurem de modificar els valors de la màscara i el desplaçament (*offset*) per aplicar les diferents transformacions espacials lineals.

3.1. Identitat

Comencem amb la transformació espacial lineal identitat. Introduïm directament el codi que permet efectuar aquesta transformació i l'analitzarem en detall més endavant. Igual que en l'anterior apartat, els programes són per imatges en escala de grisos.

```
/**
* Exemple 10: Transformació espacial lineal Identitat,
*per imatges en escala de grisos
* Francesc Martí, martifrancesc@uoc.edu, 15-04-2016
*
*/

// Declarem dos objectes de tipus PImage, un per la imatge original
// i un altre per la imatge filtrada
PImage imgOriginal;
PImage imgFilter;

// Màscara de convolució (Identitat) expressada com a matriu
float[][] matrix = {
{ 0, 0, 0 },
{ 0, 1, 0 },
{ 0, 0, 0 } };

// Dimensió de la màscara de convolució
int matrixsize = 3;

// Correcció desplaçament
int offset = 0;

void setup() {
// Carreguem la imatge
imgOriginal = loadImage("4.2.06.png");

// Creem una nova imatge amb les mateixes dimensions que la imatge original
imgFilter = createImage(imgOriginal.width, imgOriginal.height, RGB);

// Les mides de la finestra de treball permetran visualitzar
// les dues alhora
surface.setSize(2*imgOriginal.width, imgOriginal.height);

// La funció draw() s'executarà només una vegada
noLoop();
}

void draw() {
```

```
// Com sempre, hem de cridar aquestes funcions abans d'accedir a
// l'array de píxels de les imatges
imgOriginal.loadPixels();
imgFilter.loadPixels();

// Recorrem tots els píxels de la imatge
for (int x = 0; x < imgOriginal.width; x++) {
for (int y = 0; y < imgOriginal.height; y++) {
// Càlcul de la convolució espacial
int c = convolution(x, y, matrix, matrixsize, offset, imgOriginal);

// Generem un nou píxel a la imatge filtrada
int loc = x + y * imgOriginal.width;
imgFilter.pixels[loc] = color(c);
}
}

// Si fem modificacions sobre els píxels, sempre hem d'actualitzar
// l'array amb la funció updatePixels()
imgFilter.updatePixels();

// La funció image() permet visualitzar les dues imatges
image(imgOriginal, 0, 0);
image(imgFilter, imgOriginal.width, 0);

// Guardem el resultat a la carpeta "data" del projecte
imgFilter.save(dataPath("imgFilter.png"));
}

// Funció que calcula la convolució espacial
int convolution(int x, int y, float[][] matrix, int matrixsize, int offset, PImage img) {
float result = 0.0;
int half = matrixsize / 2;

// Recorrem la matriu de convolució
for (int i = 0; i < matrixsize; i++) {
for (int j = 0; j < matrixsize; j++) {
// Càlcul del píxel sobre el que estem treballant
int xloc = x + i - half;
int yloc = y + j - half;
int loc = xloc + img.width * yloc;

// Ens assegurem que agafem un píxel dintre del rang vàlid. En aquest cas estem aplicant la
// replicació de valors de píxels propers per localitzacions de píxels que surten de la imatge
loc = constrain(loc, 0, img.pixels.length-1);

// Càlcul de l'operació convolució
```

```
// Consultem el valor del canal blue (B)
result += ((imgOriginal.pixels[loc] & 0xFF) * matrix[i][j]);
}
}

// Apliquem el desplaçament
result += offset;

// Ens assegurem que el nivell de gris està en el rang (0, 255)
result = constrain(result, 0, 255);

// Retornem el nivell de gris
return (int)result;
}
```

Analitzem ara aquest codi. Com podem veure, comencem definint les variables que necessitarem per poder realitzar la transformació: la matriu de convolució (`matrix`), la dimensió de la matriu (`matrixsize`) i el desplaçament (`offset`). En aquest exemple, tenim la matriu de convolució identitat, la seva dimensió és 3 (3 × 3) i, com ja sabem, si la suma dels coeficients de la matriu és 1, el desplaçament ha de ser 0.

```
// Màscara de convolució (Identitat) expressada com a matriu
float[][] matrix = {
{ 0, 0, 0 },
{ 0, 1, 0 },
{ 0, 0, 0 } };

// Dimensió de la màscara de convolució
int matrixsize = 3;

// Correcció desplaçament
int offset = 0;
```

Com ja hem vist, recorrem els píxels de la imatge fila per fila amb el codi

```
for (int x = 0; x < imgOriginal.width; x++) {
for (int y = 0; y < imgOriginal.height; y++) {
```

I calculem l'operació convolució espacial amb la funció `convolution()`.

```
int c = convolution(x, y, matrix, matrixsize, offset, imgOriginal);
```

Un cop calculat el valor de la convolució, ja podem assignar aquest valor de gris al píxel corresponent de la imatge transformada.

```
int loc = x + y * imgOriginal.width;
```

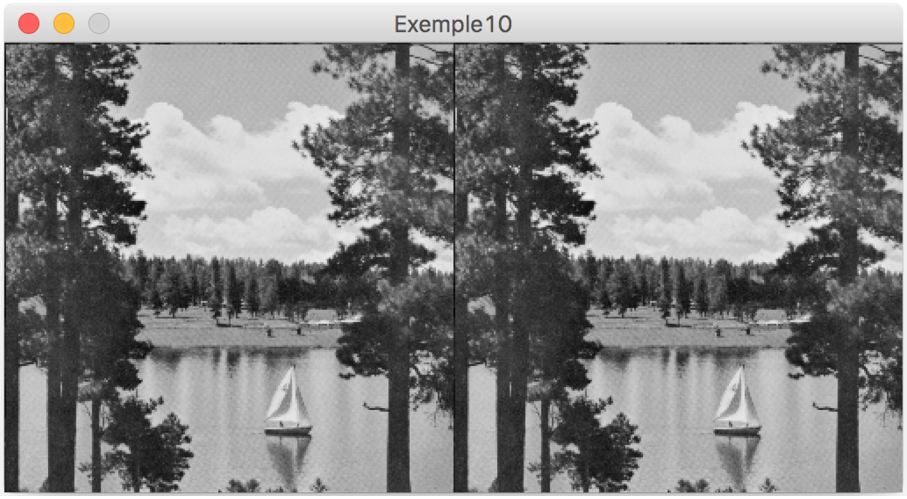
```
imgFilter.pixels[loc] = color(c);
```

En aquest codi, la funció `convolution()` conté l'algoritme encarregat de fer les operacions que hem descrit en detall en el mòdul "Transformacions espacials lineals". Li enviem les coordenades del píxel de la imatge original, els valors de la matriu de convolució, la seva dimensió, el seu desplaçament i la imatge, i ens retorna el nivell de gris que ha de tenir el píxel de la imatge transformada amb les mateixes coordenades.

```
int convolution(int x, int y, float[][] matrix, int matrixsize, int offset, PImage img)
```

A la figura 11, podem veure quin seria el resultat d'aplicar la màscara de convolució identitat a la imatge "4.2.06.png". Com no podia ser d'una altra manera, les dues imatges són idèntiques.

