Processing

Oriol Manau Galtés

PID_00216125

UOC
Universitat Oberta de Catalunya

Índex

Introducció

En aquesta activitat iniciarem el treball amb el llenguatge de programació Processing, un llenguatge que en els últims anys ha anat guanyant adeptes sobretot en l'àmbit de les arts gràfiques gràcies a la seva simplicitat i el fet de ser una alternativa real als projectes fets, per exemple, amb l'Action Script.

Processing

Com hem dit, Processing és un llenguatge de programació orientat al treball amb imatges, animacions i sons, però també és un entorn de desenvolupament que ens ofereix un conjunt complet d'eines per a poder fer els nostres projectes d'una manera fàcil i còmoda.

El llenguatge està basat en Java, n'aprofita l'estructura i potència però el simplifica perquè sigui molt senzill treballar-hi i evita la necessitat de tenir un coneixement profund del llenguatge per a poder obtenir bons resultats. En el cas que tinguem un coneixement més profund de Java el podrem aprofitar per esprémer encara més el llenguatge i aprofitar-ne els avantatges, tot i que no és necessari per a poder obtenir bons resultats.

En estar dissenyat per a ser senzill, ens permet utilitzar-lo per a introduir-nos en la programació d'una manera progressiva i obtenir resultats molt ràpidament. És per això que aquesta activitat seguirà una estructura d'ensenyament basada en exemples que s'aniran completant de mica en mica, afegint nous elements i formes d'organització del codi per a poder arribar al final amb una bona base i poder afrontar qualsevol projecte.

Documentació oficial i altres referències

1) Documentació oficial

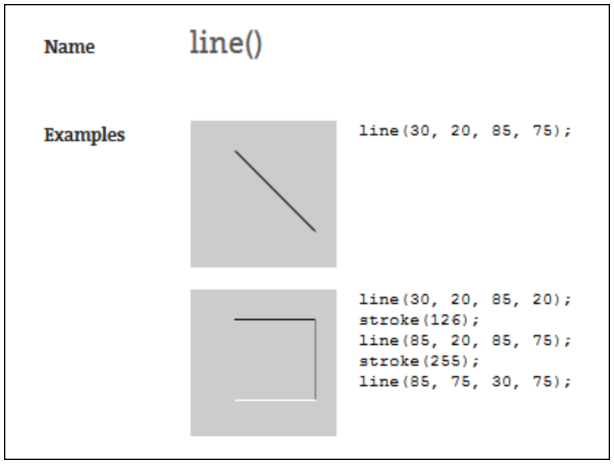
El primer lloc on hem de buscar la informació és en el web oficial, on trobareu la informació més actualitzada i una gran quantitat de referències a exemples i altres llocs on podeu trobar informació extra.

Nota

Processing.org/

Un dels apartats del web més útils és el de la referència del llenguatge, ja que aquí podrem obtenir tota la informació de com cal treballar amb les diferents opcions que ens ofereix el llenguatge. Per exemple, en la figura 1 podem veure una part de la referència per a la funció *line* que ens permet dibuixar línies.

Figura 1. Referència en línia



2) Documentació a Mosaic

Nota

Mosaic.uoc.edu

A part de la documentació oficial podeu trobar molta informació a la revista Mosaic, on cada vegada hi ha més articles de Processing excel·lents per a veure'n característiques concretes o com s'han solucionat problemes diversos utilitzant aquest llenguatge, per la qual cosa és molt recomanable fer-hi una ullada tant per veure'n el funcionament com per obtenir idees del que podem arribar a fer.

A continuació teniu una llista d'alguns dels articles disponibles que podeu consultar, però n'hi ha molts més:

- http://mosaic.uoc.edu/2012/04/30/introduccion-a-Processing
- http://mosaic.uoc.edu/2013/02/19/cursor-animado-con-lenguaje-de-programacion-Processing/
- http://mosaic.uoc.edu/2013/02/27/caleidoscopio-con-lenguaje-de-programacion-Processing/
- http://mosaic.uoc.edu/2013/02/27/sistema-de-particulas-con-lenguaje-de-programacion-Processing/
- http://mosaic.uoc.edu/2013/02/27/colisiones-y-particulas-con-lenguaje-de-programacion-Processing/
- http://mosaic.uoc.edu/2013/02/27/animacion-en-base-a-transformaciones-homogeneas-con-lenguaje-de-programacion-Processing/
- http://mosaic.uoc.edu/2013/02/27/Processing-practico/
- http://mosaic.uoc.edu/2013/03/26/Processing-practico-parte-2-de-3/
- http://mosaic.uoc.edu/2013/04/24/Processing-practico-parte-3-de-3/
- http://mosaic.uoc.edu/2013/03/25/carga-de-un-modelo-3d-con-lenguaje-de-programacion-Processing/
- http://mosaic.uoc.edu/2013/03/25/imagenes-y-luts-en-programacion-Processing/
- http://mosaic.uoc.edu/2013/03/25/efecto-dinamico-de-pointillism-en-imagenes-via-programacion-Processing/

- http://mosaic.uoc.edu/2013/03/25/sonido-con-lenguaje-de-programacion-Processing-primer-ejemplo/
- http://mosaic.uoc.edu/2013/03/25/sonido-con-lenguaje-de-programacion-Processing-segundo-ejemplo/
- http://mosaic.uoc.edu/2013/04/24/realidad-aumentada-con-lenguaje-de-programacion-Processing/
- http://mosaic.uoc.edu/2013/04/24/deteccion-de-colisiones-con-lenguaje-de-programacion-Processing/
- http://mosaic.uoc.edu/2013/04/24/acceso-a-webcam-y-luts-en-programacion-Processing/
- http://mosaic.uoc.edu/2013/04/24/acceso-a-webcam-y-procesado-en-programacion-Processing/
- http://mosaic.uoc.edu/2013/04/24/operaciones-con-imagenes-en-programacion-Processing/

3) Altres fonts d'informació

En aquest cas ningú no tindrà cap sorpresa si agafem com a tercera font d'informació Internet, on podrem trobar una gran quantitat d'informació sobre el llenguatge, com per exemple el vídeo introductori que podem veure en aquest enllaç, o llibres disponibles en línia com *Processing, un lenguaje al alcance de todos*.

Instal·lació

El primer que ens caldrà fer serà descarregar-nos el programa del web oficial per a poder-lo instal·lar en el nostre ordinador. Per a fer-ho hem de seguir aquests passos:

Nota

Processing.org/

- Anem a l' apartat de descàrregues, on podem decidir fer una donació per a donar suport al desenvolupament d'aquest projecte de codi obert.

Figura 2. Pàgina inicial de descàrrega

- Seleccionem *download* i accedirem a la pàgina on podrem seleccionar el sistema operatiu del nostre ordinador i la versió de Processing que volem descarregar.

Figura 3. Pàgina de descàrrega, selecció del sistema operatiu

- Un cop finalitzada la descàrrega només caldrà descomprimir el fitxer i apareixerà el directori on hi ha el programa.

- Per a executar-lo només caldrà entrar dins del directori creat i executar el fitxer *processing.exe* en el cas d'utilitzar el Windows o l'equivalent de cada sistema operatiu, i ja ens apareixerà l'entorn de desenvolupament que utilitzarem per a fer les activitats.

Un cop s'hagi aconseguit executar l'aplicació ja estem preparats per a elaborar el nostre primer programa.

1. Estructura bàsica d'un programa

En aquest apartat analitzarem l'estructura i els elements bàsics que hi ha d'haver en tots els programes perquè puguin funcionar correctament.

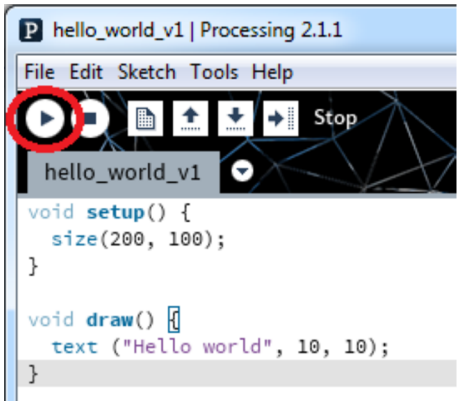
1.1. "Hola món"

Obriu l'editor del Processing i escriviu el codi següent:

```
void setup() {
   size(200, 100);
}

void draw() {
   text ("Hello world", 10, 10);
}
```

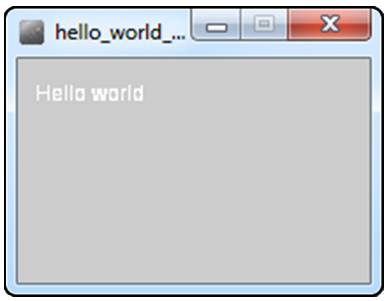
Un cop escrit executeu el codi prement el botó *play* de la interfície.

Figura 4. Execució d'un programa



En fer-ho s'obrirà una finestra nova on podrem llegir el text "Hello world".

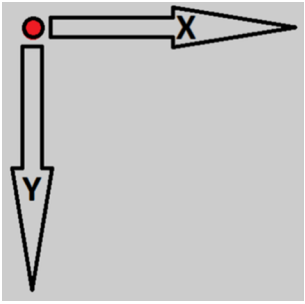
Figura 5. Finestra del programa



Vegem els elements que formen el programa:

1) La funció *setup* ens serveix per a inicialitzar els diferents paràmetres que utilitzarem en el codi i sempre és la primera funció que s'executa. Dins d'aquesta funció la primera línia sempre és una crida a la funció *size(x, y)*, que és la funció que definirà la mida de la finestra gràfica que apareixerà en executar el codi. Els paràmetres que rep la funció indiquen la mida horitzontal i vertical, respectivament, i cal tenir en compte que la posició (0, 0) és la cantonada superior esquerra. Per tant, les coordenades *x* van d'esquerra a dreta i les coordenades *y* van de dalt a baix.

Figura 6. Coordenades de la pantalla



2) La funció *draw* és on definirem els elements que volem que es pintin a la pantalla i des d'on es controlaran les diferents accions que aniran passant durant l'execució del programa. En l'exemple l'únic que fem és cridar la funció *text* i indiquem el text que volem escriure entre cometes dobles i la posició dins de la pantalla on volem que aparegui.

1.2. Elements bàsics

1.2.1. Inserció de comentaris

Per a poder entendre millor el codi que fem és important anar-lo comentant. Per exemple, podem indicar quin és l'objectiu del codi que escrivim, la funcionalitat d'una variable que declarem o els valors correctes que hauria de retornar una funció determinada, entre moltíssimes altres coses.

Fixeu-vos en aquest codi:

```
/*
 * Hello world with comments
 * Version: 2
 */

// Main initialization function
void setup() {
   size(200, 100);  //Window size
```

```
}

void draw() {
   text ("Hello world", 10, 10);
}
```

Per a afegir un comentari de diverses línies utilitzem la combinació */* text del comentari */*. En el codi podem veure que a part d'indicar el principi i final del comentari amb /* i */ hem afegit un * al principi de cada línia, això és opcional però s'acostuma a fer, per seguir l'estil recomanat per a Java, i és útil per a tenir clar que forma part d'un comentari.

Una altra manera d'afegir comentaris és amb //. En aquest s'interpreta que tot allò que hi hagi entre // i el final de la línia és un comentari. Per tant, és útil per a comentar una única línia o afegir algun comentari a continuació d'alguna instrucció.

1.2.2. Declaració de variables

Les variables són contenidors d'informació que utilitzarem al llarg del nostre programa per referenciar dades, fer càlculs, emmagatzemar informació...

Observeu aquest codi:

```
String message;

void setup() {
   size(200, 100);
   message = "Hello world";
}

void draw() {
   text (message, 10, 10);
}
```

En aquest cas hem afegit una variable de tipus *string* que es diu *message*, i l'hem declarada al principi, tot just abans de la funció *setup*. Les variables sempre les definirem d'aquesta manera, indicant el tipus de dada que emmagatzema (en aquest cas una cadena de text) i el nom que farem servir per a referenciar-la.

String

Recordeu que *string*, o cadena de caràcters, és una llista de caràcters, per exemple, una paraula o una frase.

A part de declarar la variable l'hem d'inicialitzar, i això ho fem dins de la funció *setup* i indiquem que la variable *message* serà igual a un text determinat. En aquest cas concret en què treballem amb un text utilitzem les "dobles cometes" per a limitar-lo.

Ara ja només ens falta utilitzar la variable que hem definit. Per a fer-ho escrivim el nom de la variable on voldríem que hi hagués el valor, en aquest cas dins de la funció *text*, i ja ho tenim fet. En el moment que s'executi aquesta funció en escriure el text mirarà el valor de la variable per a escriure el missatge emmagatzemat.

1.2.3. Funció setup

La funció *setup* s'utilitza per a inicialitzar les variables i la resta de paràmetres que s'utilitzaran al llarg del programa. La seva característica principal és que només s'executa una única vegada al principi de tot, i és per aquest motiu que la utilitzem per a inicialitzar els valors.

Com hem dit anteriorment, la primera instrucció que s'ha d'executar dins d'aquesta funció és per a definir la dimensió de la finestra gràfica:

```
void setup() {
   size(200, 100);
   ....
   ....
}
```

A partir d'aquí podem continuar amb la resta d'inicialitzacions que ens calguin. En general hi ha una sèrie de paràmetres globals que acostumem a definir:

1) ***size (x, y)***: com hem dit, aquesta és la funció que utilitzem per a indicar les dimensions de la finestra gràfica. En els exemples sempre hem utilitzat dos valors per a indicar la mida, però proveu a substituir la crida a la funció per aquest codi:

```
size(displayWidth, displayHeight);
```

Com podeu veure, en executar el programa la finestra és molt més gran, i de fet és de la mateixa mida que la imatge que mostra el vostre monitor. Això és així perquè *displayWidth* i *displayHeight* són unes variables pròpies de Processing que s'inicialitzen automàticament amb la dimensió actual del monitor, i per tant són molt útils quan volem que una aplicació s'executi a pantalla completa.

2) ***frameRate (x)***: és la freqüència de refresc que utilitzarem per a redibuixar el contingut de la pantalla. En parlar de la funció *draw* aprofundirem més en aquest concepte.

3) *background (r, g, b)*: és la funció que utilitzarem per a definir el color del fons de la pantalla; en aquest cas li passem el color utilitzant la notació vermell, verd, blau. Proveu a afegir aquest codi i observeu com el fons de la finestra en comptes de gris és vermell.

```
background (255, 0, 0);
```

4) *stroke (r, g, b)*: similar a la funció *background*, en aquest cas *stroke* ens permet definir el color per defecte que utilitzarem per a dibuixar els elements gràfics; proveu d'afegir aquesta línia de codi dins de *setup*:

```
stroke (0, 255, 0);
```

I aquesta altra dins de *draw*:

```
line (0, 0, 200, 100);
```

Recordeu que podeu consultar més detalls i exemples d'aquestes funcions a la referència en línia de la web del Processing o des del menú **help** del programa mateix, on també trobareu la referència fora de línia.

1.2.4. Funció *draw*

Un cop finalitzada l'execució de la funció *setup*, tots els programes fets amb el Processing continuen amb la funció *draw*, i el més important de tot és que aquesta funció s'anirà executant contínuament fins que finalitzi el programa.

És a dir, la funció *draw* s'estarà executant en un bucle infinit mentre ningú no li digui que ha de parar. A més, no ho farà de manera descontrolada, sinó que s'executarà tantes vegades per segon com ho hàgim definit amb la funció *frameRate (x)* que hem comentat en l'apartat anterior.

Dins d'aquesta funció serà on definirem totes les accions per fer, des de pintar els elements de la pantalla fins a fer tots els càlculs necessaris per a continuar amb l'execució correcta del programa.

Repetició infinita

És important que ens quedi clara la idea de repetició quan parlem de la funció *draw*, ja que l'estructura general del programa en depèn directament i fa que a vegades pugui ser una mica confús intentar entendre què fa el nostre codi.

Suposem aquest exemple:

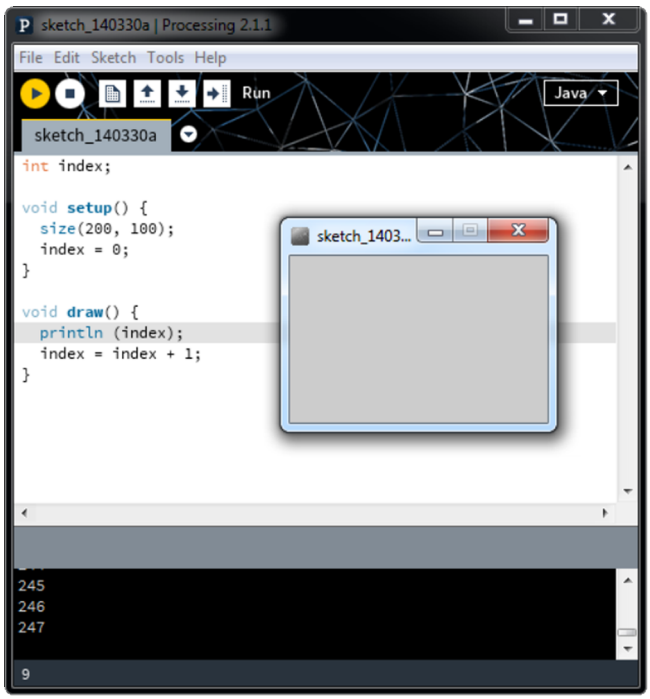
```
int index;


void setup() {
```

```
    size(200, 100);
    index = 0;
}

void draw() {
    println (index);
    index = index + 1;
}
```

Podem veure que hem declarat una variable numèrica *index* (ho veurem amb més detall en un capítol posterior) i la inicialitzem dins de la funció *setup*. A partir d'aquest moment es comença a executar la funció *draw* de manera continuada, sense parar de repetir-se, però com afecta això a la variable *index*, i sobretot, quin valor té en cada moment?

Executeu el programa i veureu una cosa similar a això:

Figura 7. Execució del programa



S'obrirà una finestra grisa, que és el nostre programa, i a la part inferior de l'editor de codi començarà a aparèixer una seqüència numèrica: en el cas de la imatge hi podem veure 245, 246, 247. Per tant, després de crear la variable i inicialitzar-la a 0 comencem a executar sense parar dues instruccions:

- Escrivim el valor de la variable.
- Incrementem el valor de la variable.

De moment no apareix la seqüència numèrica a la finestra gràfica del programa; això és així perquè estem utilitzant la instrucció *println (index)*, que s'encarrega d'escriure directament el valor de la variable *index* en aquest espai que tenim sota de l'editor del codi. Aquesta funció *println ()* és molt útil per a veure l'estat de les variables i ensenyar missatges que ens poden servir mentre estem fent el programa per a comprovar que tot funciona bé o detectar possibles problemes.