Figura 11. Finestra de l'aplicació de l'exemple 10



3.2. Negatiu

A partir d'ara, el codi de totes les transformacions espacials lineals és igual, excepte per la matriu de convolució i el desplaçament.

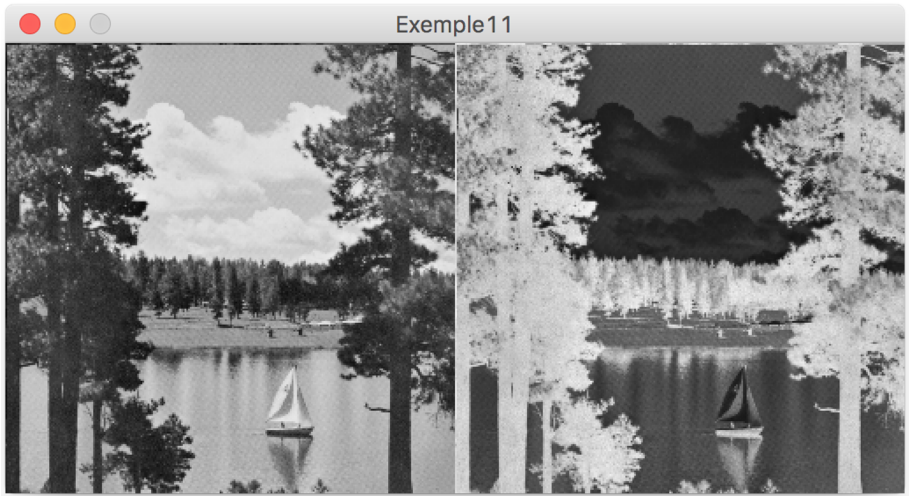
Per exemple, el codi de la transformació espacial lineal negatiu serà igual que el codi de la transformació espacial lineal identitat, però substituint la matriu de convolució i el desplaçament pels valors:

```
// Màscara de convolució Negatiu expressada com a matriu
float[][] matrix = {
{ 0, 0, 0 },
{ 0, -1, 0 },
{ 0, 0, 0 } };

// Dimensió de la màscara de convolució
int matrixsize = 3;
```

```
// Correcció desplaçament
int offset = 255;
```

Si executem el codi, a la figura 12 tenim el resultat d'aquesta transformació.

Figura 12. Finestra de l'aplicació de l'exemple 11



Com podem veure, el resultat és equivalent a efectuar transformació puntual negatiu.

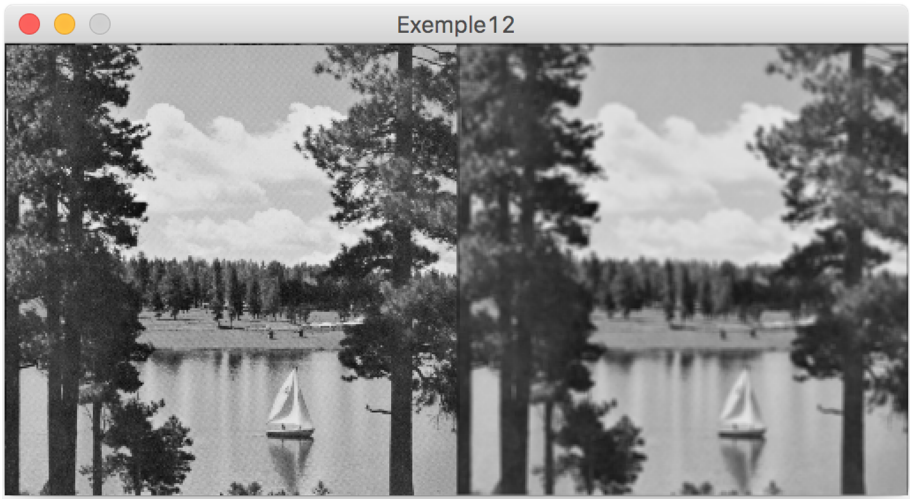
3.3. Suavització

Com a exemple de suavització, aplicarem una matriu de convolució ja estudiada en el mòdul "Transformacions espacials lineals".

```
// Màscara de convolució Suavització, expressada com a matriu
float[][] matrix = {
{ 1/9f, 1/9f, 1/9f },
{ 1/9f, 1/9f, 1/9f },
{ 1/9f, 1/9f, 1/9f } };

// Dimensió de la màscara de convolució
int matrixsize = 3;

// Correcció desplaçament
int offset = 0;
```

A la figura 13 tenim el resultat d'aquesta transformació. Com apreciem clarament, la màscara de suavització tendeix a difuminar la imatge.

Figura 13. Finestra de l'aplicació de l'exemple 12



3.4. Contorns

Acabarem aquest subapartat amb dos exemples de codi per detectar i realçar els contorns d'una imatge.

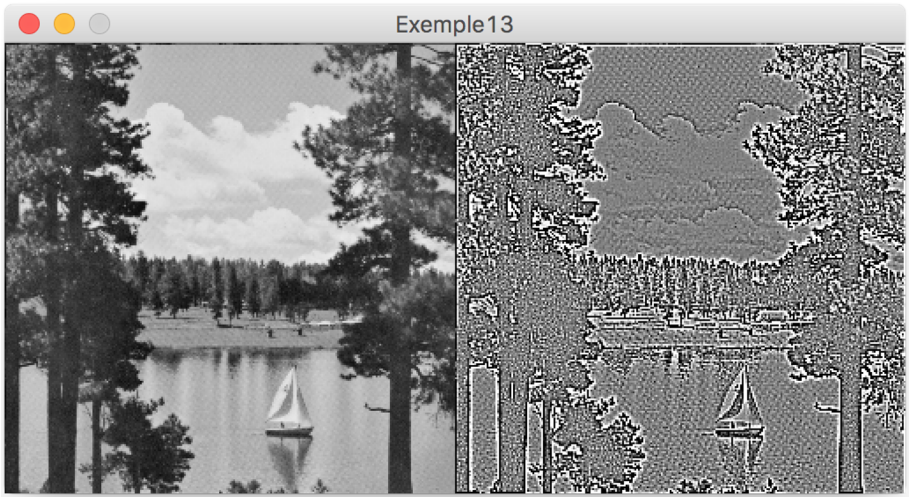
Vegeu també

Les dues màscares d'aquests programes ja han estat estudiades al mòdul "Transformacions espacials lineals", així que no ens aturarem de nou en la seva anàlisi.

```
// Màscara de convolució Contorns (detecció) expressada com a matriu
float[][] matrix = {
{ -1, -1, -1 },
{ -1, 8, -1 },
{ -1, -1, -1 } };

// Dimensió de la màscara de convolució
int matrixsize = 3;

// Correcció desplaçament
int offset = 128;
```

A la figura 14 tenim el resultat d'aquesta transformació.