Aprofitem aquest mateix codi i afegim aquesta línia dins de la funció *setup*:

```
frameRate (1);
```

Ara, en executar el programa podreu veure que els valors van apareixent cada 1 segon, acabem de definir el nombre de repeticions per segon que volem que s'executi la nostra funció *draw*. Comproveu com canviant el valor es modifica la velocitat amb què es repeteix la funció, i per tant els nombres apareixen més o menys ràpidament.

Importància de l'ordre dels elements

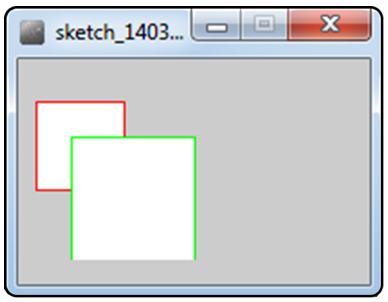
En afegir els diferents elements gràfics que volem pintar dins de la funció *draw* cal tenir en compte l'ordre utilitzat, ja que en dependrà el resultat final que obtinguem.

Utilitzem aquest codi d'exemple:

```
void setup() {
   size(200, 100);
}

void draw() {
   stroke (255, 0, 0);
   rect (10, 10, 50, 50);
   stroke (0, 255, 0);
   rect (30, 30, 70, 70);
}
```

En executar-lo podeu veure aquest resultat:

Figura 8. Ordre de pintar



Primer de tot dibuixem un quadrat de color vermell i després un de color verd, i a efectes pràctics l'últim element que pintem queda per sobre de tot el que ja hi havia pintat amb anterioritat. Això és el que s'anomena *algorisme del pintor*, ja que un pintor primer dibuixa els elements més allunyats i després va afegint la resta dels elements més propers. Si modifiquem el codi i intercanviem l'ordre dels quadrats el de color verd quedarà per darrere.

```
void draw() {
   stroke (0, 255, 0);
   rect (30, 30, 70, 70);
   stroke (255, 0, 0);
   rect (10, 10, 50, 50);
}
```

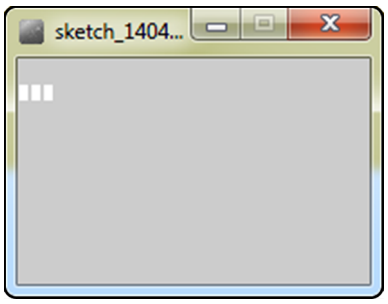
És important recordar aquesta manera de pintar els elements, ja que quan n'hi comenci a haver un nombre més important poden aparèixer efectes estranys si no ho tenim en compte i planifiquem correctament l'ordre de pintar.

Ara combinarem aquests últims conceptes, el de repetició de la funció *draw* i el de l'ordre de pintar dels elements; utilitzarem aquest codi:

```
int index;

void setup() {
   size(200, 150);
   frameRate (1);
   index = 0;
}

void draw() {
   println (index);
   text (index, 0, 10);
   index = index + 1;
}
```

És molt similar al que hem vist anteriorment, però hem afegit la instrucció *text*, que ens permet escriure un missatge a la finestra del programa indicant-ne el text i la posició. En executar el programa veureu que tenim un problema:

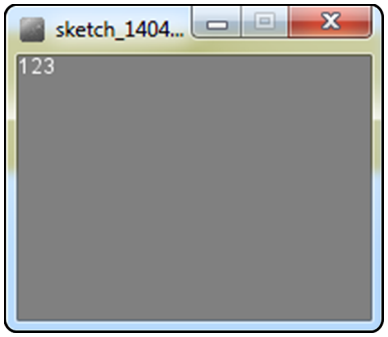
Figura 9. Text gràfic borrós



Ens apareix la finestra, però en comptes de nombres tenim una espècie de taca que no es pot llegir correctament, i per a solucionar-ho farem un petit canvi en la funció *draw*:

```
void draw() {
   background (128, 128, 128);
   println (index);
   text (index, 0, 10);
   index = index + 1;
}
```

Figura 10. Text gràfic visible



Per què es pinta bé el text ara? Com hem dit, quan pintem alguna cosa l'estem apilant per sobre del que ja hi havia, i en el nostre cas estàvem escrivint un text sobre d'un altre, i un altre, i un altre... i per tant estàvem obtenint el mateix efecte que si agafem un tros de paper i hi escrivíssim un text sobre d'un altre.

En afegir la crida *background* al principi de tot, fem que just al començar a repintar la nostra finestra fem una repintada de tot el contingut amb un color determinat. L'efecte és el mateix d'esborrar el nostre paper abans d'escriure-hi res més, i per tant el nou contingut es podrà veure perfectament.

Afegim aquesta nova instrucció abans d'escriure el text, per a poder-ne modificar la mida:

```
textSize (30);
```

En executar el codi veureu que caldrà modificar la posició on pintem el text, perquè en ser de mida més gran, se surt de la pantalla, però a partir d'ara ja podrem controlar-ne la mida per a adaptar-la a les nostres necessitats.

2. Tipus de dades bàsiques

En tots els programes necessitarem desar informació en algun moment. Per a fer-ho disposarem de diferents tipus de variables que ens permetran desar-la i disposar-ne en qualsevol moment. A continuació veurem les més bàsiques que utilitzarem en els propers apartats.

2.1. Definició i inicialització de variables

Abans de poder desar o utilitzar una variable cal que la definim i la inicialitzem, per a mantenir una estructura clara i que ens permeti entendre millor el codi ho farem de la manera següent:

```
tipus_de_variable nom_de_variable;

void setup() {
   nom_de_variable = valor_inicial;
}

void draw() {
}
```

Al principi del codi definirem les variables que utilitzarem, indicant el tipus de dada que contindrà i el nom que utilitzarem per a fer-ne referència.

Dins de la funció *setup* inicialitzarem el valor. Tot i que en la majoria de casos no és estrictament necessari, sí que és molt recomanable i ens pot estalviar més d'un problema.

2.2. Variables: booleanes

Les variables booleanes són aquelles que ens permeten indicar el concepte de cert o fals, i per tant actuen com un interruptor que podrem utilitzar per a indicar si s'ha complert alguna condició o si hem de fer alguna tasca determinada.

```
boolean a;

void setup() {
   a = true;
   a = false;
}
```

En el codi podem veure com definim la variable *a*, que després podem inicialitzar amb el valor cert o fals.

Aquest tipus de variables s'utilitzen molt amb les estructures de control, com per exemple els blocs *if* que veurem més endavant.

2.3. Variables: nombres enters

Els enters ens permeten definir nombres sense decimals, com per exemple l'índex que hem utilitzat en els exemples anteriors per a fer un comptador.

```
int index;

void setup() {
   index = 0;
}
```

Naturalment, un cop definida una variable numèrica hi podrem fer qualsevol tipus d'operació matemàtica.

```
int index;
int doble;

void setup() {
   index = 0;
   doble = 0;
}

void draw() {
   println ("index: "+ index +" valor doble: " + doble);
   index = index + 1;
   doble = index * 2;
}
```

Per exemple, en aquest cas escrivim a la finestra de missatges del programa el valor actual de la variable *índex* i el seu valor doble.

Fixeu-vos com ho hem fet per a combinar text nostre i diferents variables en el missatge que hem escrit:

```
println ("index: "+ index +" valor doble: " + doble);
```

Quan volem escriure un text nostre utilitzem les cometes dobles per a delimitar-lo, i quan volem escriure una variable n'indiquem el nom directament, però si volem concatenar text i variables utilitzem l'operació suma i construïm la frase que volem ensenyar.

2.4. Variables: nombres decimals

Els nombres amb decimals els definirem utilitzant el tipus *float*, tot i que tenim altres opcions, però ens basarem en aquest tipus, ja que és el que utilitzen les funcions de Processing quan han de treballar amb decimals.

```
float decimal;

void setup() {
   decimal = 0.0;
}
```

Un detall important en treballar amb decimals és el de remarcar-ho indicant-ho explícitament; en el cas de la inicialització utilitzem el valor **0,0** per a fer-ho evident. És important per a evitar problemes com els que genera el codi següent:

```
int a;
int b;
float result;

void setup() {
   a = 11;
   b = 2;

   result = a / b;
   println ("a/b = " + result);

   result = 11 / 2;
   println ("a/2 = " + result);

   result = a / 2.0;
   println ("a/2 = " + result);
}
```

En executar el codi veureu el resultat següent:

```
a/b = 5.0
a/2 = 5.0
a/2 = 5.5
```

Quan agafem dues variables enteres i les dividim el resultat que ens retorna és un nombre enter, i per tant n'arrodoneix el valor; aquest és el cas de les dues primeres operacions.

En canvi, si indiquem que un dels operands és un decimal ja ens retorna un resultat amb decimals, i en aquest cas ho hem fet dividint entre **2,0**.

És important recordar-ho perquè pot produir resultats estranys i a vegades aquest tipus de problema és difícil de detectar.

2.5. Variables: conjunts

Els conjunts o *arrays* són una agrupació de dades que podrem emmagatzemar i als quals podrem accedir des d'una única variable. En definir una variable del tipus *array* cal que li diguem el tipus d'informació que contindrà, i ho fem de la manera següent:

```
int[] int_array;
```

En aquest exemple definim un *array* d'enters, indicant el tipus de dades que conté l' *array* i a continuació afegint ***[]*** abans d'escriure el nom de la variable.

Podem inicialitzar-los de dues maneres diferents en funció de si coneixem totes les dades que han de tenir des del principi o si les volem anar afegint després.

2.5.1. Inicialització amb tots els valors coneguts

En aquest cas la inicialització s'ha de fer en el mateix moment de declarar la variable, i s'indiquen tots els elements que el formen separats per comes i entre claudàtors.

```
int[] int_array = { 10, 20, 30 };

void setup() {
}
```

2.5.2. Inicialització sense valors

En aquest cas haurem d'indicar quants elements volem que contingui l' *array* com a màxim.

```
int[] int_array;

void setup() {
   int_array = new int [3];
}
```

Utilitzem la paraula clau *new*, el tipus de dades que tindrà l' *array* i el nombre d'elements entre claudàtors. Amb això ja tenim l' *array* preparat per a poder-hi desar dades, però, al contrari del cas anterior, de moment no en té cap. Ara veurem com podem desar i llegir informació.

2.5.3. Accés a les dades de l' array

Independentment de com hàgim inicialitzat l' *array*, ara ja el podrem utilitzar per a desar-hi informació i accedir-hi quan ens calgui, i per fer-ho ens cal indicar quin element o posició volem modificar o llegir. Per exemple, si volem desar el valor 444 a la posició 1 ho fem de la manera següent:

```
int_array[1]=444;
```

I si volguéssim escriure aquest valor el podríem consultar fent:

```
println (int_array[1]);
```

La posició que estem consultant o modificant és l'índex de l' *array*, i ens indica la casella on fem aquestes operacions. L'índex d'un *array* comença pel valor 0 i va fins al nombre d'elements que té l' *array* menys 1.

Per exemple, en les inicialitzacions que hem vist definíem un *array* de 3 posicions; per tant, el seu índex anirà de 0 a 2.

```
int[] int_array = { 10, 20, 30 };

void setup() {
   println (int_array[0]);
   println (int_array[1]);
   println (int_array[2]);
}
```

Aquí podem veure que per a accedir als 3 enters que tenim desats utilitzem els índex 0, 1 i 2.

Per a consultar el nombre d'elements que pot guardar un *array* podeu utilitzar:

```
println (int_array.length);
```

En afegir *.length* després d'una variable *array* ens retornarà aquesta informació, que després podrem utilitzar per a accedir a les dades (en veurem exemples més endavant).

2.6. Variables: cadenes de text

Com hem vist, quan volem escriure un text utilitzem les cometes dobles per a passar-lo a la funció de pintar, però si el volem desar en una variable per utilitzar-lo en un altre moment hem d'utilitzar el tipus *String*. Com en el cas anterior, per a inicialitzar una cadena de text utilitzarem *new*, de manera que tindrem un codi d'aquest estil:

```
String text;


void setup() {
   text = new String ("Hello");
   println (text);
}
```

Per tant, primer definim el nom de la variable i després la inicialitzem dintre de la funció *setup*. Tingueu en compte que la inicialització només cal fer-la una vegada, i per tant, si volem canviar el text no cal tornar a utilitzar el format *new String* (*"text nou"*), sinó que es pot fer directament:

```
void setup() {
   text = new String ("Hello");
   text = "bye!";
   println (text);
}
```

En aquest cas actualitzem el valor igualant-lo directament al nou text limitat per cometes dobles.

A part de poder assignar un text o llegir-lo, tenim altres funcionalitats, com poder fer comparacions, recuperar una part del text, etc., però això ho anirem veient en els exemples futurs.

2.7. Variables: color

El Processing és un llenguatge que té com a objectiu el treball amb gràfics, i per tant el color és un paràmetre molt important que ens interessarà poder desar per utilitzar-lo en els nostres programes.

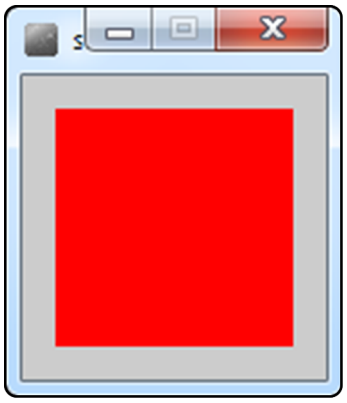
La manera més fàcil de definir una variable de color és aquesta:

```
color c;


void setup() {
   c = color (255, 0, 0);
}
```

```
void draw() {
   background (c);
}
```

Fixeu-vos que després de definir la variable utilitzem una funció que també es diu *color(r, g, b)*, i a la qual passem els components vermell, verd i blau del color que estem definint (valors entre 0 i 255), i a continuació ja podem utilitzar la nostra variable per a definir el color de fons, per exemple.

Figura 11. Variable color



Si en algun moment voleu consultar el valor del color i l'escriviu directament per pantalla obtindreu un valor molt difícil d'interpretar, però hi ha una solució molt simple:

```
color c;

void setup() {
   c = color (255, 120, 60);

   println ("Direct value: " + c);
   println ("Hexadecimal value: " + hex(c));
}
```

I la sortida per pantalla és:

```
Direct value: -34756
Hexadecimal value: FFFF783C
```

Si ensenyem el valor directe és un nombre negatiu, i és molt complicat d'interpretar a quin color real fa referència, però si utilitzem la funció *hex()* el que fem és convertir aquest color a hexadecimal, i en aquest cas la interpretació és molt més senzilla, ja que obtenim un nombre que podem descompondre en 4 blocs ben definits AARRGGBB, que es corresponen a:

- **AA**: Valors entre 00 i FF (0 a 255 en decimal), defineixen el canal alfa, o el que és el mateix, la transparència del color.

- **RR**: Valors entre 00 i FF (0 a 255 en decimal), defineixen el canal vermell.
- **GG**: Valors entre 00 i FF (0 a 255 en decimal), defineixen el canal verd.
- **BB**: Valors entre 00 i FF (0 a 255 en decimal), defineixen el canal blau.

Per tant, ara sí que és més fàcil poder interpretar aquest color, i per tant si en algun moment n'hem de comprovar algun ens serà més intuïtiu.

2.8. Exemple: "Hola món" utilitzant variables

Ara que ja coneixem les variables bàsiques, evolucionarem el codi que havíem fet inicialment per a afegir-hi alguns elements més interessants. Començarem a partir d'aquest punt:

```
void setup() {
   size(300, 200);
}

void draw() {
   textSize (30);
   text ("Hello world", 10, 30);
}
```

El primer que farem serà potser el més lògic, substituir la cadena de text per una variable *string*:

```
String message;

void setup() {
   size(300, 200);

   message = new String ("Hello world");
}

void draw() {
   textSize (30);
   text (message, 10, 30);
}
```

Naturalment, en executar el codi obtenim el mateix resultat, però utilitzar una variable ens pot ser molt útil quan hem d'escriure el mateix text en diferents llocs del programa. En aquests casos ens assegurem que si s'ha de canviar el text en algun moment només caldrà modificar-lo en un únic lloc (dins de *setup*, on l'hem inicialitzat), i a partir d'aquest moment es veurà reflectit el canvi a tots els llocs on utilitzàvem aquesta variable.

Ara farem un altre canvi; mireu el codi següent:

```
String message;
int size;

void setup() {
   size(300, 200);

   message = new String ("Hello world");
   size = 10;
}

void draw() {
   textSize (size);
   text (message, 10, 30);
}
```

Hem fet el mateix que en el text, però amb la mida. Això ens permet fer:

```
String message;
int size;

void setup() {
   size(300, 200);
   frameRate (1);

   message = new String ("Hello world");
   size = 10;
}

void draw() {
   textSize (size);
   text (message, 10, 30);
   size = size + 1;
}
```

Ara el text va creixent de mica en mica, però hi ha un problema i no es veu bé, oi? Recordeu com ho solucionàvem? Exacte, amb la funció *background*, i aquesta vegada també farem servir una variable del tipus color:

```
String message;
int size;
color c;

void setup() {
   size(300, 200);
   frameRate (1);

   message = new String ("Hello world");
```

```
    size = 10;
    c = color (100, 100, 100);
}

void draw() {
    background (c);
    textSize (size);
    text (message, 10, 30);
    size = size + 1;
}
```

Ara sí, ja el podem llegir sense problemes i veiem com va creixent. Naturalment, també podríem utilitzar variables per a definir la posició del text, de manera que en podríem modificar la posició de la mateixa manera que modifiquem la mida.

Per acabar aquest exemple, farem que el missatge vagi alternant entre dos textos diferents, i per això utilitzarem un *array* de textos:

```
String[] message;
int size;
color c;

void setup() {
    size(300, 200);
    frameRate (1);

    message = new String [2];
    message[0] = new String ("Hello");
    message[1] = new String ("world!");
    size = 10;
    c = color (100, 100, 100);
}

void draw() {
    background (c);
    textSize (size);
    text (message[(size % 2)], 10, 30);
    size = size + 1;
}
```

Mirem en detall què hem fet:

- Definim la variable *array*, que conté informació de tipus *string*.

- *message = new String [2]*: inicialitzem l' *array* i diem que tindrà lloc per a 2 elements de tipus *string*.

- *message[0] = new String ("Hello")*: inicialitzem cada una de les posicions (en aquest cas l'element 0) amb una dada de tipus *string*. Aquí el que pot sobtar és que tornem a fer un *new String*, però recordeu que és perquè per a crear una cadena de text normal també ho havíem de fer d'aquesta manera. En el cas d'un enter o un decimal posaríem directament el valor, repasseu l'exemple que hi ha a l'apartat d' *arrays* per a veure-ho ben clar.

- Un cop tenim els dos texts carregats en l' *array*, un a la posició 0 i un altre a la posició 1, cal buscar la manera que es vagin alternant. En el nostre cas aprofitarem la variable *size*, que es va incrementant, i la funció mòdul (*%*). En fer el càlcul (*size % 2*) ens està retornant el valor 0 quan el valor *size* és parell o 1 quan és senar, i per tant ho podem aprofitar per a seleccionar quin element de l' *array* ensenyem a cada moment.

2.9. Exemple: calculant el temps

Una dada que ens pot ser molt útil és el temps, ja que pot ser vital per a fer una animació que sigui correcta o simplement per a tenir un control general del programa. Amb aquest exemple, a part de treballar amb variables i fer alguns càlculs, també entendrem millor la funció *draw* i el fet que es vagi executant constantment al llarg del temps.

Començarem calculant quant temps passa entre cada execució de la funció *draw*.

```
int time_now;
int time_old;
int time_delta;

void setup() {
   size (400, 150);
   frameRate (20);

   time_now = 0;
   time_old = 0;
   time_delta = 0;
}

void draw() {

   time_now = millis();
   time_delta = time_now - time_old;
   time_old = time_now;
```

```
    background(0, 0, 0);

    textSize (30);
    text (time_delta, 50, 35);
    text ("ms", 130, 35);
}
```

Analitzem el codi detalladament:

1) Declarem i inicialitzem 3 variables enteres que ens caldran per a fer el càlcul.

2) Inicialment fixem que el programa funcioni a 20 fps (plans per segon); això vol dir que cada segon redibuixarem la imatge 20 vegades. Per tant, el temps que passa entre cada pla que pintem és d'1/20 = 0,05 s, o el que és el mateix, 50 ms.

3) Dins de la funció *draw* hi ha el càlcul que realment ens interessa, però abans de veure'l comentarem la funció *millis()*. Aquesta funció retorna els mil·lisegons que han passat des que hem executat el programa, i la utilitzarem per a calcular el temps que passa entre dues execucions de la funció *draw*.

4) Ara sí, veurem com utilitzem aquestes variables i quins càlculs hi fem. Per a fer-ho simularem l'execució del programa pas per pas, i per tant veurem com es van modificant els valors cada vegada que la funció *draw* s'executa:

a) Inicialitzacions: aquí encara no s'ha executat la funció *draw* i inicialitzem les 3 variables a 0 (*time_now = 0*, *time_delta = 0*, *time_old = 0*).

b) *Draw* 1: primera execució de la funció, i primera actualització dels valors de les variables. Suposem que el temps que hem trigat per a inicialitzar-ho tot i arribar fins aquí ha estat de 10 ms, i per tant *time_now = 10*. Aquesta variable la utilitzarem per a indicar el temps total que fa que executem el programa des del principi de tot.

A continuació actualitzem el valor de *time_delta*, que és el temps actual menys el temps en què es va executar per última vegada la funció *draw*: és a dir, *time_old*. Per tant, el valor de *time_delta* és el que estem buscant, és el temps que ha passat entre dos plans del nostre programa.

Per acabar amb el cicle, fem *time_old* = *time_now*, i això ho fem perquè hem de deixar el valor de *time_old* actualitzat perquè quan es torni a executar la funció *draw* tingui el valor correcte.

c) *Draw* 2: la segona execució de la funció s'ha de fer 50 ms més tard (recordeu que hem definit 20 fps), per tant, *time_now* serà igual a 60 ms (els 10 ms que hem trigat a inicialitzar el programa + 50 ms entre els dos plans).

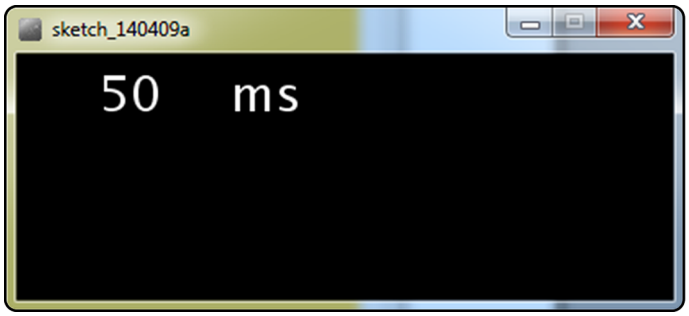
time_delta serà igual a 60 – 10 = 50 ms, és a dir, el temps actual menys el temps de l'última execució de la funció. Aquests 50 ms ja són els que volíem obtenir.

Per acabar, *time_old = 60 ms*, per preparar l'execució següent.

d) *Draw* n: a partir d'aquí anem repetint el mateix per a tenir el valor actualitzat en cada moment.

Tot i que sembla una manera complicada de fer el càlcul, és la que ens ofereix una de les millors solucions, un cop tenim clar el concepte de repetició.

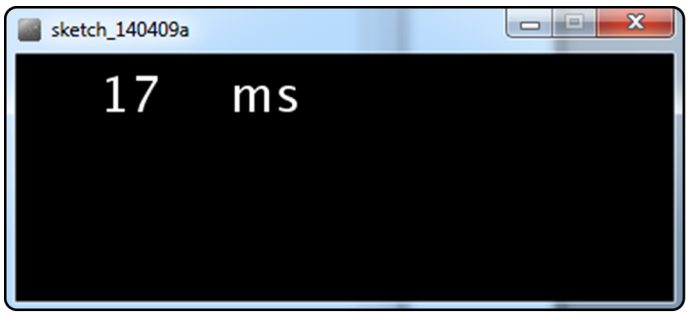
Figura 12. Calculant el temps entre plans



Ara que tenim el programa fet, comenteu la línia on definim els plans per segon que volem i torneu a executar-lo:

```
//frameRate (20);
```

Figura 13. Temps mínim entre plans



En eliminar la línia que controla els plans per segon, el programa intenta executar-lo al màxim de ràpid, de manera que el valor que obtenim ens diu el temps mínim entre plans que és capaç de generar el nostre ordinador.

A partir dels mil·lisegons que hem calculat podem obtenir els segons dividint entre 1.000, i ho fem amb una nova variable *sec*:

```
int time_now;
int time_old;
int time_delta;
float sec;

void setup() {
   size (400, 150);
   //frameRate (20);

   time_now = 0;
   time_old = 0;
   time_delta = 0;
   sec = 0;
}

void draw() {

   time_now = millis();
   time_delta = time_now - time_old;
   time_old = time_now;
   sec = time_delta / 1000.0;

   background(0, 0, 0);

   textSize (30);
   text (time_delta, 50, 35);
   text ("ms", 150, 35);
   text (sec, 50, 70);
   text ("segons", 150, 70);
}
```

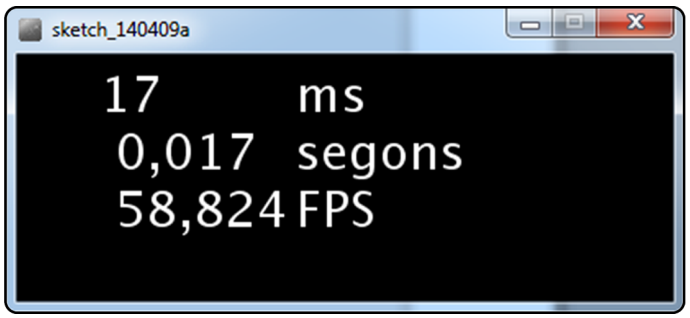
Fixeu-vos que en el codi hem dividit entre 1.000,**0**; recordeu el motiu? Proveu que passaria si dividíssim entre 1.000.

Figura 14. Temps en segons

Per acabar calcularem els plans per segon que estem obtenint. Per a fer-ho només haurem de fer 1/temps entre plans, però igual que abans, ho farem indicant 1,**0**/temps entre plans per no tenir problemes.

```
int time_now;
int time_old;
int time_delta;
float sec;
float fps;

void setup() {
   size (400, 150);
   //frameRate (20);

   time_now = 0;
   time_old = 0;
   time_delta = 0;
   sec = 0;
   fps = 0;
}

void draw() {

   time_now = millis();
   time_delta = time_now - time_old;
   time_old = time_now;
   sec = time_delta / 1000.0;
   fps = 1.0 / sec;

   background(0, 0, 0);

   textSize (30);
   text (time_delta, 50, 35);
   text ("ms", 170, 35);
   text (sec, 50, 70);
   text ("segons", 170, 70);
   text (fps, 50, 105);
   text ("FPS", 170, 105);
}
```

Figura 15. Plans per segon



I aquí tenim el resultat, no solament sabem quant temps triguem a executar la funció *draw*, sinó que a més podrem utilitzar aquesta informació per a calcular com va evolucionant el programa, les animacions, les transicions entre pantalles, etc.

3. Tipus de dades gràfiques

En l'apartat anterior hem vist les variables bàsiques que s'utilitzen de manera general en qualsevol tipus de programa, però ara és el moment de veure les eines que ens ofereix el Processing per a crear continguts gràfics, centrant-nos en les primitives i la gestió del color.

Per a fer-ho ens basarem en el codi següent, que simplement dibuixa una finestra de 500 × 400 píxels de color blanc:

```
void setup() {
   size(500, 400);
}

void draw() {
   background (255, 255, 255);
}
```

3.1. Formes bàsiques

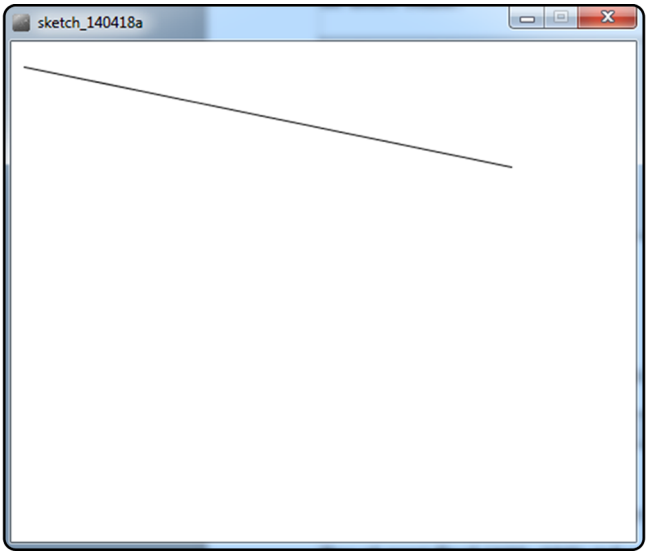
3.1.1. Línies

Afegim el codi següent dins de la funció *draw*:

```
line (10, 20, 400, 100);
```

En executar el codi podem veure que ens apareix una línia dibuixada des del punt inicial (10, 20) fins el punt final (400, 100), tal com hem indicat en els paràmetres que hem passat a la funció *line*.

Figura 16. Dibuixar línies



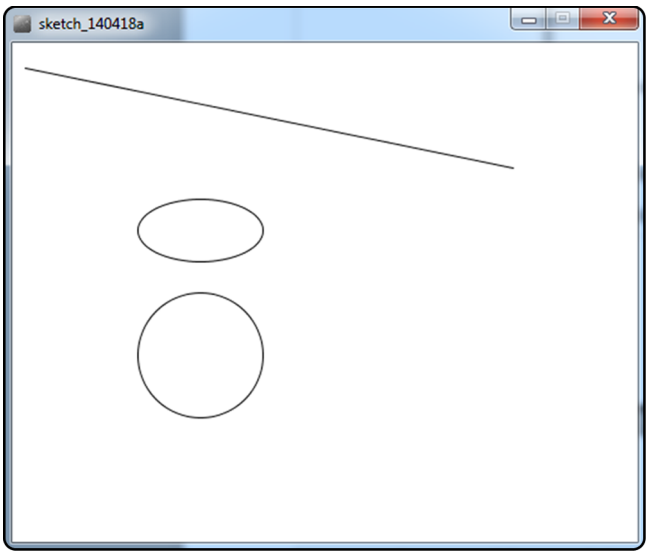
En aquest cas, i en general per a qualsevol element que dibuixem, el més important serà tenir clar la posició que haurà de tenir i com la controlarem, ja que el fet de crear una línia, afegir una imatge, etc., no serà cap problema, gràcies al fet que el Processing ens ho simplifica molt, i per tant la nostra preocupació final serà la de la composició de la pantalla que volem mostrar.

3.1.2. El·lipses

Afegim les línies al codi següents:

```
ellipse (150, 150, 100, 50);
ellipse (150, 250, 100, 100);
```

Figura 17. Dibuixar el·lipses



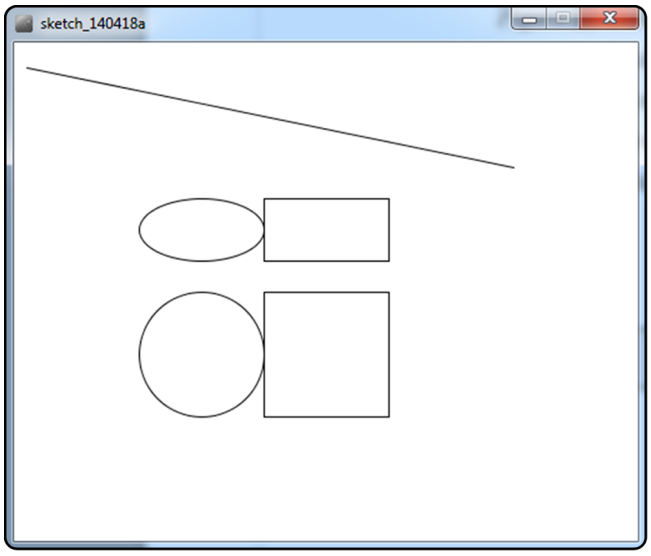
Igual que en el cas de la línia, la funció *ellipse* necessita que li indiquem 4 paràmetres: els dos primers fan referència a la posició del centre de l'el·lipse, el tercer és l'amplada i el quart l'alçada. Per tant, per a poder crear un cercle haurem de definir els dos últims paràmetres amb el mateix valor, tal com hem fet a la segona línia de codi que hem afegit (valor 100).

3.1.3. Rectangles

Afegim les línies al codi següents:

```
rect (200, 125, 100, 50);
rect (200, 200, 100, 100);
```

Figura 18. Dibuixar rectangles

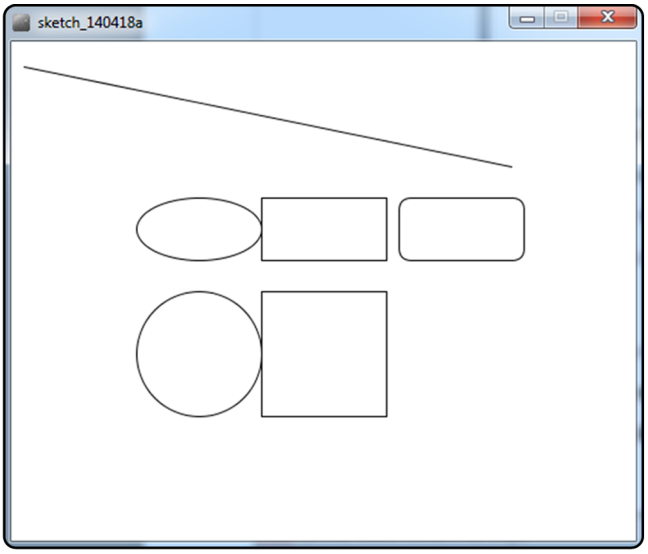


A diferència de les el·lipses, en el cas dels rectangles els dos primers paràmetres que indiquem indiquen la posició del vèrtex superior esquerra de la figura, i a continuació indiquem l'amplada i l'alçada que tindrà. A l'exemple podem veure com hem definit 2 rectangles que podrien contenir les el·lipses que havíem definit abans.

La funció *rect* té una particularitat que encara no havíem vist en cap altre cas, i és que li podem indicar un nombre diferent de paràmetres per aconseguir diferents resultats. En l'exemple anterior li hem donat quatre paràmetres, però observeu què passa si afegim la crida següent, amb cinc paràmetres:

```
rect (310, 125, 100, 50, 10);
```

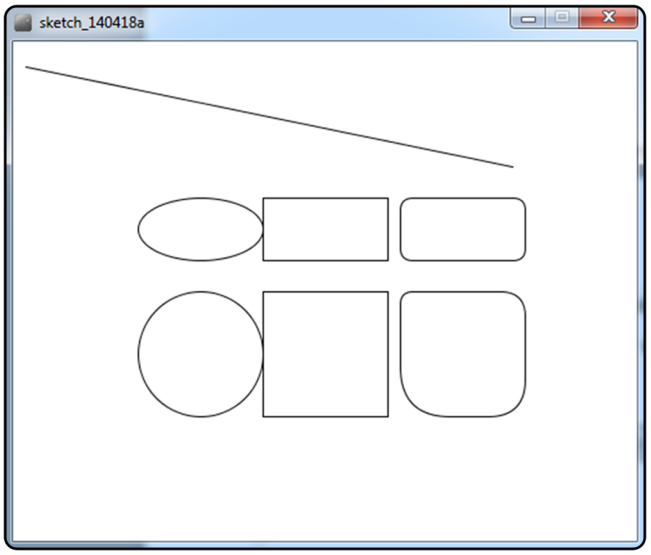
Figura 19. Dibuixar rectangles amb opcions (1)



Amb aquest paràmetre extra aconseguim definir el radi que tindran les quatre cantonades del rectangle, però si en comptes d'afegir un únic paràmetre n'afegim quatre més podrem modificar cada radi per separat:

```
rect (310, 200, 100, 100, 10, 20, 30, 40);
```

Figura 20. Dibuixar rectangles amb opcions (2)



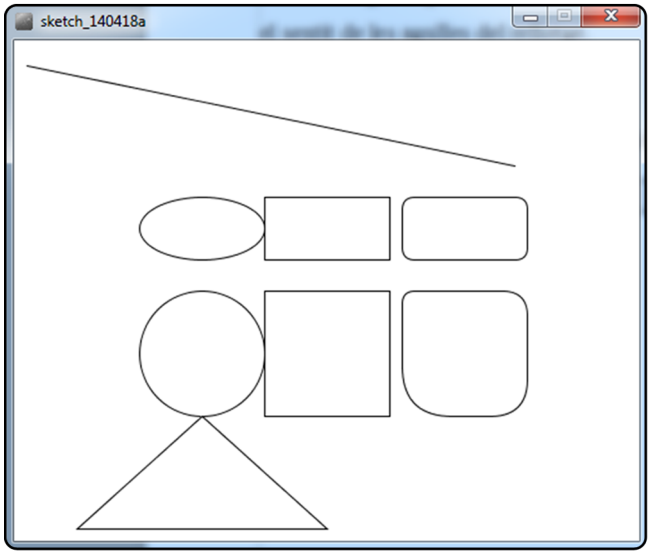
Fixeu-vos que el primer radi fa referència a la cantonada superior esquerra, i que la resta s'aplica en el sentit de les agulles del rellotge.