Figura 14. Finestra de l'aplicació de l'exemple 13

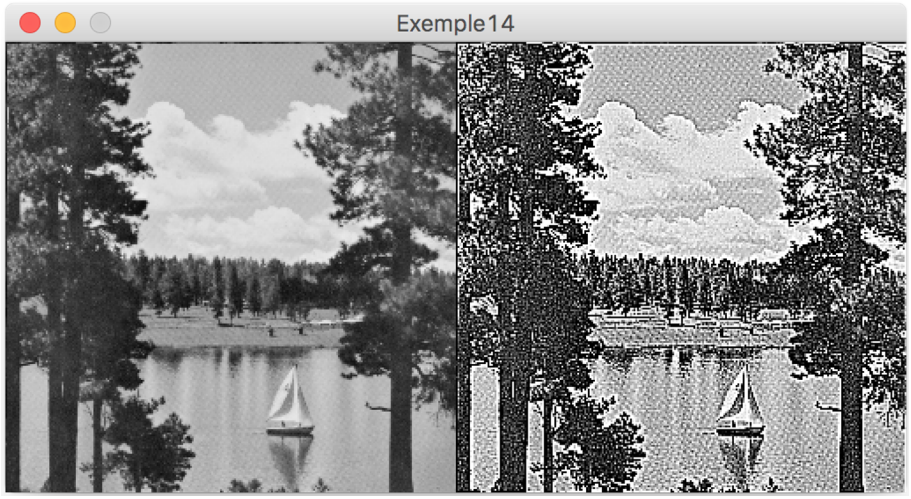


Finalment, un codi que ens permet realçar els contorns d'una imatge podria ser aquest:

```
// Màscara de convolució Contorns (realçament) expressada com a matriu
float[][] matrix = {
{ -1, -1, -1 },
{ -1, 9, -1 },
{ -1, -1, -1 } };

// Dimensió de la màscara de convolució
int matrixsize = 3;

// Correcció desplaçament
int offset = 0;
```

A la figura 15 tenim el resultat d'aquesta transformació.

Figura 15. Finestra de l'aplicació de l'exemple 14



Com observem, en tots aquests exemples, simplement hem d'anar canviant els coeficients de la màscara de convolució i corregint el valor del desplaçament.

Nota

S'aconsella a l'estudiant implementar totes les màscares estudiades al mòdul "Transformacions espacials lineals", analitzar els resultats i comparar-los amb els resultats obtinguts a Photoshop, per a la seva perfecta comprensió.

4. Transformacions espacials no lineals

Com ja vam veure al mòdul "Transformacions espacials no lineals", hi ha transformacions espacials que no compleixen les condicions de linealitat i, per tant, no poden calcular-se mitjançant una màscara de convolució.

En aquest apartat veurem com aplicar les transformacions espacials no lineals erosió, dilatació, obertura i tancament. Com ja vam avançar a l'apartat 1, ho farem utilitzant el mètode `filter()` de la classe PImage, en comptes de fer les operacions píxel a píxel, com hem fet en els dos darrers apartats.

4.1. Erosió

Ja sabem que l'operador erosió, per a cada píxel de la imatge original, cerca el nivell de gris més petit present en la finestra de treball i assigna aquest nivell de gris al píxel corresponent de la imatge transformada. Per tant, aquesta transformació genera una imatge més fosca que la imatge original.

Anem a veure en detall com podem implementar aquesta transformació a Processing amb el mètode `filtre()`, encara que el codi és pràcticament igual que el codi que hem vist a l'exemple 4.

```
/**
* Exemple 15: Transformació espacial no lineal Erosió
* Francesc Martí, martifrancesc@uoc.edu, 15-04-2016
*
*/

// Declarem un objecte de tipus PImage
PImage img;

void setup() {
// Carreguem la imatge
img = loadImage("faces.png");

// Les mides de la finestra de treball permetran visualitzar
// la imatge original i la imatge filtrada
surface.setSize(2*img.width, img.height);

// La funció draw() s'executarà només una vegada
noLoop();
}

void draw() {
```

```
// Visualitzem la imatge abans de filtrar
image(img, 0, 0);

// Apliquem el filtre, amb el mètode Erosió
img.filter(ERODE);

// Visualitzem la imatge després de filtrar
image(img, img.width, 0);

// Guardem el resultat de la transformació a la carpeta "data" del projecte
img.save(dataPath("imgFilter2.png"));
}
```

A la figura 16 tenim el resultat d'aquesta transformació amb la imatge original a l'esquerra i la imatge erosionada –i, per tant, més fosca– a la dreta.

4.2. Dilatació

Per la seva part, l'operador dilatació realitza l'operació contrària, és a dir, per a cada píxel de la imatge original, cerca el nivell de gris més gran present en la finestra de treball i assigna aquest nivell de gris al píxel amb les mateixes coordenades de la imatge transformada. Per tant, aquesta transformació genera una imatge més clara que la imatge original.

Veiem també un exemple de com podem implementar aquesta transformació a Processing fent servir la funció `filter()`. De fet, el codi és el mateix que el codi de l'operador erosió, però substituint la línia on especifiquem el mètode del filtre.

```
// Apliquem el filtre, amb el mètode Dilatació
img.filter(DILATE);
```

A la figura 17 tenim el resultat d'aquesta transformació. Com podem veure, la imatge transformada és sensiblement més clara que la imatge original.

Figura 17. Finestra de l'aplicació de l'exemple 16

4.3. Obertura i tancament

Per acabar la teoria d'aquest apartat, veurem com programar els filtres obertura i tancament a Processing.

Com hem estudiat, el filtre obertura consisteix a aplicar primer un filtre erosió seguit d'un filtre dilatació, respectant l'element estructural. L'objectiu d'aquesta transformació és eliminar els objectes clars més petits que l'element estructural, però intentant preservar la resta d'informació de la imatge.