3.1.4. Triangles

Afegim el codi següent dins de la funció *draw*:

```
triangle (50, 390, 150, 300, 250, 390);
```

Figura 21. Dibuixar triangles



En el cas dels triangles sempre haurem d'indicar sis paràmetres, que definiran la posició dels tres vèrtexs de la figura. En el nostre cas són (50, 390), (150, 300) i (250, 390).

3.2. Gestió del color de les formes

Ara que podem dibuixar figures bàsiques veurem com podem indicar el color que volem que tinguin. Per a fer-ho ens basarem en aquest codi inicial:

```
color c_black;
color c_white;
color c_green;
color c_red;
color c_blue;

void setup() {
   size(500, 400);
   c_black = color (0, 0, 0);
   c_white = color (255, 255, 255);
   c_green = color (0, 255, 0);
   c_red = color (255, 0, 0);
   c_blue = color (0, 0, 255);
}

void draw() {
   background (c_black);
   rect (10, 10, 100, 100);
}
```

Per simplificar l'ús dels colors hem creat 5 variables, que hem inicialitzat amb els colors blanc, negre, vermell, verd i blau; d'aquesta manera no caldrà que introduïm els valors a mà cada vegada que els hàgim d'utilitzar. Podeu veure com en el cas del color de fons ja hem utilitzat la variable que indica el color negre en comptes dels valors (0, 0, 0).

També tenim un quadrat que per defecte es pintarà de color blanc.

3.2.1. Emplenament de les formes

En dibuixar una forma podem definir el color amb què pintarem l'interior o si, al contrari, volem que sigui transparent. Per a fer-ho utilitzarem les funcions *fill* i *noFill*, tal com podem veure en el codi següent, que afegim dins de la funció *draw*:

```
void draw() {
   background (c_black);
   fill (c_red);
   rect (10, 10, 100, 100);
}
```

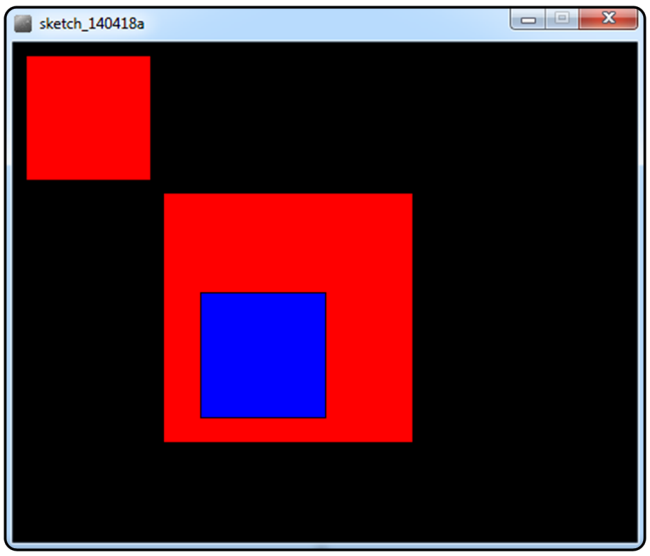
Figura 22. Funció *fill*: quadrat vermell

Ja ho tenim, ara el quadrat és de color vermell; n'afegirem algun més:

```
void draw() {
   background (c_black);
   fill (c_red);
   rect (10, 10, 100, 100);
   rect (120, 120, 200, 200);
   fill (c_blue);
   rect (150, 200, 100, 100);
```

```
}
```

Figura 23. Funció *fill*: ordre de pintar



Ja hem afegit un parell de figures; recordeu que l'últim requadre de color blau apareix a sobre del vermell perquè l'hem dibuixat després i el Processing sempre afegeix els nous elements per sobre dels que ja hi havia (l'algorisme del pintor).

Aquesta característica també és molt important de tenir en compte quan treballem amb funcions que modifiquen la manera de com es pinten els elements, com és el cas de la funció *fill*.

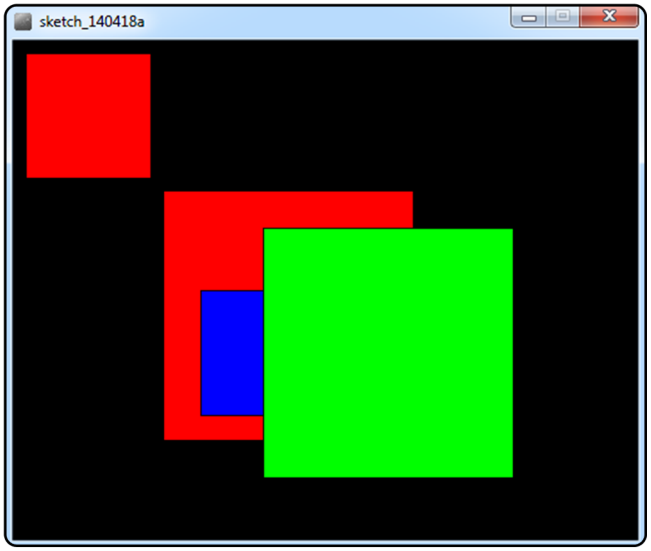
Des del moment que hem definit que volíem pintar les formes de color vermell, totes les noves formes que definim seran vermelles fins que tornem a canviar el color. En el nostre cas, abans de dibuixar l'últim requadre hem dit que començaríem a utilitzar el color blau.

És important que tinguem aquesta característica present quan utilitzem el Processing, ja que ens trobarem molts exemples en què serà necessari que ho tinguem clar per a aconseguir els resultats que volem.

Ara afegirem un altre requadre de color verd:

```
fill (c_green);
rect (200, 150, 200, 200);
```

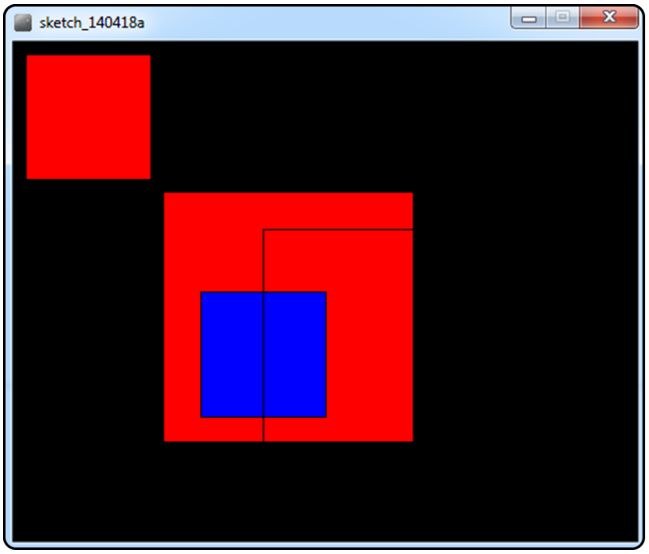
Figura 24. Funció *noFill*: figura opaca



Aquest quadrat que hem afegit està tapant als altres dos, però el que realment volem és poder-los veure, i per tant necessitem dir que en dibuixar aquest últim requadre no el pinti de cap color, que sigui transparent. Per a fer-ho, en comptes d'utilitzar la funció *fill* utilitzarem la funció *noFill*. Per tant, eliminem el codi que hem afegit i utilitzem el següent:

```
noFill ();
rect (200, 150, 200, 200);
```

Figura 25. Funció *noFill*: figura transparent



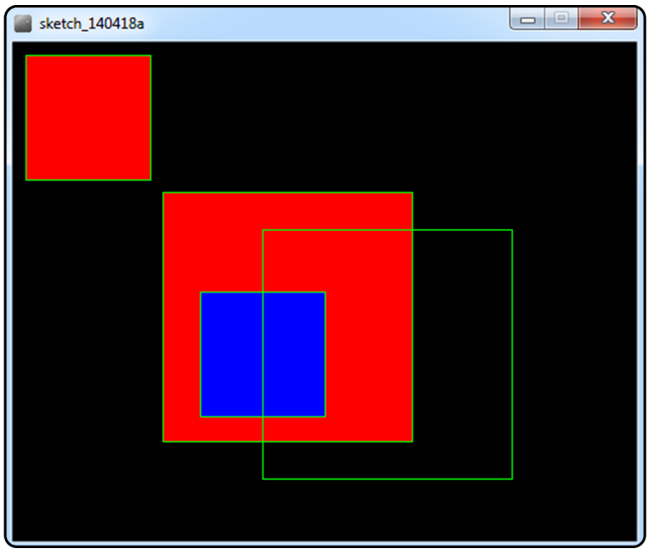
Ara ja tenim el quadrat transparent, i fixeu-vos que podem veure que té el contorn de color negre. En ser de color negre es confon amb el color de fons, però en l'apartat següent ho solucionarem.

3.2.2. Contorn de les formes

El primer que farem serà afegir aquest codi al principi de la funció *draw*:

```
stroke (c_green);
```

Figura 26. Funció *stroke*

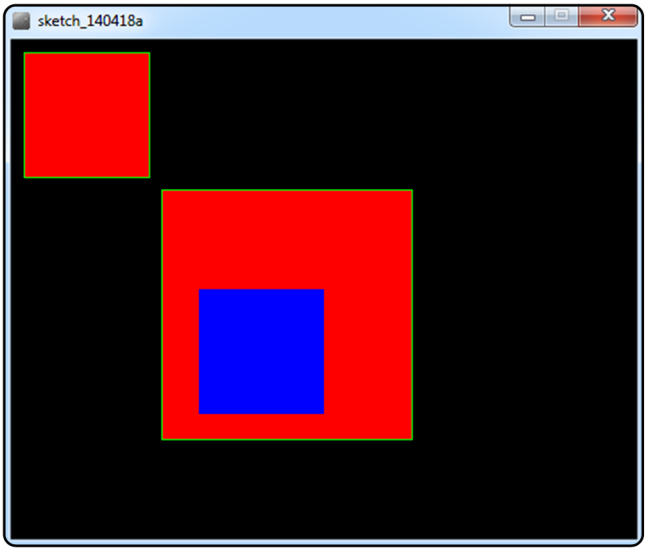


Aquesta funció defineix el color dels contorns de les figures que dibuixem, i en seleccionar el color verd, en comptes del negre que s'utilitza per defecte, podem veure clarament que tots els requadres estan contornejats.

Si, per exemple, volguéssim que el quadre de color blau no tingués cap contorn, just abans de dibuixar-lo utilitzaríem la funció *noStroke*:

```
fill (c_blue);
noStroke();
rect (150, 200, 100, 100);
```

Figura 27. Funció *noStroke*



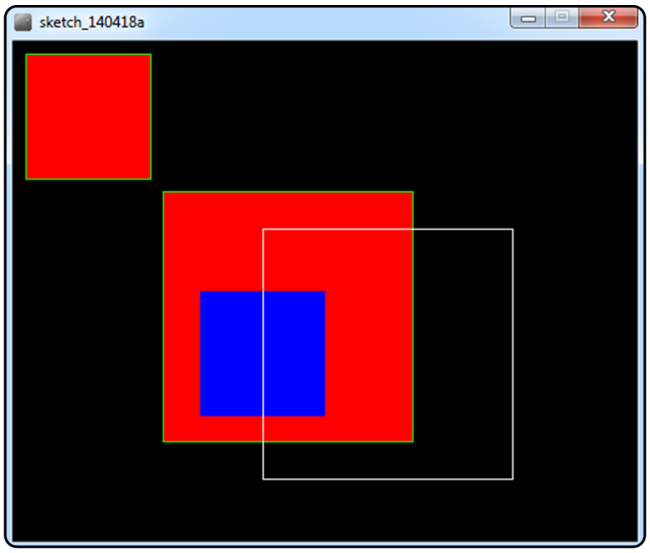
En afegir aquesta instrucció el quadre blau ja no té cap contorn, però on és el quadrat transparent que hi havia per sobre? Doncs és al mateix lloc, però com que era transparent i ara ja no pintem els contorns no el veiem. Per solucionar-ho caldrà indicar que volem pintar els contorns utilitzant la funció *stroke*, i en aquest cas ho farem de color blanc. En fer-ho el codi de la funció *draw* ens queda d'aquesta manera:

```
void draw() {
   background (c_black);
   stroke (c_green);
   fill (c_red);
   rect (10, 10, 100, 100);
   rect (120, 120, 200, 200);
   fill (c_blue);
   noStroke();
   rect (150, 200, 100, 100);
   noFill ();
   stroke (c_white);
   rect (200, 150, 200, 200);
}
```

- Indiquem que el color de fons és el negre.
- Definim que volem contornejar de color verd i pintar de color vermell.
- Dibuixem dos quadrats.
- Definim que volem pintar de color blau i que no volem contornejar.
- Dibuixem el quadrat petit.
- Definim que no volem pintar les figures i les volem contornejar de color blanc.
- Dibuixem el quadrat gran.

I el resultat és aquest:

Figura 28. Funcions stroke/noStroke/fill/noFill



Amb aquest exemple finalitzem la part de gestió del color de les formes. A part de les funcions que hem vist, cal tenir en compte la importància de l'ordre en què definim els elements i les seves característiques, ja que, com hem pogut veure, el resultat pot canviar radicalment. Això fa que sigui molt interessant planificar els diferents elements que apareixeran per pantalla abans de començar a escriure el codi per poder-lo estructurar correctament.

4. Animació bàsica

Com vam comentar al començar aquest curs de Processing, un dels objectius d'aquest llenguatge és el de permetre'ns crear animacions de manera fàcil. Per a fer-ho, el primer que ens cal és poder generar continguts, com per exemple les formes bàsiques que ja hem vist, i el pas següent és aprendre com les podem animar.

En parlar d'animacions és fàcil pensar en grans produccions de pel·lícules o videojocs, però cal recordar que qualsevol element animat, com per exemple una simple icona que canvia si hi cliquem a sobre, és una animació. Per tant, en aquest capítol ens iniciarem en tècniques d'animació bàsica que ens permetran seguir evolucionant per a obtenir resultats més complexos.

L'exemple que construirem serà el d'una bola en moviment; inicialment es tractarà d'una animació molt bàsica i hi anirem afegint alguns controls extres per donar una mica més de gràcia a l'animació final.

El codi base amb el qual començarem a treballar és aquest:

```
void setup() {
   size(800, 400);
   frameRate (20);
}

void draw() {
   background(255, 255, 255);
   fill (0, 0, 0);
}
```

Com podeu veure, hem definit una finestra que pintem de color blanc i que s'actualitzarà 20 vegades per segon.

Ara el que farem serà afegir la bola que volem animar. En definir-la fixeu-vos que utilitzarem variables per a indicar-ne la posició, ja que serà necessari que les puguem anar variant per a indicar-ne la posició actual.

```
int ball_x;
int ball_y;
int ball_radius;

void setup() {
   size(800, 400);
   frameRate (20);
```

```
    ball_x = 10;
    ball_y = 200;
    ball_radius = 10;
}

void draw() {
    background(255, 255, 255);
    fill (0, 0, 0);

    ellipse (ball_x, ball_y, ball_radius, ball_radius);
}
```

Figura 29. Animació: punt de partida

Ara ja podem veure la nostra bola situada en el punt des d'on s'iniciarà el moviment. El que farem a continuació serà modificar-ne la posició al llarg del temps perquè es vagi desplaçant. Per a fer-ho afegim aquesta línia de codi en la funció *draw()*:

```
ball_x = ball_x + 1;
```

Ja ho tenim! En executar el programa la bola es desplaçarà fins a l'altra punta de la finestra fins a desaparèixer. Podem dir que aquest és el programa equivalent a l'"hola món" en el cas de les animacions, i a partir d'aquí afegirem algunes modificacions per fer-lo una mica més interessant.

4.1. Estructura condicional ifelse

A continuació farem que la bola s'aturi al final de la finestra. Per a fer-ho haurem de comprovar la posició actual i actuar en conseqüència, i és per això que utilitzarem la instrucció *if...else*.

Modifiquem la funció *draw* perquè quedi d'aquesta manera:

```
void draw() {
   background(255, 255, 255);
   fill (0, 0, 0);

   ellipse (ball_x, ball_y, ball_radius, ball_radius);

   if (ball_x < 800){
      ball_x = ball_x + 1;
   }else{
      text ("STOP", 400, 200);
   }
}
```

Analitzem el que hem afegit. L'estructura *if...else* ens permet comprovar una condició i fer una sèrie de tasques si es compleix o unes altres si no es compleix. En el nostre cas:

- *if (ball_x < 800)*: comprovem si la posició de la bola en l'eix *x* és inferior a 800, que és l'amplada de la nostra finestra.

- Si la condició es compleix, la bola encara no ha arribat al final de la finestra, incrementem la variable *ball_x* perquè la bola continuï avançant.

- Si la condició no es compleix, la bola ha arribat al final, saltem al bloc de codi que hi ha en el *else*, i en aquest cas escrivim un text a la pantalla i ja no incrementem la variable de la posició de la bola.

4.2. Variables *height* i *width*

Ara que hem aconseguit que la bola es mogui fins al final de la finestra, canviarem la mida de la nostra finestra i passarem de 800×400 píxels a 400×400 píxels:

```
size(400, 400);
```

Si executem el codi la bola es començarà a desplaçar, però en arribar al final de la finestra no s'aturarà perquè nosaltres li havíem dit que no ho fes fins arribar a la posició 800.

Per a solucionar aquest tipus de problemes i poder-nos despreocupar de les dimensions que té la finestra a cada moment disposem de les variables *height* i *width*, que contenen la informació de l'alçada i amplada de la finestra del programa en cada moment. Modifiquem la funció *draw* d'aquesta manera:

```
void draw() {
   background(255, 255, 255);
```

```
    fill (0, 0, 0);

    text ("Height: "+height+", Width: "+width, 10, 15);
    ellipse (ball_x, ball_y, ball_radius, ball_radius);

    if (ball_x < width){
       ball_x = ball_x + 1;
    }else{
       text ("STOP", 400, 200);
    }
}
```

A part d'imprimir un text amb la informació de l'alçada i l'amplada, només hem canviat la condició de *if*:

```
if (ball_x < width)
```

Ara en comptes de comprovar si la posició és més petita que 800 ho comparem amb l'amplada que té la finestra en cada moment. Comproveu que si en torneu a canviar les dimensions, com per exemple 200×400 píxels, la bola es continua parant al final de la finestra correctament.

En general, intentarem aprofitar aquestes variables per a evitar que un canvi de les dimensions de la finestra ens obliguin a repassar tot el codi per a adaptar-lo a la mida nova.

4.3. Simulacions físiques

Amb l'exemple anterior hem aconseguit desplaçar una bola per la pantalla a una velocitat marcada pel nombre de plans per segon que estàvem dibuixant (20 fps). Si en comptes de 20 fps haguéssim indicat 10 o 30 fps la bola s'hauria mogut més lentament o ràpidament, ja que l'increment de la posició només depèn dels cops que es cridi la funció *draw*.

```
ball_x = ball_x + 1;
```

Això pot ser un problema si el que volem fer és una simulació física més realista, en què el moviment dels objectes no es basi en la velocitat de repintada del nostre programa sinó del temps.

Reescriurem el codi perquè la bola caigui en comptes de moure's horitzontalment:

```
int ball_x;
int ball_y;
int ball_radius;
```

```
void setup() {
   size(400, 400);
   frameRate (20);

   ball_x = 200;
   ball_y = 10;
   ball_radius = 10;
}

void draw() {
   background(255, 255, 255);
   fill (0, 0, 0);

   ellipse (ball_x, ball_y, ball_radius, ball_radius);

   if (ball_y < height){
      ball_y = ball_y + 10;
   }else{
      text ("STOP", width/2, height/2);
   }
}
```

En executar el programa veureu que la bola cau ràpidament fins a aturar-se, i el que farem ara serà fer que el moviment sigui real, una simulació física de com una bola de veritat cauria si la deixéssim anar. Per a fer-ho necessitarem dues coses:

1) Una referència de temps real, ja que la velocitat de repintada que definim no ens garanteix que es compleixi aquest temps. Per a calcular el temps utilitzarem el mateix sistema que ja hem vist anteriorment.

Vegeu també

El sistema per a calcular el temps es tracta en el subapartat 2.9, sobre el tipus de dades bàsiques.

2) Una equació que ens defineixi el moviment que volem fer. Aquesta pot ser la part més complicada en funció del que vulguem fer: en aquest cas és tracta de la funció de desplaçament d'un cos accelerat. La recordeu, de les assignatures de física?

$$x = x^0 + v_0 t + \frac{1}{2} a t^2$$

En el nostre cas, aquesta serà la més complicada que veurem, i ens servirà per a calcular la posició de la bola cada vegada que l'hàgim de repintar. Per a emplenar aquesta equació també utilitzarem la del càlcul de la velocitat, que veurem directament en el codi.

Ara veurem les variables que utilitzarem:

```
// Time variables
```

```
int time_now;
int time_old;
int time_delta;

// Ball variables
int ball_x;
int ball_radius;
float ball_y;
float ball_y_old;
float ball_y_speed;
float ball_y_speed_old;
float ball_y_acceleration;
```

Per a calcular el temps definim les tres variables que ja havíem vist: el temps actual, el temps de l'execució anterior de la funció *draw*, i el temps que ha passat entre les dues execucions.

Per a la bola mantenim les variables del radi i la posició *x*, però modifiquem la variable de la posició *y* i diem que sigui de tipus *float*. Això ho fem per assegurar-nos que en fer els càlculs tinguem més precisió i la bola es mogui d'una manera més suau. També hem definit quatre variables més que necessitarem per a fer el càlcul: la posició *y* de l'última vegada que s'ha mogut la bola, la velocitat actual i la velocitat anterior i l'acceleració.

Les inicialitzacions queden d'aquesta manera:

```
void setup() {
   size(400, 400);
   frameRate (20);

   //Time init
   time_now = 0;
   time_old = 0;
   time_delta = 0;

   //Ball init
   ball_x = 200;
   ball_y = 10.0;
   ball_y_old = 10.0;
   ball_radius = 10;
   ball_y_speed = 0.0;
   ball_y_speed_old = 0.0;
   ball_y_acceleration = 9.8;
}
```

Aquí no hi ha cap sorpresa, inicialitzem els temps a zero, i la posició de la bola en el punt d'on volem que surti, amb la velocitat inicial a zero i l'acceleració de la Terra.

Ara ens toca fer el càlcul del temps: utilitzarem el mateix codi que ja havíem vist i comentat, inserint-lo al principi de la funció *draw*:

```
time_now = millis();
time_delta = time_now - time_old;
time_old = time_now;
```

La variable *time_delta* és la que utilitzarem per a fer el càlcul de la nova posició, i ho fem d'aquesta manera:

```
    if (ball_y < height){
       ball_y_speed = ball_y_speed_old + (ball_y_acceleration * (time_delta/1000.0));
       ball_y = ball_y_old + (ball_y_speed_old * (time_delta/1000.0)) +
       ( (ball_y_acceleration * (time_delta/1000.0) * (time_delta/1000.0)) / 2.0 );

       ellipse (ball_x, ball_y, ball_radius, ball_radius);

       ball_y_speed_old = ball_y_speed;
       ball_y_old = ball_y;
    }else{
       text ("STOP", width/2, height/2);
    }
```

Mentre la bola no arribi al terra:

- Calculem la velocitat que ha de tenir en funció de la velocitat anterior i el temps que ha passat. Fixeu-vos que estem dividint la variable *time_delta* entre 1.000,0, ja que aquest valor és en mil·lisegons i nosaltres el necessitem expressat en segons.

- Calculem la nova posició utilitzant l'equació que hem vist.

- Dibuixem la bola en la posició nova que acabem de calcular.

- Actualitzem el valor de les variables *antigues* amb les que hem calculat ara, d'aquesta manera ja estarem preparats per a fer el proper càlcul.

I en el cas que arribi a baix, fem que s'aturi i ensenyem el missatge a la pantalla.

A partir d'aquí seria força senzill modificar aquest codi per a simular el típic exemple d'una bola disparada per un canó. Us animeu a provar-ho?

4.4. Taula de billar

Per acabar, farem l'exemple de la bola que va rebotant per la pantalla.

Vegeu també

L'estructura condicional *if...else* es tracta en el subapartat 4.1. d'aquest mòdul.

El codi està basat en el que hem vist en parlar de l'estructura *if...else*:

```
int ball_x;
int ball_y;
int ball_radius;

void setup() {
   size(800, 400);
   frameRate (20);

   ball_x = 10;
   ball_y = 200;
   ball_radius = 10;
}

void draw() {
   background(255, 255, 255);
   fill (0, 0, 0);

   ellipse (ball_x, ball_y, ball_radius, ball_radius);

   if (ball_x < 800){
      ball_x = ball_x + 1;
   }else{
      text ("STOP", 400, 200);
   }
}
```

Abans de veure el codi de la solució intenteu modificar-lo vosaltres mateixos per a aconseguir que la bola vagi rebotant per les parets contínuament.

Una possible solució podria ser aquesta:

```
int ball_x;
int ball_y;
int ball_radius;
int inc_x;
int inc_y;

void setup() {
   size(800, 400);
   frameRate (20);
```

```
    ball_x = 10;
    ball_y = 200;
    ball_radius = 10;

    inc_x = 5;
    inc_y = 2;
}

void draw() {
    background(255, 255, 255);
    fill (0, 0, 0);

    ball_x = ball_x + inc_x;
    ball_y = ball_y + inc_y;

    ellipse (ball_x, ball_y, ball_radius, ball_radius);

    if ( (ball_x <= 0) || (ball_x >= width)) {
        inc_x = inc_x * -1;
    }
    if ( (ball_y <= 0) || (ball_y >= height)) {
        inc_y = inc_y * -1;
    }
}
```

Per a fer el control hem definit dues variables que ens diuen quant ens hem de desplaçar cap a cada direcció. En el cas que la bola sobresurti per algun costat, per exemple per la dreta de la finestra, multipliquem per –1 aquest desplaçament i per tant fem que la bola torni enrere.

5. Interacció bàsica: capturar el ratolí

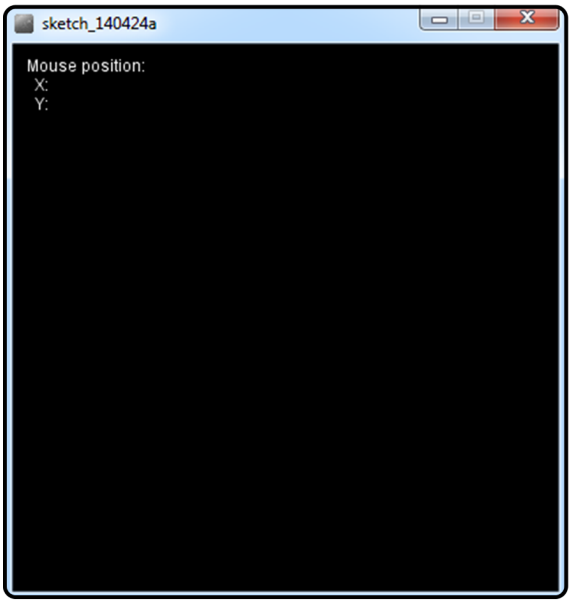
Amb els exemples que hem vist fins ara sempre hem estat creant programes que en executar-los feien directament la tasca definida i finalitzaven, però hi haurà molts casos en què necessitarem que l'usuari interactuï per donar indicacions sobre què vol fer.

Per a donar solució a aquesta necessitat veurem quines opcions ens dóna el Processing per a gestionar el ratolí, i farem un seguit d'exercicis per a comprovar-ne el funcionament.

El codi base que utilitzarem és aquest:

```
void setup() {
   size(400, 400);
   frameRate (20);
}

void draw() {
   background(0, 0, 0);
   text ("Mouse position:\n  X: "+"\n  Y:",10, 20);
}
```

Figura 30. Interacció amb el ratolí: codi inicial



De moment només definim la finestra i preparem el text en què informarem de la situació del ratolí.

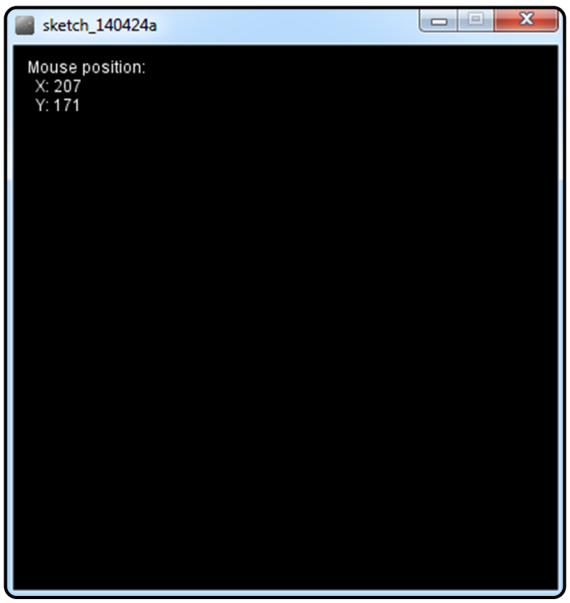
5.1. Coneixem la posició actual

Aquest és un altre cas en què el Processing ens demostra que està pensat per a facilitar-nos la feina, ja que posa a la nostra disposició un parell de variables que s'actualitzen automàticament per indicar-nos la posició del ratolí en tot moment.

Aquestes variables són *mouseX* i *mouseY*, i les podem consultar directament des de qualsevol lloc del nostre codi. Per exemple, si canviem la línia de text que estàvem pintant i la deixem d'aquesta manera:

```
text ("Mouse position:\n  X: "+mouseX+"\n  Y: "+mouseY,10, 20);
```

Figura 31. Interacció amb el ratolí: mouseX/mouseY



D'una manera molt fàcil estem ensenyant en tot moment les coordenades on ens trobem, i per tant podrem utilitzar aquesta informació per a interactuar amb el programa.

5.2. Perseguim el ratolí

Ara que sabem com podem consultar la posició farem que una bola el segueixi indefinidament. Per tant, el primer que caldrà fer és definir les variables que ens caldran per a dibuixar-la:

```
float ball_x;
float ball_y;
float dir_x;
float dir_y;
```

I a continuació inicialitzar-les:

```
ball_x = width/2.0;
ball_y = height/2.0;
dir_x = 1.0;
dir_y = 1.0;
```

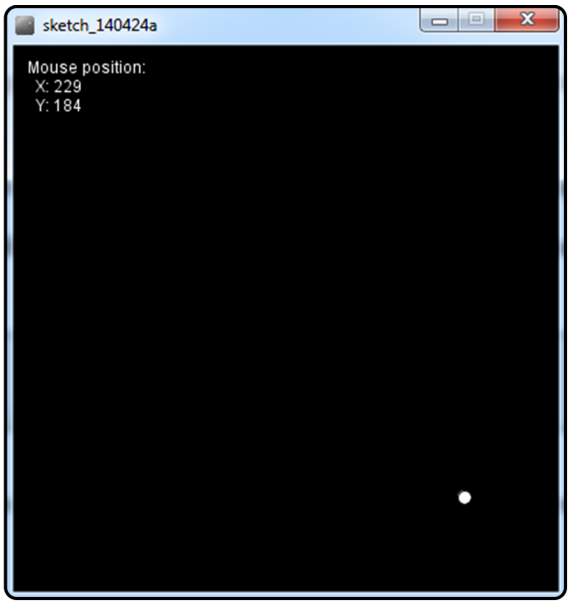
Podem veure que tenim dues parelles de variables:

- *ball_**: ens serveix per a indicar la posició actual de la bola i està inicialitzada en el centre de la finestra.
- *dir_**: indica cap a on es mourà la bola; de moment diem que es mogui cap avall a la dreta.

L'únic que ens falta és dibuixar la bola i calcular-ne el desplaçament. De moment ho fem sense tenir en compte el ratolí:

```
ball_x = ball_x + dir_x;
ball_y = ball_y + dir_y;
ellipse (ball_x, ball_y, 10, 10);
```

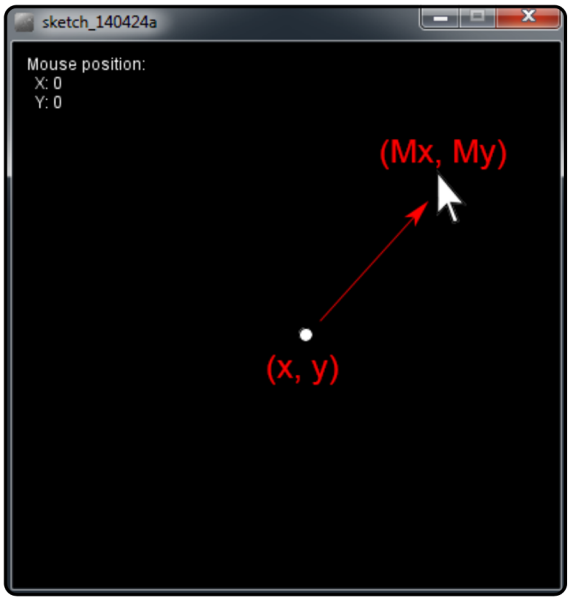
Figura 32. Perseguim el ratolí



I ja ho tenim, una bola que es mou des del centre fins a la cantonada inferior dreta.

Ara el que ens cal calcular és el valor de les variables *dir_x* i *dir_y* perquè en cada moment la bola es mogui cap on hi ha el ratolí.

Figura 33. Càlcul del desplaçament



Com podem veure a la imatge, coneixem la posició actual de la bola i la del ratolí, de manera que hem de calcular la diferència que hi ha entre les dues posicions ($Mx - x$, $My - y$). Traduït al codi seria així:

```
void draw() {
   background(0, 0, 0);
   text ("Mouse position:\n  X: "+mouseX+"\n  Y: "+mouseY,10, 20);

   dir_x = mouseX - ball_x;
   dir_y = mouseY - ball_y;

   ball_x = ball_x + dir_x;
   ball_y = ball_y + dir_y;
   ellipse (ball_x, ball_y, 10, 10);
}
```

Hem afegit les dues línies que recalculen el valor de *dir_x* i *dir_y*, de manera que, en comptes d'incrementar sempre la posició de la bola amb un valor igual, utilitza el ratolí com a referència.

5.3. Capturar els clics del ratolí

A part de poder conèixer la posició del ratolí, el Processing també ens dóna les eines per a poder conèixer l'estat de la resta d'elements del ratolí, com per exemple si hem clicat un botó, si l'hem desplaçat, etc.

De totes les funcions i variables que podeu trobar a la referència n'utilitzarem un parell per a detectar quan fem clic, i fer que la bola es mogui o quedi parada. Per a fer-ho ens basarem en aquest codi:

```
float ball_x;
float ball_y;
float dir_x;
float dir_y;
boolean run;

void setup() {
   size(400, 400);
   frameRate (20);

   ball_x = width/2.0;
   ball_y = height/2.0;
   dir_x = 1.0;
   dir_y = 1.0;
   run = false;
}

void draw() {
   background(0, 0, 0);
   text ("Mouse position:\n  X: "+mouseX+"\n  Y: "+mouseY,10, 20);

   if (run == true){
      dir_x = mouseX - ball_x;
      dir_y = mouseY - ball_y;

      ball_x = ball_x + dir_x;
      ball_y = ball_y + dir_y;
   }
   ellipse (ball_x, ball_y, 10, 10);
}
```

Com podeu veure, és el mateix codi de la bola que seguia al ratolí, però hem fet un parell de canvis:

- Hem afegit la variable *run*, que utilitzarem per a indicar si la bola s'ha de moure o no. De moment l'hem deixada inicialitzada amb el valor *false*, de manera que encara que moguem el ratolí, la bola no es mourà.

- Hem afegit un bloc *if...else* que comprova el valor de la variable *run*. En el cas que el valor sigui *true* calculem el moviment que ha de fer la bola de la mateixa manera que havíem vist en l'exemple anterior.

Si executeu el codi tal com està, la bola es quedarà quieta al centre de la finestra. Si proveu a inicialitzar la variable *run* a *cert*, veureu com sí que es mourà.

Però el que nosaltres volem és que el valor d'aquesta variable canviï en fer clic amb el ratolí. Per a fer-ho haurem d'afegir una altra funció en el nostre codi:

```
void mousePressed() {
   run = true;
}
```

Per tant, ara tindrem la funció *setup*, la funció *draw*, i després d'aquesta hem afegit la funció *mousePressed*. Aquesta funció és la que el Processing executarà sempre que premem algun botó del ratolí, i en el nostre cas només la utilitzem per a fer que la variable *run* tingui el valor *cert*.

Amb aquest canvi, quan executem el programa veurem que la bola està quieta al centre de la finestra encara que moguem el ratolí, però en el moment que fem clic ens començarà a perseguir.

Fem-ho una mica més interessant: per a aconseguir-ho utilitzarem la variable *mouseButton*. Aquesta variable també l'actualitza automàticament el Processing i ens informa de l'últim botó del ratolí que hem clicat. Per tant, el que podem fer és modificar la funció *mousePressed* i fer que en clicar amb el botó esquerre la bola es mogui i que amb el dret s'aturi. El canvi que hem de fer és molt senzill:

```
void mousePressed() {
   if (mouseButton == LEFT){
      run = true;
   }else{
      if (mouseButton == RIGHT){
         run = false;
      }
   }
}
```

Simplement cal identificar el botó que hem clicat i actualitzar la variable *run* en funció del que vulguem que passi.