El codi d'aquesta transformació és molt senzill, ja que només hem d'aplicar el filtre de Processing dues vegades, la primera amb el paràmetre ERODE i la segona amb el paràmetre DILATE, com podem veure en el següent programa.

```
/**
* Exemple 17: Transformació espacial no lineal Obertura
* Francesc Martí, martifrancesc@uoc.edu, 15-04-2016
```

```
*
*/

// Declarem un objecte de tipus PImage
PImage img;

void setup() {
// Carreguem la imatge
img = loadImage("faces.png");


// Les mides de la finestra de treball permetran visualitzar
// la imatge original i la imatge filtrada
surface.setSize(2*img.width, img.height);

// La funció draw() s'executarà només una vegada
noLoop();
}

void draw() {

// Visualitzem la imatge abans de filtrar
image(img, 0, 0);

// Apliquem el filtre, amb el mètode Erosió
img.filter(ERODE);

// Apliquem el filtre, amb el mètode Dilatació
img.filter(DILATE);

// Visualitzem la imatge després de filtrar
image(img, img.width, 0);

// Guardem el resultat de la transformació a la carpeta "data" del projecte
img.save(dataPath("imgFilter2.png"));
}
```

A la figura 18 podem veure el resultat d'aplicar una transformació obertura a la imatge «faces.png» fent servir aquest programa.

Figura 18. Finestra de l'aplicació de l'exemple 17

El filtre tancament, per la seva part, s'aconsegueix aplicant primer un filtre dilatació seguit d'un filtre erosió. Com sabem, l'objectiu d'aquest filtre és el contrari del filtre obertura: eliminar els objectes foscos més petits que l'element estructural, però –de nou– intentant preservar la resta d'informació de la imatge.

El codi d'aquest filtre és exactament igual que el del programa que acabem de veure, però canviant l'ordre dels filtres.

```
// Apliquem el filtre, amb el mètode Dilatació
img.filter(DILATE);

// Apliquem el filtre, amb el mètode Erosió
img.filter(ERODE);
```

Així doncs, a la figura 19 tenim el resultat d'aplicar una transformació espacial no lineal tancament a la imatge "faces.png" fent servir aquest codi.

Figura 19. Finestra de l'aplicació de l'exemple 18

5. Transformacions geomètriques

El darrer apartat d'aquest mòdul el dedicarem a les transformacions geomètriques.

En els apartats anteriors, ens hem centrat a desenvolupar programes que transformaven els nivells de gris dels píxels d'una imatge (i els colors en alguns casos) per generar una nova imatge. Però en cap cas hem modificat la posició dels píxels de la imatge.

Les transformacions que modifiquen la posició dels píxels de la imatge original reben el nom de transformacions geomètriques, i en aquest apartat veurem com programar algunes d'elles a Processing.

5.1. Zoom i delmació

Començarem l'anàlisi de les transformacions geomètriques a Processing amb el zoom i la delmació. Aquestes dues transformacions permeten augmentar o disminuir el nombre de píxels d'una imatge, provocant un augment o disminució de les seves dimensions, respectivament.

Vegeu també

El zoom i la delmació s'estudien al mòdul "Transformacions geomètriques".

Per realitzar aquestes operacions, a Processing disposem de la funció `resize()`, que, segons el seu paràmetre, aplicarà un zoom o una delmació sobre la imatge. Per a valors més petits que 1 aplicarà una delmació, i per a valors major que 1, un zoom.

Veiem-ho amb un exemple.

```
/**
* Exemple 19: Transformació geomètrica Zoom i Delmació
* Francesc Martí, martifrancesc@uoc.edu, 15-04-2016
*
*/

// Declarem un objecte de tipus PImage
PImage img;

// Per valors > 1 apliquem un zoom, valors < 1 una delmació
float resizeValue = 1.5;

void setup() {
// Carreguem la imatge
img = loadImage("4.1.04.png");
```

```
// Les mides de la finestra de treball han de permetre visualitzar
// la imatge original i la imatge transformada
surface.setSize(int((resizeValue+1)*img.width), int(max(resizeValue,1)*img.height));

// La funció draw() s'executarà només una vegada
noLoop();
}

void draw() {

// Visualitzem la imatge abans d'ampliar
image(img, 0, 0);

// Apliquem la transformació geomètrica "resize"
img.resize(int(resizeValue*img.width), int(resizeValue*img.height));

// Visualitzem la imatge després d'aplicar-li la transformació
image(img, img.width/resizeValue, 0);

// Guardem el resultat de la transformació a la carpeta "data" del projecte
img.save(dataPath("imgFilter2.png"));
}
```

En aquest codi, el valor del paràmetre de la funció `resize()` és `1.5` i, per tant, les dimensions de la imatge transformada és 1,5 vegades més gran que les de la imatge original.

A la figura 20 tenim el resultat d'aquesta transformació, amb la imatge original a l'esquerra i la imatge transformada a la dreta.

Figura 20. Finestra de l'aplicació de l'exemple 19

Com queda remarcat, si volguéssim aplicar una delmació, simplement haurí-em de fer servir un paràmetre amb un valor més petit que 1. Per exemple:

```
float resizeValue = 0.5;
```

A la figura 21 podem veure el resultat d'aplicar una funció `resize()` amb un valor de `0.5` sobre la imatge "4.1.04.png", de nou amb la imatge original a l'esquerra i la imatge transformada a la dreta.

Figura 21. Finestra de l'aplicació de l'exemple 19

Abans de finalitzar aquest apartat, convé fer notar que molts estudiants confonen les funcions `resize()` i `scale()`, ja que semblen fer el mateix.

Si executem el següent codi, podem observar que la finestra de l'aplicació del programa també mostra la imatge reduïda. Aparentment, hem fet la mateixa operació que a l'exemple anterior.

```
/**
* Exemple 20: Transformació Scale
* Francesc Martí, martifrancesc@uoc.edu, 15-04-2016
*
*/

// Declarem un objecte de tipus PImage
PImage img;

// Valor de l'escala
float scaleValue = 0.5;

void setup() {
```

```
// Carreguem la imatge
img = loadImage("4.1.04.png");

// Mides de la finestra de treball
surface.setSize(int(scaleValue*img.width), int(scaleValue*img.height));

// La funció draw() s'executarà només una vegada
noLoop();
}

void draw() {

// Modifica les dimensions del sistema de coordenades, però no modifica
// les dimensions de la imatge
scale(scaleValue);

// Visualitzem la imatge
image(img, 0, 0);

// Guardem la imatge a la carpeta "data" del projecte
img.save(dataPath("img2.png"));
}
```

En canvi, si comparem les imatges que hem guardat amb els dos programes, podem veure que el primer programa guarda la imatge transformada, però, en canvi, el segon programa torna a guardar la imatge original, sense reduir.