5.4. Utilització de la roda del ratolí

Finalitzarem l'apartat de gestió del ratolí veient com podem utilitzar la informació del desplaçament de la roda. Per a fer-ho ampliarem una mica més l'exemple i farem que puguem ajustar la velocitat del moviment de la bola amb la roda del ratolí.

En aquest cas haurem d'afegir una altra funció en el nostre codi. Després de la funció *mousePressed* afegiu aquest codi:

```
void mouseWheel(MouseEvent event) {
```

```
    float incr = event.getAmount();
    println(incr);
}
```

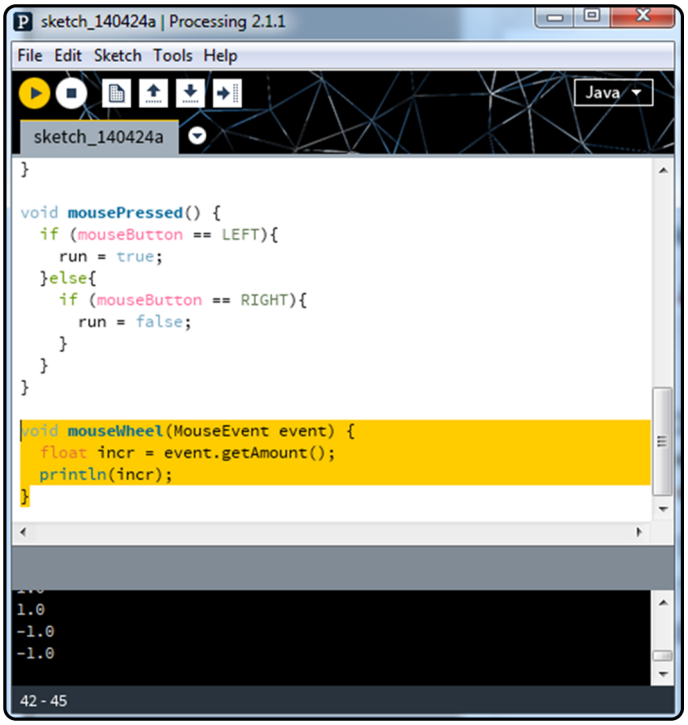
La funció *mouseWheel* s'executa sempre que fem girar la roda del ratolí, i en aquest cas per a recuperar la informació de cap on hem girat la roda tenim la variable *event*; més endavant veurem amb més detall com funcionen les funcions, de moment ens interessa saber que la variable *incr* valdrà 1 o –1 en funció del sentit en què girem la roda del ratolí.

Vegeu també

Les funcions es tracten en l'apartat 6 d'aquest mòdul.

Si executeu el codi i feu girar la roda podreu veure com apareix el valor a la finestra de missatges de l'editor, gràcies al *println* que fem:

Figura 34. Roda del ratolí



El que farem a continuació és aprofitar aquesta informació per a modificar una variable que ens indicarà la velocitat de la bola i que podrà tenir un valor entre 1 i 100:

- El valor 1 indicarà la velocitat més lenta.
- El valor 100 indicarà la més ràpida, que serà la velocitat amb què s'està movent ara la bola.

Abans de modificar realment la velocitat de la bola, afegirem aquesta variable i el codi suficient per a poder comprovar que n'estem modificant el valor.

Afegim la variable:

```
float vel;
```

I la inicialitzem a la màxima velocitat:

```
vel = 100.0;
```

A més, modifiquem el missatge de text perquè també indiqui la velocitat actual:

```
text ("Mouse position:\n  X: "+mouseX+"\n  Y: "+mouseY+"\n  Vel: "+vel,10, 20);
```

I ara ens cal modificar la funció *mouseWheel* per a actualitzar el valor de la velocitat:

```
void mouseWheel(MouseEvent event) {
   float incr = event.getAmount();

   vel = vel + incr;
   if (vel < 1.0){
      vel = 1.0;
   }
   if (vel > 100.0){
      vel = 100.0;
   }
}
```

Ara en executar el codi podreu veure com el valor de la velocitat va canviant, i mai no és inferior a 1 ni superior a 100, que són els límits que volíem tenir.

Per finalitzar caldrà modificar el càlcul del moviment de la bola per a adaptar-lo a la velocitat:

```
if (run == true){
      dir_x = mouseX - ball_x;
      dir_y = mouseY - ball_y;

      dir_x = (dir_x * vel) / 100.0;
      dir_y = (dir_y * vel) / 100.0;

      ball_x = ball_x + dir_x;
      ball_y = ball_y + dir_y;
}
```

Hem afegit les dues línies centrals. El que estem fent és agafar el desplaçament que faria la bola i calcular el percentatge en funció de la velocitat que tenim seleccionada. D'aquesta manera si la velocitat és 10, només ens desplaçarem el 10% de la distància que hauríem de recórrer, i per tant la bola es mourà més lentament que si anem al 50% o al 100%.

Al final hem obtingut un programa en què afegint dues funcions extres podem controlar amb el ratolí si la bola es mou o no, cap a on es mou i a quina velocitat d'una manera força senzilla gràcies a tota la informació que podem obtenir amb les ajudes que ens ofereix Processing.

6. Organització del codi

Per a ajudar a organitzar el codi d'un programa i fer-lo més fàcil d'entendre i gestionar utilitzarem dues eines diferents: les funcions i els *frames*.

6.1. Les funcions

Una funció és un conjunt d'instruccions que utilitzem per a fer unes accions concretes. Fins ara ja hem vist i utilitzat algun exemple de funcions: *setup*, *draw*, *mouseWheel*, etc.

6.1.1. Estructura bàsica de les funcions

L'estructura bàsica d'una funció és la següent:

```
<tipus_retorn> <nom_funció> (<parametres_que_rep>){
   <codi_de_la_funció>
   return <variable_que_retornem>
}
```

Anem per parts:

1) Tipus de retorn: quan executem una funció i es realitzen les accions que tingui definides pot ser que al final ens hagi de tornar algun resultat. Per exemple, si tinguéssim una funció que calculés el complementari d'un color voldríem que en finalitzar la funció ens tornés aquest color per a poder-lo fer servir. En aquests casos cal indicar-ho perquè tothom ho sàpiga i el programa funcioni correctament.

De moment totes les funcions que hem vist tenien com a tipus de retorn *void*, i això indica que aquesta funció no retorna cap valor:

```
void setup(){}
```

Però si, per exemple, hagués de retornar un nombre enter, hauríem d'utilitzar *int* per a indicar-ho:

```
int funcioTornamUnNombre(){}
```

2) Nom de la funció: és el nom que utilitzarem quan vulguem executar el codi que conté. Per convenció general el nom comença per una lletra minúscula i sense espais en blanc. Si el nom de la funció està compost per diverses paraules farem que la inicial de cada una sigui en majúscules excepte la primera; per exemple: *aixoEsElNomDunaFuncio*.

És important que els noms siguin autodescriptius per facilitar l'enteniment del codi.

3) Paràmetres que rep la funció: en executar una funció li podem passar tants paràmetres o variables com siguin necessaris perquè la funció pugui fer les tasques que té definides. Seguint amb l'exemple de la funció que calcula el complementari d'un color, cal que li passem el color original perquè pugui fer el càlcul, no?

Per a indicar els paràmetres que rep farem servir la mateixa notació que utilitzem per a definir les variables del nostre programa, amb la diferència que, si n'hi ha més d'una, utilitzarem comes per a separar-les:

```
void nomFuncio (int nombre, String text){}
```

En aquest cas la funció rep dos paràmetres, un de tipus enter i una cadena de text, que podrem utilitzar dintre de la funció amb els noms que hem definit: *nombre* i *text*, respectivament.

4) Codi de la funció: és el conjunt d'instruccions que realitzem, i treballarem de la mateixa manera que ho hem estat fent fins ara amb les funcions *setup*, *draw*, etc.

5) Variable que retornem: en el cas que la funció retorni algun paràmetre l'haurem d'indicar en finalitzar la funció utilitzant l'estructura:

```
return variable
```

variable és el nom de la variable que té el valor que volem tornar. En aquest cas és obligatori que el tipus de variable sigui el mateix que el tipus que hem dit que retornava la funció. És a dir, si hem definit la funció d'aquesta manera:

```
int tornaEnter(){}
```

la variable que retornem haurà de ser de tipus *int*, ja que si no el programa ens donarà un error.

6.1.2. Definim la primera funció

A partir del codi de la bola que es mou que havíem creat en el capítol anterior, crearem una funció nova per a fer el càlcul de la posició de la bola fora de la funció *draw*.

El codi inicial és aquest:

```
float ball_x;
float ball_y;
float dir_x;
float dir_y;
boolean run;
float vel;

void setup() {
   size(400, 400);
   frameRate (20);

   ball_x = width/2.0;
   ball_y = height/2.0;
   dir_x = 1.0;
   dir_y = 1.0;
   run = false;
   vel = 100.0;
}

void draw() {
   background(0, 0, 0);
   text ("Mouse position:\n  X: "+mouseX+"\n  Y: "+mouseY+"\n  Vel: "+vel,10, 20);

   if (run == true){
      dir_x = mouseX - ball_x;
      dir_y = mouseY - ball_y;

      dir_x = (dir_x * vel) / 100.0;
      dir_y = (dir_y * vel) / 100.0;

      ball_x = ball_x + dir_x;
      ball_y = ball_y + dir_y;
   }
   ellipse (ball_x, ball_y, 10, 10);
}

void mousePressed() {
   if (mouseButton == LEFT){
      run = true;
```

```
    }else{
       if (mouseButton == RIGHT){
          run = false;
       }
    }
}

void mouseWheel(MouseEvent event) {
    float incr = event.getAmount();

    vel = vel + incr;
    if (vel < 1.0){
       vel = 1.0;
    }
    if (vel > 100.0){
       vel = 100.0;
    }
}
```

Podem veure que en la funció *draw* tenim un bloc de codi que calcula la posició de la bola:

```
    if (run == true){
       dir_x = mouseX - ball_x;
       dir_y = mouseY - ball_y;

       dir_x = (dir_x * vel) / 100.0;
       dir_y = (dir_y * vel) / 100.0;

       ball_x = ball_x + dir_x;
       ball_y = ball_y + dir_y;
    }
```

El que farem serà definir una nova funció que farà aquesta feina:

```
void calculatePosition() {
    dir_x = mouseX - ball_x;
    dir_y = mouseY - ball_y;

    dir_x = (dir_x * vel) / 100.0;
    dir_y = (dir_y * vel) / 100.0;

    ball_x = ball_x + dir_x;
    ball_y = ball_y + dir_y;
}
```

Fixeu-vos que aquesta funció no retorna cap valor: simplement es dedica a consultar i modificar les variables que havíem definit des del principi, i és el mateix codi que teníem dintre de la funció *draw*. Ara ens falta executar aquesta nova funció des d'on abans teníem aquest codi:

```
void draw() {
   background(0, 0, 0);
   text ("Mouse position:\n  X: "+mouseX+"\n  Y: "+mouseY+"\n  Vel: "+vel,10, 20);

   if (run == true){
      calculatePosition();
   }
   ellipse (ball_x, ball_y, 10, 10);
}
```

Com podem veure, dintre de la funció *draw* on abans hi havia el codi per a calcular la posició ara només estem cridant la nova funció que hem definit. Això fa que la funció *draw* sigui més fàcil d'entendre, ja que en comptes d'uns càlculs més o menys complexos tenim una funció que amb el seu nom ja ens diu què fa.

Un altre avantatge és que podem utilitzar aquesta mateixa funció des de llocs diferents sense haver de copiar tot el codi, i per tant, si en algun moment fem un canvi s'aplicarà automàticament a tots els llocs des d'on utilitzem la funció.

6.1.3. Altres exemples

Ara veurem algun exemple més, i per a fer-ho ens basarem en aquest codi:

```
int i;

void setup() {
   size(200, 100);
   frameRate (1);

   i = 1;
}

void draw() {
   background(0, 0, 0);
   text ("i value: "+i,10, 20);

   i = i +1;
}
```

Com podeu veure, és un programa molt senzill que es dedica a incrementar el valor d'una variable. A partir d'aquí veurem diferents maneres de modificar-lo per a treballar amb funcions que rebin paràmetres o no, i que retornin paràmetres o no.

Funció *plusOne*

Comencem pel cas més fàcil: crearem una funció que no rebrà cap paràmetre ni retornarà res, i que s'encarregarà d'incrementar la variable *i*:

```
void draw() {
   background(0, 0, 0);
   text ("i value: "+i,10, 20);

   plusOne();
}

void plusOne(){
   i = i + 1;
}
```

Definim la funció indicant que no retorna res amb el *void* davant del nom de la funció, i que no rep cap paràmetre, ja que no n'hi ha cap de definit entre els parèntesis. A dins de la funció actualitzem directament el valor de la variable *i*.

L'últim que hem de fer és cridar la funció des de la funció *draw* perquè s'executi i incrementi el valor.

Funció *plusX*

En aquest cas volem crear una funció que ens permeti incrementar el valor de la variable amb el nombre que vulguem en cada moment; per tant, ens interessarà passar aquest nombre com a paràmetre. Ho fem de la manera següent:

```
void draw() {
   background(0, 0, 0);
   text ("i value: "+i,10, 20);

   plusX(10);
}

void plusX (int x){
   i = i + x;
}
```

Fixeu-vos com hem definit el paràmetre que rep la funció. En ser un enter indiquem el tipus *int* i a continuació el nom de la variable *x*. I serà aquesta variable la que sumarem per a incrementar el valor de *i*.

Per a cridar la funció l'únic que hem de fer és dir quant volem incrementar el valor, i ja ho tenim solucionat; en el nostre cas s'anirà incrementant de 10 en 10.

Funció *addXY*

L'últim exemple serà el d'una funció que rep dos nombres i retorna la suma de tots dos; el codi resultant queda així:

```
void draw() {
   background(0, 0, 0);
   text ("i value: "+i,10, 20);

   i = addXY (i, 10);
}

int addXY (int x, int y){
   int sum;
   sum = x + y;
   return sum;
}
```

En aquest cas, a part de rebre dues variables, hem definit que la funció retorni un enter (en comptes de *void* indiquem *int*), i al final de la funció utilitzem el paràmetre *return* per a indicar que volem tornar el valor de la variable *sum*.

La crida de la funció també varia una mica; en aquest cas, com que ens retorna un valor cal que l'assignem a alguna variable (la *i*), i li passem els dos paràmetres que volem sumar (el valor actual de la *i* i l'increment que volem aplicar).

6.2. Els *frames* i la gestió del flux del programa

En començar a definir un programa el primer que cal respondre és què volem que faci, independentment de com ho acabem programant. Aquesta separació entre idea i implementació a vegades és molt difícil de fer, sobretot perquè és molt fàcil caure en la temptació de començar a escriure el programa tot i no haver previst tots els detalls.

Aquestes ànsies per començar a programar són un problema en qualsevol tipus de programa, però si intervenen elements gràfics aquests problemes es poden agreujar perquè l'ordre com realitzem les tasques pot ser més crític que quan fem certes operacions amb dades.

Per a intentar minimitzar els possibles problemes és molt interessant tenir una descripció prou detallada del que volem aconseguir, i aquí és on els *frames* ens poden ajudar.

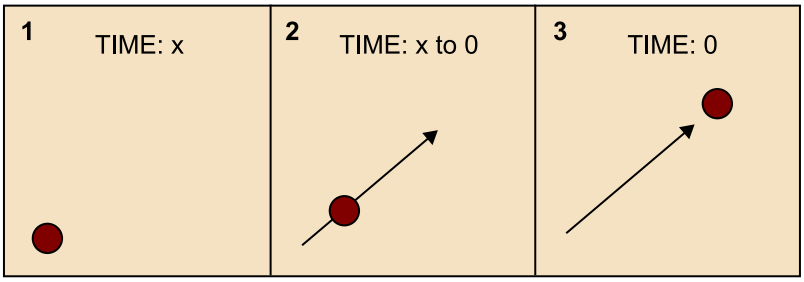
Podem entendre un *frame* com un moment concret del nostre programa, i es podria fer una analogia amb els *frames* que s'utilitzen amb els programes d'autoria, com per exemple l'Adobe Flash, en què per a cada *frame* definim uns elements gràfics i una sèrie d'accions que es poden fer en aquest moment determinat.

6.2.1. El guió animat i el diagrama d'estats

En planificar un programa serà estrany si es pot resoldre amb un únic *frame*, i és per això que per a agrupar-los utilitzarem un guió animat, de la mateixa manera que si estiguéssim preparant una pel·lícula, i el més important és que contingui la descripció de tots els *frames* que componen el nostre programa.

Per exemple, suposem que volem fer un programa en què podrem definir el temps que una bola s'estarà movent, i que un cop finalitzi el moviment es quedi quieta fins que li indiquem que torni al seu punt inicial. El guió animat resultant podria ser similar a aquest:

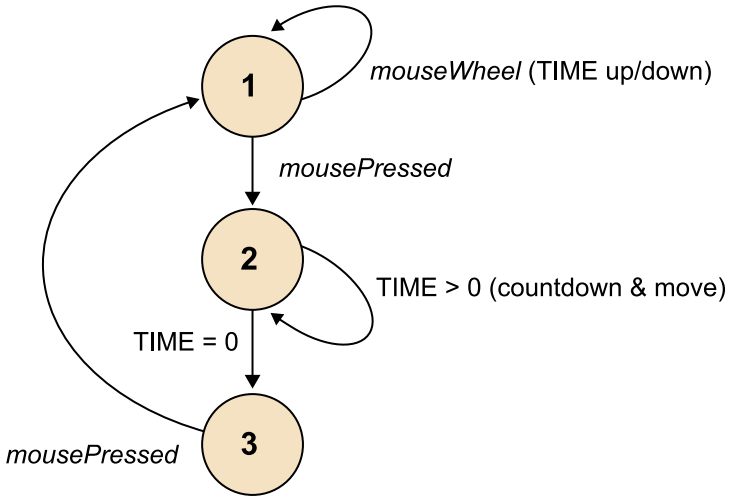
Figura 35. Guió animat



Fixeu-vos que no cal que tingui gràfics increïbles, el més important és que ens descrigui els moments clau del nostre programa perquè ho puguem tenir en compte en crear-lo.

Ara el que ens cal és afegir què podem fer en cada *frame* i com passarem d'un a l'altre, i per a fer-ho utilitzarem el que s'anomenen *màquines d'estats*:

Figura 36. Màquina d'estats



El que hem fet és agafar cada *frame* i dibuixar-lo dins d'un cercle. A continuació amb fletxes indiquem quines accions o esdeveniments es poden produir i quines conseqüències tenen. Ara els veurem:

1) Estat o *frame* 1: la bola està quieta a la part inferior esquerra de la finestra i hi podem interactuar de dues maneres diferents:

- Amb la roda del ratolí modifiquem el temps que voldrem que es mogui la bola.
- En fer clic al ratolí saltarem al *frame* 2.

2) Estat o *frame* 2: en aquest estat la bola es començarà a moure i s'iniciarà el compte enrere.

- Mentre el temps pendent sigui superior a 0 la bola es continua movent.
- Quan el temps sigui 0, saltarem a l'estat següent.

3) Estat o *frame* 3: aquí la bola quedarà quieta fins que cliquem el ratolí, moment en què tornarem al *frame* 1.

Amb aquests dos elements tenim clar què volem aconseguir sense haver escrit ni una línia de codi; per tant, aquest sistema ens serveix tant per a casos com el Processing, en què haurem de programar un codi, com per a casos en què tinguem una eina gràfica com l'Adobe Flash que ens ho permet fer més visualment. L'important d'aquesta feina prèvia és poder visualitzar ràpidament el nostre programa per a tenir una referència clara del que hem de fer a cada moment, i així simplificar el pas següent, que és la codificació.

6.2.2. Del guió animat al codi: l'estructura switch

Ara que tenim el guió animat i la màquina d'estats creats ens cal un sistema per a codificar-ho de la manera més senzilla possible, i per a fer-ho utilitzarem una estructura de control que té certes similituds a una que ja havíem vist: *if...else*.

Amb *if...else* podem comprovar una condició i realitzar una acció si es compleix o una altra si no es compleix. Però, què passa si hi pot haver més opcions? Una solució és utilitzar estructures *if...else* imbricades:

```
if (c == 1){
   //Do 1
}else{
   if (c == 2){
      //Do 2
   }else{
      if (c == 3){
         //Do 3
      }else{
         //Do 4
      }
   }
}
```

Però aquest sistema ens pot generar estructures molt carregades i que a la llarga compliquen la comprensió del codi si no som gaire estructurats. La solució és utilitzar l'estructura de control *switch*:

```
switch(num) {
   case 0:
      println("Zero");
      break;
   case 1:
      println("One");
      break;
   default:
      println("None");
      break;
}
```

Amb aquesta estructura primer de tot indiquem la variable que volem comprovar, que en el nostre cas s'anomena *num*. A partir d'aquí sempre farem el mateix:

1) ***case X***: comprova si la variable indicada és igual al valor X, i en cas afirmatiu s'executa el codi que hi ha a partir d'aquest punt.

2) *break*: fixeu-vos que la comprovació que fa *case* acaba amb dos punts en comptes d'utilitzar els claudàtors *{}*, com és habitual. Per això cal que indiquem al final el codi que hem d'executar dins de cada *case* amb la instrucció *break*. Si no ho féssim es continuarien executant totes les línies de codi encara que formessin part del *case* següent.

3) *default*: ens permet definir l'acció per defecte que volem fer si no hem trobat cap *case* que compleixi amb la condició.

A diferència de l'estructura *if...else*, podem veure com tenim un codi molt més clar, tot al mateix nivell independentment del nombre de condicions que vulguem comprovar, cosa que ens simplificarà el seguiment del que estem fent a cada moment.

Ara veurem com quedaria l'estructura de control del nostre guió animat:

```
int frame;

void setup() {
   size(400, 400);
   frameRate (20);

   frame = 1;
}

void draw() {
   background(0, 0, 0);

   switch (frame){
      case 1:
         //Stopped ball, time control, waiting click
         break;
      case 2:
         //Moving ball, countdown
         break;
      case 3:
         //Stopped ball, waiting click
         break;
   }
}
```

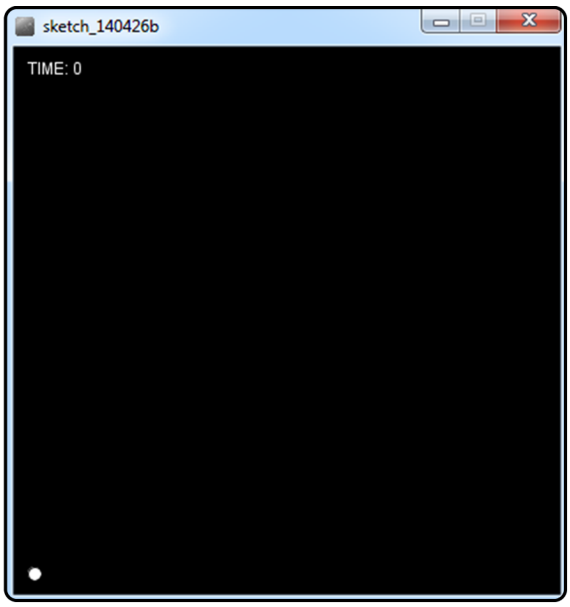
Definim i inicialitzem una variable anomenada *frame* que utilitzarem per a saber a cada moment en quin estat del programa ens trobem, i a continuació traduïm la nostra màquina d'estats a una estructura *switch*. De moment sim-

plement hem definit l'estructura, no hem realitzat cap acció, però el que és interessant és que ja tenim un codi estructurat que només cal completar amb les accions concretes.

Comencem a completar el codi, i el primer que farem serà afegir els elements gràfics que necessitem:

```
int frame;
int ball_x;
int ball_y;
int time;
String message;
float crono;

void setup() {
   size(400, 400);
   frameRate (20);

   frame = 1;
   ball_x = 15;
   ball_y = height - 15;
   time = 0;
   message = new String ("TIME: ");
   crono = 0.0;
}

void draw() {
   background(0, 0, 0);

   switch (frame){
      case 1:
         //Stopped ball, time control, waiting click
         break;
      case 2:
      //Moving ball, countdown
         break;
      case 3:
         //Stopped ball, waiting click
         break;
   }

   ellipse (ball_x, ball_y, 10, 10);
   text (message+time, 10, 20);
}
```

Figura 37. Estructura bàsica de màquina d'estats



Afegim les variables per a controlar la posició de la bola, el temps i el text del missatge que ensenyarem, i al final de la funció *draw* dibuixem la bola i el text.

Ara anirem codificant el que hem de fer en cada un dels estats o *frames*.

Estat 1

El primer que hem definit és que amb la roda del ratolí puguem modificar el temps que la bola s'estarà movent; per tant, haurem d'utilitzar la funció *mouseWheel*:

```
void mouseWheel(MouseEvent event) {
   float incr = event.getAmount();

   time = time + int(incr);
   if (time < 0){
      time = 0;
   }
}
```

A partir de la informació que ens retorna el ratolí actualitzem la variable *time*, però en aquest cas hem de fer una feina extra. La nostra variable és un nombre enter, mentre que la informació del ratolí és un nombre decimal, i per defecte no podem sumar nombres de tipus diferents.

Per a solucionar-ho utilitzem la funció *int*, que s'encarrega de transformar el nombre decimal en un d'enter, i d'aquesta manera sí que podem actualitzar correctament la nostra variable.

Si executeu el programa podreu comprovar com ja s'actualitza el temps a la finestra i que a més a més mai no podrà ser negatiu, ja que també hem afegit aquesta comprovació en el nostre codi.

Ara ens cal detectar quan cliquem el ratolí per a anar a l'estat següent. Per a fer-ho havíem vist que existia la funció *mousePressed*, però en aquest cas utilitzarem una variable que també es diu igual (*mousePressed*) i que conté la informació de si s'ha premut algun boto del ratolí; ho farem d'aquesta manera:

```
void draw() {
   background(0, 0, 0);

   switch (frame){
      case 1:
         //Stopped ball, time control, waiting click
         if (mousePressed){
            crono = time;
            frame = 2;
         }
         break;
      case 2:
         //Moving ball, countdown
         break;
      case 3:
         //Stopped ball, waiting click
         break;
   }

   ellipse (ball_x, ball_y, 10, 10);
   text (message+time, 10, 20);
   text (frame, width-20, 20);
}
```

Dintre del *case 1*, comprovem si aquesta variable és certa, i en cas afirmatiu diem que l'estat actual ja no és el nombre 1, sinó que serà el 2, i a més inicialitzarem el cronòmetre amb el temps que tenim seleccionat.

A part, hem afegit un altre text a la part superior dreta de la finestra que ens indica el pla actual, i d'aquesta manera tindrem una indicació mentre estiguem fent el programa.

Estat 2

Quan arribem a aquest estat vol dir que hem seleccionat un temps i que hem d'iniciar el moviment de la bola, i per tant afegirem el codi per controlar el temps:

```
void draw() {
   background(0, 0, 0);

   clock();

   switch (frame){
      case 1:
         //Stopped ball, time control, waiting click
         if (mousePressed){
            crono = time;
            frame = 2;
         }
         break;
      case 2:
         //Moving ball, countdown
         countdown();
         time = int (crono);
         break;
```

Afegim dues funcions noves, *clock*, que s'encarrega de comptar el temps que ha passat entre cada execució de la funció *draw*, i la funció *countdown*, que actualitzarà el temps que ens falta per a parar la bola.

Utilitzem funcions extres per no sobrecarregar la funció *draw* i que continuï essent el màxim de clara possible. Un cop actualitzat el temps fem el mateix amb la variable *time*, que és la que utilitzem per a pintar per pantalla.

La funció *clock* és aquesta:

```
void clock() {
   time_now = millis();
   time_delta = time_now - time_old;
   time_old = time_now;
}
```

Com podeu veure, és el mateix codi que ja havíem utilitzat anteriorment. L'única diferència és que ara el codi està en una funció a part.

La funció *countdown* és molt simple:

```
void countdown() {
   crono = crono - (time_delta / 1000.0);
}
```

Si executeu el codi veureu que el temps va disminuint, però en arribar a zero no s'atura i continua amb temps negatius. Aquesta és una condició que ja havíem definit en la nostra màquina d'estat, i ara la implementarem:

```
case 2:
         //Moving ball, countdown
         countdown();
         time = int (crono);
         if (time <= 0){
            frame = 3;
         }
         break;
```

L'únic que ha calgut ha estat indicar que si el comptador arriba a zero saltem a l'estat 3, i com que en aquest estat no descomptem el temps aquest ja no apareix amb nombres negatius.

Ara ja només ens falta que la bola es mogui per la pantalla, i per a fer-ho crearem una altra funció que s'encarregarà de moure-la, que cridarem des del *case*, com hem fet amb el cronòmetre.

```
case 2:
         //Moving ball, countdown
         countdown();
         time = int (crono);
         move_ball();
         if (time <= 0){
            frame = 3;
         }
         break;
```

```
void move_ball() {
   ball_x = ball_x + 1;
   ball_y = ball_y - 1;
}
```

Part de la gràcia de crear funcions extres és que les podem reutilitzar des de diversos llocs del programa, i si en algun moment volguéssim canviar el tipus de moviment de la bola només caldria modificar aquesta funció per a aplicar-ho automàticament a tots els llocs que utilitzin aquesta funció.

Si tornem a executar el codi veureu com mentre estem a l'estat 1 podem modificar el temps, i en clicar el ratolí saltem a l'estat 2. En aquest estat la bola es comença a desplaçar i el temps decreix fins a arribar a zero, moment en què passem a l'estat 3 automàticament.

Estat 3

Aquest estat és molt simple: simplement estarem esperant que l'usuari premi algun botó del ratolí. Quan ho faci direm que l'estat torna a ser l'inicial i podrem tornar a controlar el temps que volem que la bola es desplaci.

```
case 3:
         //Stopped ball, waiting click
         if (mousePressed){
            frame = 1;
            ball_x = 15;
            ball_y = height - 15;
         }
         break;
```

I després de codificar aquest últim estat ja tenim el programa fet. Hem començat definint què volíem fer i hem anat avançant pas per pas per poder construir el programa d'una manera ordenada. Tot i que no és l'única manera de programar, sí que és molt interessant tenir alguna metodologia que ens obligui d'alguna manera a ser ordenats des del punt de vista del disseny del programa (guió animat i màquina d'estats), i de la codificació (funcions).

7. Treballar amb imatges

És el moment d'afegir una mica d'alegria als nostres programes, i ho farem utilitzant imatges. Penseu quina quantitat d'elements gràfics que hi ha en qualsevol programa actual: icones, fons, cursors, elements animats, etc.

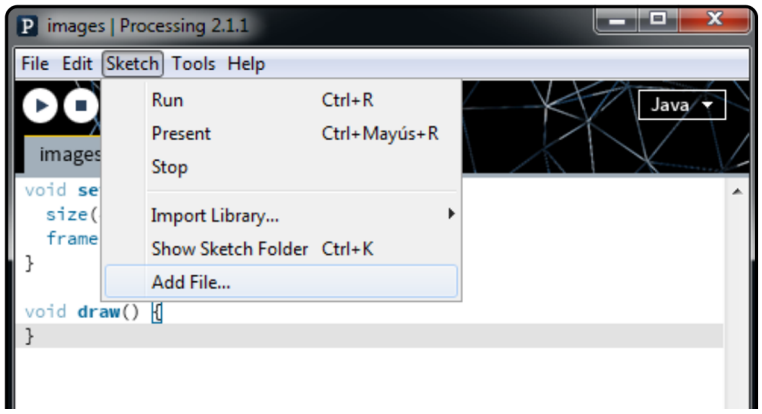
7.1. Elements bàsics

Com sempre, comencem amb un codi de base per poder-lo evolucionar de mica en mica. Si l'executeu veureu que s'obre una finestra de color gris:

```
void setup() {
   size(800, 331);
   frameRate (20);
}

void draw() {
}
```

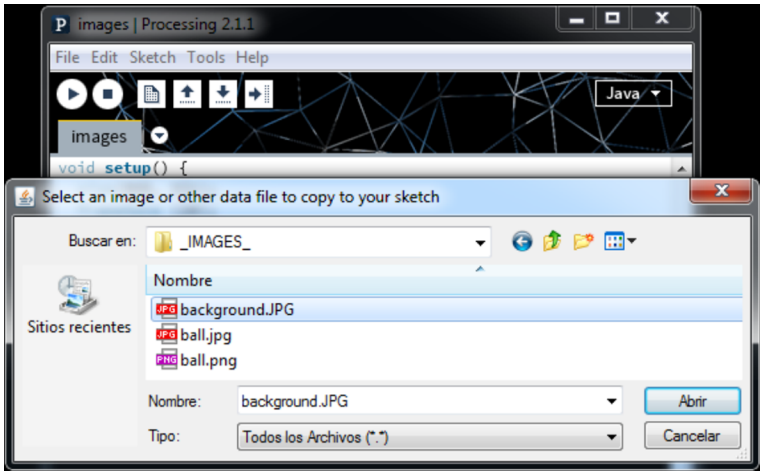
El següent que hem de fer per a poder utilitzar imatges és carregar-les en l'editor del Processing. Això ho fem anant al menú *Sketch*, opció *Add File*.

Figura 38. Carregar imatge (1)



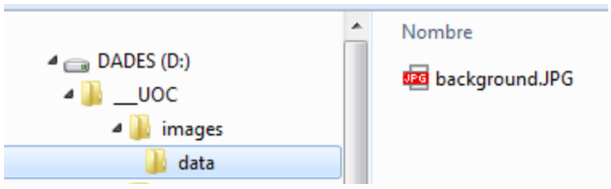
I a continuació seleccionem la imatge que volem carregar:

Figura 39. Carregar imatge (2)



Això el que fa és copiar la imatge dins del directori on hi ha desat el codi, concretament dintre d'una carpeta anomenada *data*. Si no volem fer aquest pas utilitzant els menús també podem fer la còpia de les imatges manualment en aquesta carpeta.

Figura 40. Carregar imatge (3)



Aquí podem veure la carpeta on hi ha el codi, que es diu *images*, que conté la carpeta *data*, que és on hi haurà les imatges i la resta d'elements que puguem necessitar (sons, vídeos...).

Ara que tenim el fitxer de la imatge en el lloc correcte per a poder-hi accedir, la carregarem i utilitzarem en el nostre programa. Modifiquem el codi d'aquesta manera:

```
PImage img;

void setup() {
   size(800, 331);
   frameRate (20);

   img = loadImage ("background.JPG");
}

void draw() {
   image (img, 0, 0);
}
```

Figura 41. Carregar una imatge de fons

Analitzem els canvis que hem afegit:

1) Definim una variable del tipus *Pimage*. Aquest tipus de dada ens permet emmagatzemar-hi imatges i fer-hi operacions bàsiques.

2) Inicialitzem la variable que hem creat utilitzant la funció *loadImage*, aquesta funció carrega la imatge tot suposant que el fitxer es troba en el directori *data*.

És molt important indicar el nom del fitxer tenint en compte les majúscules i minúscules, ja que si no coincideix exactament ens donarà un error i no el carregarà.

També és important carregar les imatges dintre de la funció *setup*, ja que el procés pot trigar una mica, i si ho féssim després mentre estem interactuant amb el programa hi podria haver moments de lentitud.

3) Per a pintar la imatge a la finestra utilitzem la funció *image*, i li passem la variable que conté la imatge i la posició de la cantonada superior esquerra d'aquesta. En el nostre cas la imatge és de la mateixa mida que la de la finestra que hem definit.

7.2. Tipus d'imatges que podem utilitzar

Els tipus o formats que ens poden interessar més són aquests tres:

- **JPG**: és el format que utilitzarem més, ens permet carregar imatges i mantenir una mida de fitxer reduïda.

- **GIF**: pot ser útil per a imatges petites amb detalls molt fins com els de les icones, per exemple, però l'ús més interessant és que podrem carregar imatges animats i veure aquesta animació en executar el programa.

- **PNG**: aquest format ens permet treballar amb imatges amb zones transparents, cosa que ens pot donar força joc. En veurem un exemple a continuació.