La raó és que la funció `scale()` realitza els canvis sobre el sistema de coordenades de la finestra de l'aplicació, no sobre la imatge. Seria equivalent a modificar el valor del zoom en treballar en un document de text: estem modificant l'espai de treball, la forma de veure aquell document, no estem canviant les mides de la font ni cap altre element.

5.2. Translació

Anàlogament a la transformació `scale()` que acabem de veure, tant la translació com la rotació són transformacions que, a Processing, modifiquen les coordenades del sistema, no la imatge en sí. Com hem vist, aquest fet l'hem de tenir sempre present, sobretot si volem guardar el resultat de les transformacions efectuades.

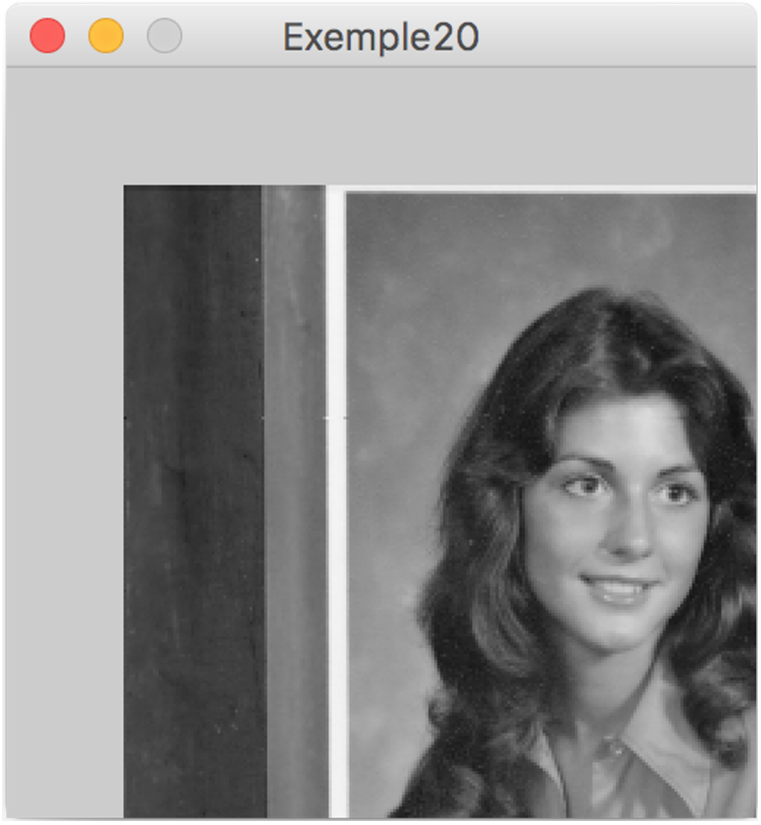
La funció encarregada d'aplicar translacions a Processing és la funció `translate()`. Bàsicament, el que fa aquesta funció és traslladar l'origen del sistema de coordenades (0,0) que, per defecte, es troba a la cantonada superior esquerra de la finestra de l'aplicació, a una altra posició de la finestra.

Veiem això amb un senzill exemple, on desplacem l'origen del sistema de coordenades 40 píxels cap a baix i 40 píxels cap a la dreta.

```
/**
* Exemple 21: Transformació Translació
* Francesc Martí, martifrancesc@uoc.edu, 15-04-2016
*
*/

// Declarem un objecte de tipus PImage
PImage img;

// Coordenades del nou origen
int coorX = 40;
int coorY = 40;

void setup() {
// Carreguem la imatge
img = loadImage("4.1.04.png");

// Mides de la finestra de treball
surface.setSize(img.width, img.height);

// La funció draw() s'executarà només una vegada
noLoop();
}

void draw() {

// Movem el sistema de coordenades
translate(coorX, coorY);

// Visualitzem la imatge
image(img, 0, 0);

// Guardem el contingut de la finestra de l'aplicació com imatge
// a la carpeta "data" del projecte
save(dataPath("img2.png"));
}
```

A la figura 22 tenim el resultat d'aquesta transformació.

Figura 22. Finestra de l'aplicació de l'exemple 21



Com podem veure, ara la imatge no es mostra completa, ja que, en traslladar el sistema de coordenades, part de la imatge cau forma de les dimensions de la finestra de l'aplicació. Una forma de solucionar-ho seria ampliant les dimensions de la finestra

```
// Mides de la finestra de treball
surface.setSize(img.width+ coorX, img.height+ coorY);
```

Però ara la pregunta seria: com guardarem aquesta transformació al nostre disc dur? Com hem vist abans, amb

```
img.save(dataPath("img2.png"));
```

no guardaríem el canvi de posició, ja que la transformació s'efectua sobre el sistema de coordenades, no sobre la imatge. Si el que volem és guardar el que es veu a la finestra de l'aplicació hem de fer servir simplement `save()`.

```
save(dataPath("img2.png"));
```

Per tant, amb el codi `img.save()` guardem el contingut de la imatge –encara que aquesta ni tan sols aparegui a la finestra de l'aplicació– i amb `save()` guardem el contingut de la finestra de l'aplicació.

```
img.save(dataPath("img2.png"));
```

Una cosa semblant passa amb la funció `filter()` explicada anteriorment, i que acostuma a provocar errors en el codi dels estudiants. Si, per exemple, en un programa aplico un filtre amb el codi

```
filter(GRAY);
```

estic aplicant aquest filtre al contingut de la finestra de l'aplicació. No estic aplicant el filtre sobre cap imatge de tipus PImage. Encara que la finestra mostri la imatge, són dues coses diferents. Una cosa és la imatge en si i una altra cosa és el que es visualitza a la finestra de l'aplicació. Si guardem la imatge amb

```
img.save(dataPath("img2.png"));
```

veurem que la guardem sense haver-li aplicat el filtre anterior.

5.3. Rotació

La darrera transformació geomètrica que veurem és la rotació que, com el seu nom ja indica, el que fa és rotar el sistema de coordenades de la finestra de l'aplicació.

La funció que a Processing realitza aquesta operació rep el nom de `rotate()` i el seu paràmetre indica l'angle de rotació expressat en radians.

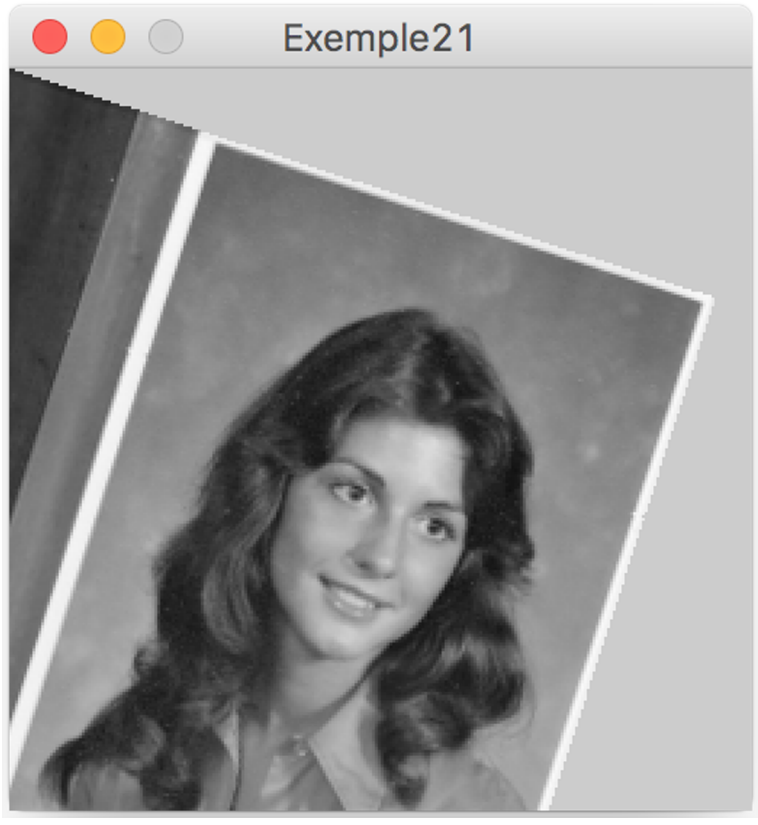
Podem veure un senzill exemple en el següent codi:

```
/**
* Exemple 22: Transformació Rotació
* Francesc Martí, martifrancesc@uoc.edu, 15-04-2016
*
*/

// Declarem un objecte de tipus PImage
PImage img;

// Valor de la rotació
float rotateValue = PI/10;

void setup() {
// Carreguem la imatge
img = loadImage("4.1.04.png");

// Mides de la finestra de treball
surface.setSize(img.width, img.height);

// La funció draw() s'executarà només una vegada
noLoop();
```

```
}

void draw() {

// Rotem el sistema de coordenades
rotate(rotateValue);

// Visualitzem la imatge
image(img, 0, 0);

// Guardem el contingut de la finestra de l'aplicació com imatge
// a la carpeta "data" del projecte
save(dataPath("img2.png"));
}
```

A la figura 23 tenim el resultat d'executar el codi de l'exemple 22.

Figura 23. Finestra de l'aplicació de l'exemple 22



Com podem veure, igual que ens va passar amb la translació, ara la imatge no es mostra completa, ja que en fer la rotació, agafant l'origen de coordenades com a punt fix, un fragment de la imatge queda fora dels límits de la finestra de l'aplicació.

5.4. Composició de transformacions

Per acabar aquest apartat, introduirem el concepte de composició de transformacions i veurem com ens permeten solucionar problemes com el de l'exemple anterior.

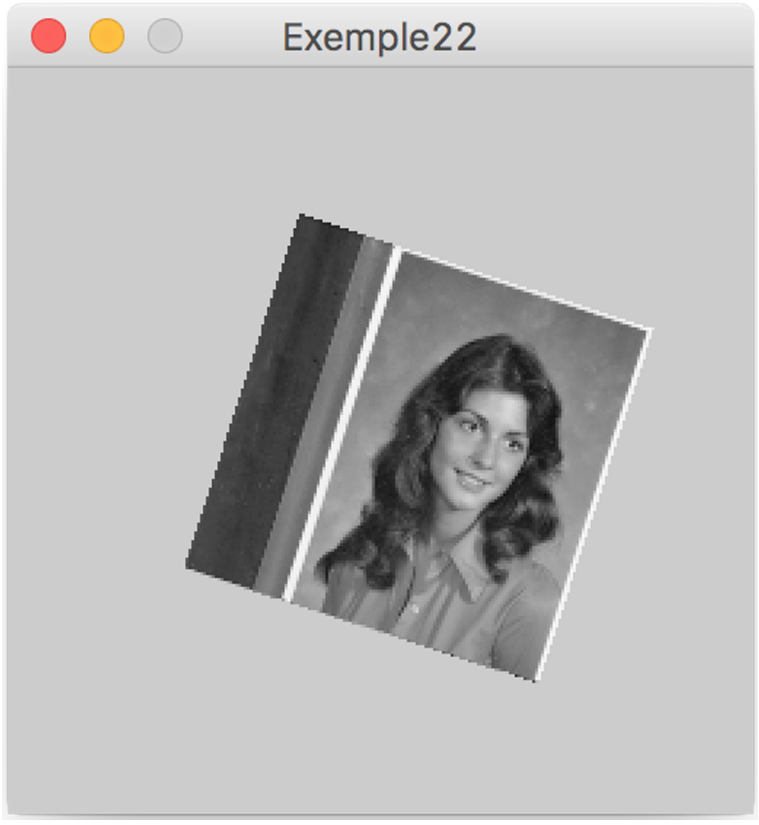
En aquest exemple, efectuem un canvi d'escala, una translació i una rotació. Rotarem i traslladarem la imatge, però la mostrarem sencera a la finestra de l'aplicació aplicant també un canvi d'escala.

```
/**
* Exemple 23: Composició de Transformacions (I)
* Francesc Martí, martifrancesc@uoc.edu, 15-04-2016
*
*/

// Declarem un objecte de tipus PImage
PImage img;

// Valor de l'escala
float scaleValue = 0.5;

// Coordenades del nou origen
int coorX = 200;
int coorY = 100;

// Valor de la rotació
float rotateValue = PI/10;

void setup() {
// Carreguem la imatge
img = loadImage("4.1.04.png");

// Mides de la finestra de treball
surface.setSize(img.width, img.height);

// La funció draw() s'executarà només una vegada
noLoop();
}

void draw() {

// Modifica les dimensions del sistema de coordenades, però no modifica
// les dimensions de la imatge
scale(scaleValue);

// Movem el sistema de coordenades
```

```
translate(coorX, coorY);

// Rotem el sistema de coordenades
rotate(rotateValue);

// Visualitzem la imatge
image(img, 0, 0);

// Guardem el contingut de la finestra de l'aplicació com imatge
// a la carpeta "data" del projecte
save(dataPath("img2.png"));
}
```

A la figura 24 tenim el resultat d'aquesta composició de transformacions.

Figura 24. Finestra de l'aplicació de l'exemple 23



Per acabar, parlarem breument de les funcions `pushMatrix()` i `popMatrix()`, ja que són funcions imprescindibles per controlar les composicions de transformacions geomètriques. Com hem vist, amb les funcions `scale()`, `translate()` i `rotate()` modifiquem el sistema de coordenades de la finestra de l'aplicació. Això pot provocar que, després de realitzar diverses transformacions, no acabem de tenir clara quina és la configuració del sistema, o que es necessitin diversos càlculs per tornar a l'estat inicial.

Les funcions `pushMatrix()` i `popMatrix()` permeten controlar les transformacions geomètriques isolant els seus efectes. Qualsevol transformació que s'apliqui després d'una funció `pushMatrix()` deixarà d'estar activa després d'una funció `popMatrix()`. Cada funció `pushMatrix()` ha de tenir la seva corresponen funció `popMatrix()`, per determinar correctament l'àmbit d'acció de les transformacions.

Anem a veure com funcionen amb un exemple. Com podem veure a l'exemple 23, els efectes de les transformacions `translate()` i `rotate()` estan isolats. Només tenen efecte sobre la primera visualització que efectuem, `image(img, 0, 0)`, no sobre la segona. Això ens permet restablir el sistema de coordenades original, amb l'excepció de l'escala, que, com que no està entre les funcions `pushMatrix()` i `popMatrix()` sí que té efectes permanents.

```
/**
* Exemple 24: Composició de Transformacions (II)
* Francesc Martí, martifrancesc@uoc.edu, 15-04-2016
*
*/

// Declarem un objecte de tipus PImage
PImage img;

// Valor de l'escala
float scaleValue = 0.5;

// Coordenades del nou origen
int coorX = 200;
int coorY = 100;

// Valor de la rotació
float rotateValue = PI/10;

void setup() {
// Carreguem la imatge
img = loadImage("4.1.04.png");

// Mides de la finestra de treball
surface.setSize(img.width, img.height);

// La funció draw() s'executarà només una vegada
noLoop();
}

void draw() {

// Aquesta transformació afecta a les dues imatges que visualitzarem
```

```
  scale(scaleValue);

  pushMatrix();

  // Movem el sistema de coordenades
  translate(coorX, coorY);

  // Rotem el sistema de coordenades
  rotate(rotateValue);

  // Visualitzem la imatge
  image(img, 0, 0);

  popMatrix();

  // Visualitzem un altre cop la imatge
  image(img, 0, 0);

  // Guardem el contingut de la finestra de l'aplicació com imatge
  // a la carpeta "data" del projecte
  save(dataPath("img2.png"));
  }
```

Activitats

Activitats de l'apartat 1

1. Escriu un programa a Processing que mostri a la finestra de l'aplicació quatre imatges, totes amb les mateixes dimensions (pots agafar lliurement quatre imatges de la teoria). Les imatges s'hauran de mostrar d'una en una i, en clicar sobre una imatge, mostrarem la següent, de forma indefinida.

A més, la finestra de l'aplicació s'haurà d'ajustar a les dimensions de les imatges. És a dir, si les dimensions de la imatge són 300 × 200 píxels, la mida de la finestra també haurà de ser 300 × 200 píxels.

2. Escriu un programa a Processing que mostri a la finestra de l'aplicació quatre imatges, totes amb les mateixes dimensions. La primera imatge es mostrarà en prémer la tecla [1], la segona imatge en prémer la tecla [2], etc.

Anàlogament a l'exercici anterior, la finestra de l'aplicació s'haurà d'ajustar a les dimensions de les imatges.

3. Sobre la imatge donada «car.png», quins són els valors RGB del punt (230, 50)? Escriu un programa a Processing que calculi i mostri per pantalla aquest valor.

4. Cerca informació sobre la funció `brightness()` de Processing i repeteix l'exercici 3 fent servir aquesta funció.

5. Escriu un programa a Processing que detecti tots els punts de la imatge «car.png« que tinguin nivell de vermell més petit de 100. Canvia el valor de vermell d'aquests punts a 0 i mostra el resultat a la finestra del programa.

Per exemple, si detectem que els valors RGB d'un punt de la imatge «car.png« és (90, 250, 48) hem de modificar el valor a (0, 250, 48).

En prémer la tecla [A] el programa haurà de guardar el resultat en un nou arxiu de nom «car2.png« al disc dur.

6. Utilitzant la funció `filter()` de Processing, escriu un programa a Processing que realitzi les següents transformacions sobre la imatge «car.png»:

a) Binarització

b) Dilatació

c) Erosió

d) Inversió

Inicialment, el programa ha de mostrar la imatge «car.png» ajustada a la finestra de l'aplicació, de manera que cada vegada que es faci clic sobre aquesta finestra es mostri la imatge amb un dels filtres.

7. Amb ajuda del codi de la pàgina https://processing.org/examples/histogram.html, amplia el codi de l'exemple 4, de manera que també es mostri a la finestra de l'aplicació l'histograma de la imatge transformada a escala de grisos.

Activitats de l'apartat 2

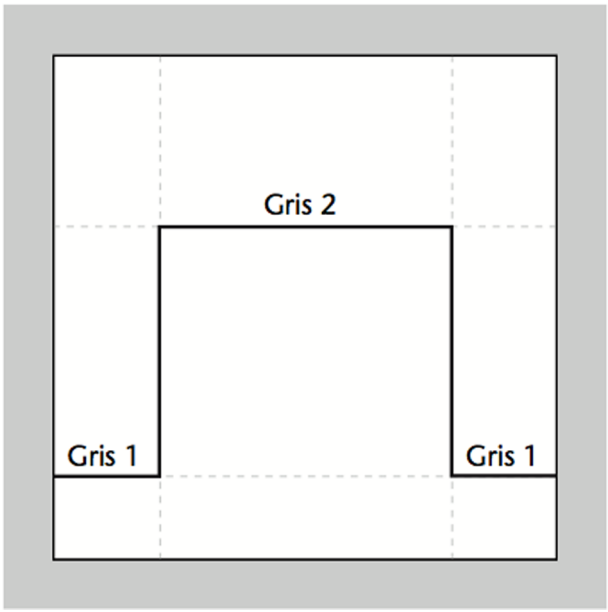
1. a) Escriu un programa a Processing que mostri per pantalla el resultat de restar píxel a píxel la imatge «30.png» a la imatge «28.png». Fes que la finestra de l'aplicació mostri només la nova imatge que es genera.

b) Realitza la mateixa operació a Photoshop i compara els resultats.

2. a) Escriu un programa a Processing que converteixi la imatge RGB «car.png» a escala de grisos fent servir els tres mètodes descrits a http://www.rapidtables.com/convert/image/rgb-to-grayscale. El programa haurà de mostrar les quatre imatges (original i en escala de grisos) a la finestra de treball alhora, per poder comparar les diferències.

b) A Photoshop, canvia la imatge original «car.png» a mode d'escala de grisos amb «Imagen > Modo > Escala de grises» i compara el resultat amb els resultats de l'apartat anterior. Quin dels mètodes anteriors genera un resultat més semblant al de Photoshop?

3. Implementa un programa a Processing que apliqui sobre la imatge «27.png» la transformació puntual determinada per la corba



on, el valor de «Gris 1» és 30 i el valor de «Gris 2» és 127.

4. Modifica el codi de l'exemple 9 de manera que la corba que defineix la transformació puntual d'aclariment i enfosquiment per parts tingui 4 «parts». Aplica el programa sobre la imatge "4.1.02.png" i millora el seu contrast.

Activitats de l'apartat 3

1. Implementa un programa a Processing que sigui capaç d'aplicar totes les màscares vistes a l'apartat 3 a la imatge "4.2.06.png", de manera que, si l'usuari prem la tecla [1], la finestra de l'aplicació mostri la imatge de la figura 11; si prem [2], la imatge de la figura 12; si prem [3] la de la figura 13, etc.

2. Fes que el programa anterior guardi totes les imatges de les transformacions efectuades i compara els resultats obtinguts amb els resultats de Photoshop aplicant les mateixes màscares de convolució. Realitza aquestes comparacions fent servir histogrames.

Activitats de l'apartat 4

1. A Photoshop, realitza les mateixes operacions d'erosió i dilatació que les efectuades sobre la imatge «faces.png» als exemples 15 i 16, i dedueix quina és la forma i dimensió de l'element estructurant utilitzat a Processing.

2. Programa un filtre d'erosió amb Processing, amb un element estructurant quadrat de dimensions 3 x 3, de forma anàloga a com hem programat les transformacions puntuals i les espacials lineals als apartats 1, 2 i 3 (és a dir, sense fer servir la funció `filtre()`).

3. Desenvolupa un programa similar al programa de l'exercici 2, però basat en una transformació espacial no lineal dilatació.

4. Fes un programa a Processing que apliqui un filtre de mediana a una imatge donada. Per programar l'algoritme d'aquesta transformació, i poder ordenar un conjunt de píxels per nivell de gris, pots fer servir la funció `sort()` de Processing.

Activitats de l'apartat 5

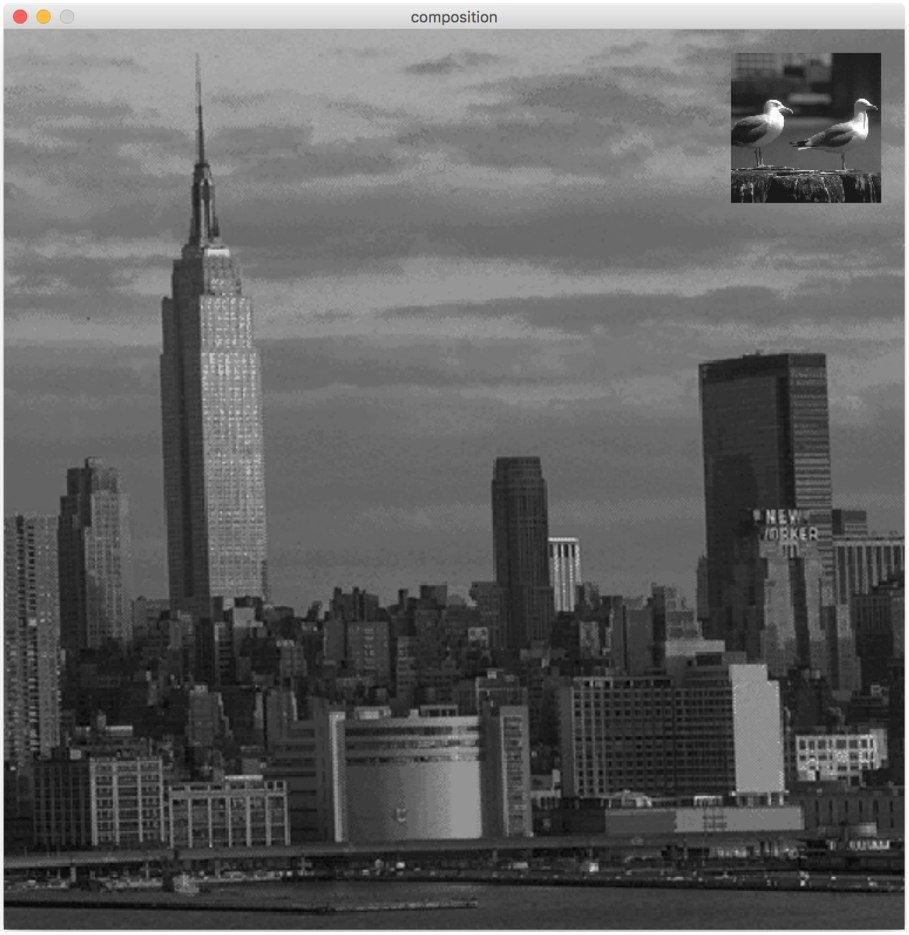
1. a) A Photoshop, sobre la imatge «grid.png», aplica els tres mètodes de delmació estudiats, de manera que es redueixi a la meitat la grandària de la imatge. Quin mètode ofereix millors resultats?

b) Escriu un programa a Processing que faci la mateixa operació amb la funció `scale()`. Quin mètode de delmació fa servir aquesta funció?

c) Repeteix el punt anterior fent servir el mètode `resize()`. Quin mètode de delmació fa servir aquesta funció? És el mateix que el de la funció `scale()`?

2. Escriu un programa a Processing que mostri per pantalla la imatge «peppers.png» i que efectuï una rotació de 45º de la imatge cada cop que se li faci clic. Aplica-li també una transformació `scale()`, quan sigui necessari, per fer que la imatge sempre es mostri sencera a la finestra de l'aplicació.

3. Escriu un programa a Processing que efectuï les transformacions necessàries a les imatges «28.png» i «27.png» perquè es mostrin per pantalla de forma similar a la imatge que veiem a continuació.



Les mides reals de la imatge gran han de ser 768 × 768 píxels i les de la imatge petita 128 × 128, i la imatge interior ha d'estar a 20 píxels de les vores més properes.

Bibliografia

Greenberg, I.; Xu, D.; Kumar, D. (2013). *Processing: creative coding and generative art in processing 2*. Berkeley, California: Friends of ED.

Reas, C.; Fry, B. (2014). *Processing: A programming handbook for visual designers and artists*. Cambridge, Massachusetts: MIT Press.

Shiffman, D. (2015). *Learning processing: A beginner's guide to programming images, animation, and interaction*. Burlington, MA: Morgan Kaufmann.