7.3. Afegim una bola

Carregarem una altra imatge i l'afegirem al codi d'aquesta manera:

```
PImage img;
PImage ball;

void setup() {
   size(800, 331);
   frameRate (20);

   img = loadImage ("background.JPG");
   ball = loadImage ("ball.jpg");
}

void draw() {
   image (img, 0, 0);
   image (ball, 20, 20);
}
```

Figura 42. Una bola "maquillada"

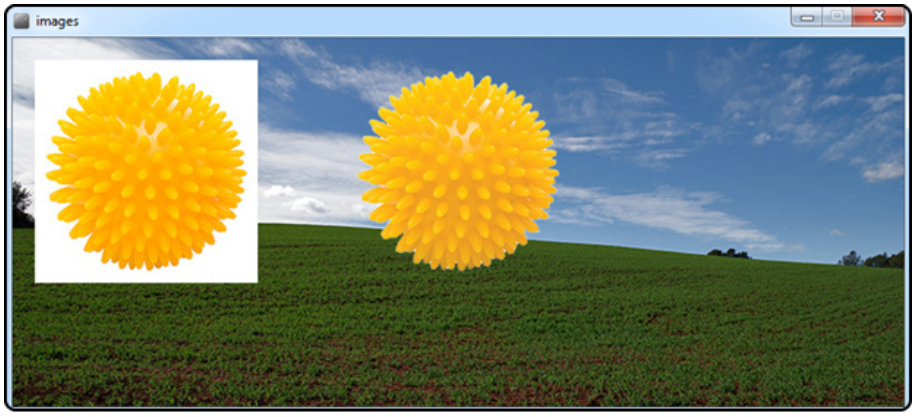
Per fi tenim una bola més interessant que les que havíem utilitzat fins ara. Podríem modificar tots els exemples que havíem vist fins ara substituint el cercle que dibuixàvem per la funció *image* i continuarien funcionant igual però amb uns gràfics millorats.

És clar que tenim un petit problema, ja que si la imatge és rectangular els marges s'adapten perfectament, però en aquest cas ens apareix un marge que ens molesta. Ara ho solucionarem:

```
PImage img;
```

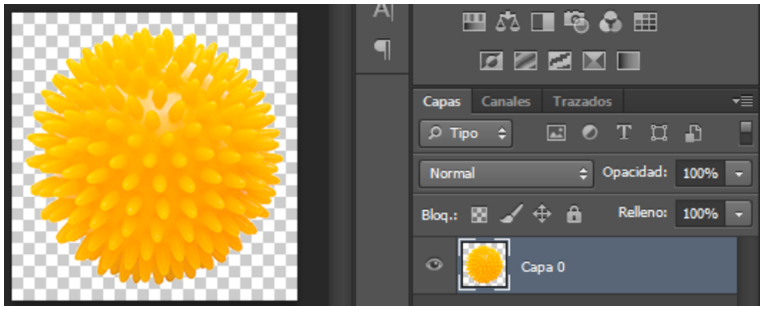
```
PImage ball;
PImage ball_alfa;

void setup() {
   size(800, 331);
   frameRate (20);

   img = loadImage ("background.JPG");
   ball = loadImage ("ball.jpg");
   ball_alfa = loadImage ("ball.png");
}

void draw() {
   image (img, 0, 0);
   image (ball, 20, 20);
   image (ball_alfa, 300, 20);
}
```

Figura 43. Imatges transparents



Quina diferència, oi? Hem carregat una altra imatge amb format PNG que hem preparat amb el Photoshop. En aquest cas el que hem fet és editar la imatge perquè quedés en una capa amb el fons transparent. Aquest és l'aspecte de la imatge vista des del Photoshop:

Figura 44. Preparem la imatge amb el Photoshop

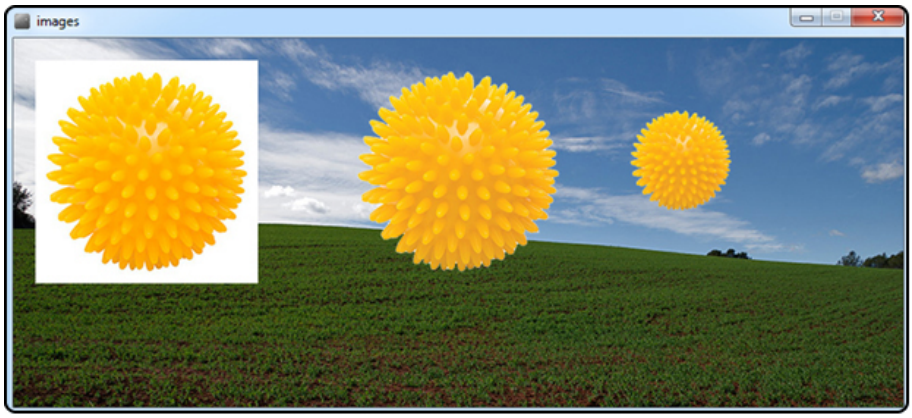


Ara si utilitzem aquesta imatge la integració amb el fons és perfecta.

Per finalitzar, modificarem les dimensions en què pintem aquesta imatge:

```
void draw() {
   image (img, 0, 0);
   image (ball, 20, 20);
   image (ball_alfa, 300, 20);
   image (ball_alfa, 550, 60, 100, 100);
}
```

Figura 45. Redimensionar la imatge



Hem pintat la mateixa imatge transparent, però aquesta vegada en comptes d'indicar només el nom de la variable i la posició hem afegit dos paràmetres que ens permeten definir la mida horitzontal i vertical de la imatge.

En el nostre exemple hem indicat una mida de 100×100 i la imatge es veu correctament perquè és quadrada, però no hi ha res que ens prohibeixi indicar una mida de 200×100, per exemple, i obtenir una imatge deformada. Us animem a provar-ho.

8. Transformacions bàsiques

Ara que ja sabem com podem generar formes bàsiques i utilitzar imatges és possible que ens calgui adaptar-les a certes situacions. Per exemple, quan desplacem la bola potser ens agradaria que fimbrés una mica com si fos una pilota de plàstic.

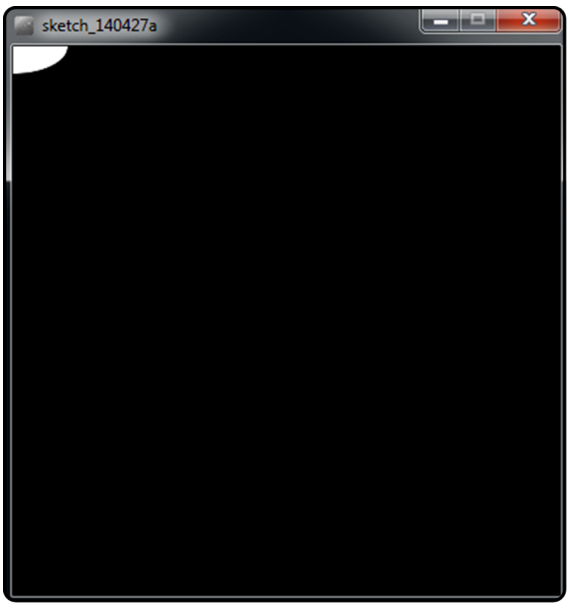
Per a donar solució a aquesta necessitat i a altres podem utilitzar les transformacions.

8.1. Transformacions

Veurem tres transformacions bàsiques: rotació, translació i escala, i per a fer-ho utilitzarem aquest codi com a punt de partida, en què dibuixem una el·lipse centrada en el punt (0, 0):

```
void setup() {
   size(400, 400);
   frameRate (20);
}

void draw() {
   background(0, 0, 0);
   ellipse (0, 0, 80, 40);
}
```

Figura 46. Codi base de transformacions

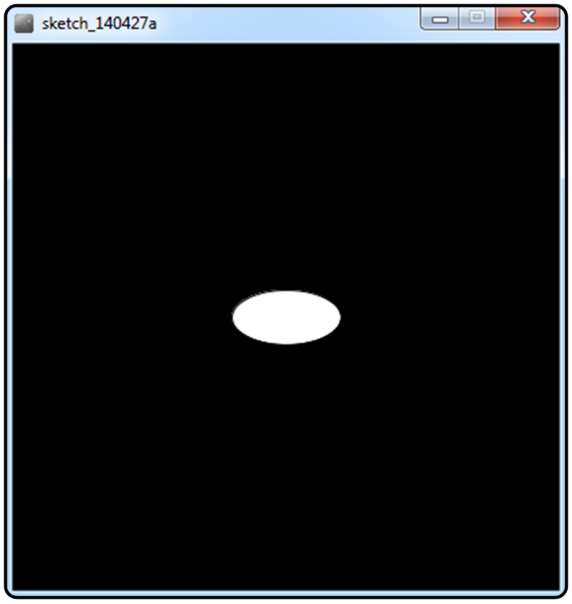


8.1.1. Translació

El primer que farem serà desplaçar l'el·lipse al centre de la finestra, i per a fer-ho utilitzarem la funció *translate* d'aquesta manera:

```
void draw() {
   background(0, 0, 0);
   translate ((width/2), (height/2));
   ellipse (0, 0, 80, 40);
}
```

Figura 47. Translació



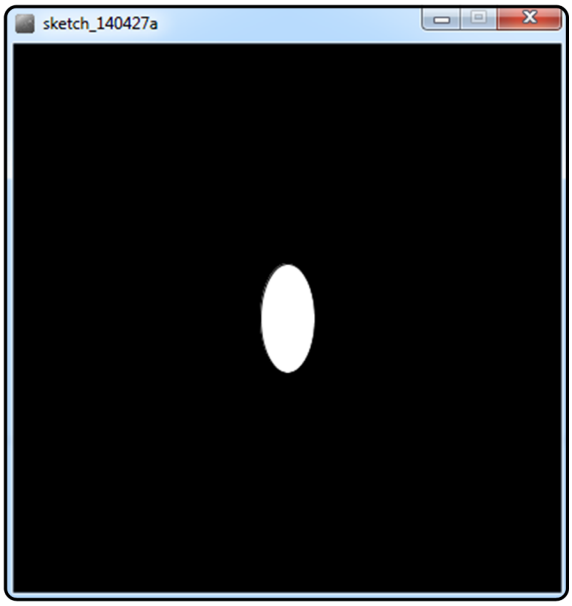
A la funció només cal que li indiquem el desplaçament que volem aplicar als elements que dibuixarem; en el nostre cas hem dit que es desplacés l'equivalent a la meitat de l'amplada de la finestra i la meitat de l'alçada; per tant, la figura ha quedat centrada. Un cop executada aquesta funció a tot el que dibuixem a la finestra se li aplicarà la mateixa translació.

8.1.2. Rotació

Vegem el codi directament:

```
void draw() {
   background(0, 0, 0);
   translate ((width/2), (height/2));
   rotate (HALF_PI);
   ellipse (0, 0, 80, 40);
}
```

Figura 48. Rotació



En aquest cas utilitzem la funció *rotate* i hi indicarem l'angle que volem rodar en graus radiants. Nosaltres hem seleccionat π 1/2, és a dir, 90 . Si voleu utilitzar graus, en podeu fer la conversió amb la funció radiants o la inversa amb la funció *degree*:

```
Xradians = radians (Ydegree);
Ydegree = degrees (Xradians);
```

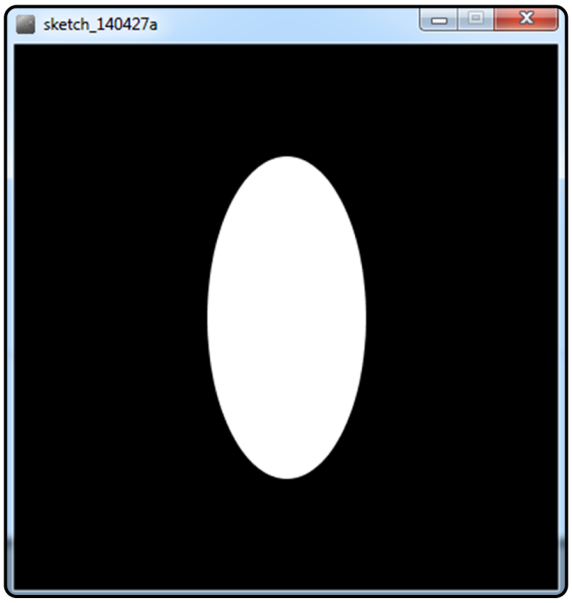
Una cosa que hem de tenir en compte és que les rotacions sempre agafen com a punt de referència les coordenades (0, 0) i això pot fer que a vegades no tinguem el resultat que ens esperaríem *a priori*, però això ho veurem amb més detall una mica més endavant.

8.1.3. Escala

Per finalitzar, veurem la funció *scale*, que ens permet fer un escalat proporcional (igual per a l'eix *x* i *y*) o aplicar diferents factors a cada eix per a deformar els objectes:

```
void draw() {
   background(0, 0, 0);
   translate ((width/2), (height/2));
   rotate (HALF_PI);
   scale (3.0);
   ellipse (0, 0, 80, 40);
}
```

Figura 49. Escala



En aquest cas hem fet un escalat amb un factor 3 (3 vegades més gran) i que afecta els dos eixos per igual. El mateix resultat el podríem haver obtingut amb aquest codi:

```
scale (3.0, 3.0);
```

La diferència és que ara hem indicat l'escala dels eixos per separat, i per tant podríem haver aplicat factors diferents a cada un.

Igual que en el cas de la rotació, l'escala s'aplica agafant com a referència l'origen de coordenades (0, 0), factor que haurem de tenir en compte en utilitzar la transformació.

8.2. El problema de la referència a l'origen de coordenades

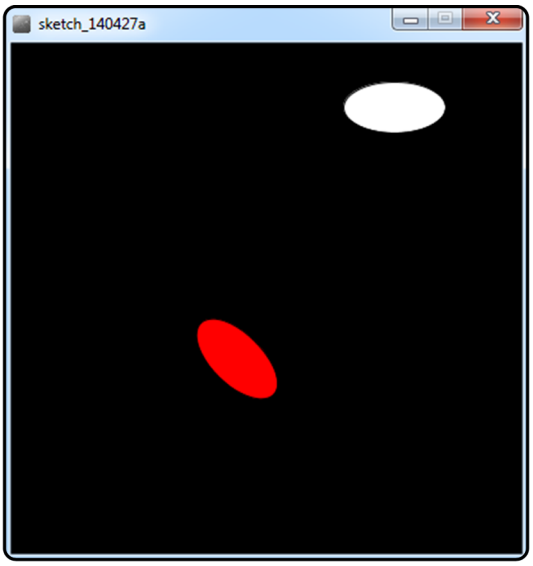
Com hem dit, les transformacions de rotació i escala utilitzen com a referència l'origen de coordenades, i en realitat la translació també ho fa. Però, què vol dir això?

Per veure-ho mirem un exemple:

```
void setup() {
   size(400, 400);
   frameRate (20);
}

void draw() {
   background(0, 0, 0);
   fill (255, 255, 255);
   ellipse (300, 50, 80, 40);
```

```
    fill (255, 0, 0);
    rotate (HALF_PI/2);
    ellipse (300, 50, 80, 40);
}
```

Figura 50. Transformacions i origen de coordenades (1)



En el codi podem veure com hem dibuixat dos el·lipses de la mateixa mida i a la mateixa posició; l'única diferència és que a la vermella se li ha aplicat una rotació. El problema és que aquesta rotació segurament no és la que esperàvem, i potser ens hauria agradat que la figura quedés girada en el mateix lloc, no?

En realitat el que ha passat és que s'ha rodat tenint com a origen de la transformació les coordenades (0, 0):

Figura 51. Transformacions i origen de coordenades (2)

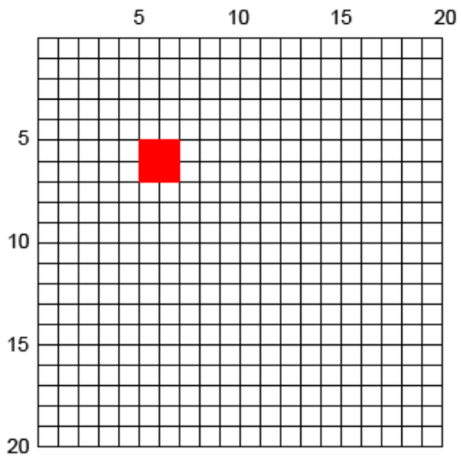
Si tenim clar que el punt sobre el qual es roda és l'origen de coordenades, sí que es pot entendre el resultat que hem obtingut, però això ens planteja una pregunta: com podem aconseguir el resultat que esperàvem?

La solució és moure l'origen de coordenades i situar-lo en el punt que volem fer servir de referència per a la rotació, i per a fer això utilitzem la funció *translate*. Què estrany, no havíem dit que aquesta funció s'utilitzava per a desplaçar el que dibuixàvem?

Doncs sí, el resultat és que en utilitzar *translate* els objectes apareixen desplaçats, però no és perquè els canviï les coordenades, sinó perquè ha modificat l'origen de coordenades que utilitza per a dibuixar-los. Intentem aclarir-ho amb un exemple pas per pas; primer creem un quadrat:

```
rect (5, 5, 2, 2);
```

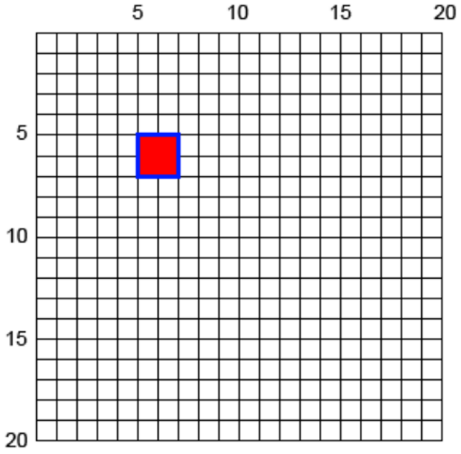
Figura 52. Transformacions i origen de coordenades (3)



Fins aquí cap misteri, ara en crearem un altre al mateix lloc però en comptes de reomplir-lo amb un color sòlid només el contornejarem de color blau:

```
void draw() {
   background(0, 0, 0);
   fill (255, 0, 0);
   stroke (255, 0, 0);
   rect (5, 5, 2, 2);

   noFill ();
   stroke (0, 0, 255);
   rect (5, 5, 2, 2);
}
```

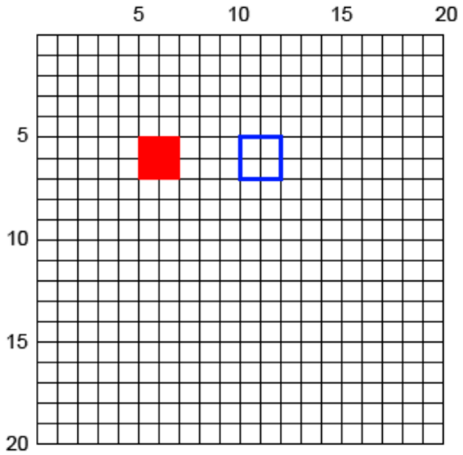
Figura 53. Transformacions i origen de coordenades (4)



D'acord, ara desplaçarem el segon requadre amb la funció *translate*:

```
void draw() {
   background(0, 0, 0);
   fill (255, 0, 0);
   stroke (255, 0, 0);
   rect (5, 5, 2, 2);

   translate (5, 0);
   noFill ();
   stroke (0, 0, 255);
   rect (5, 5, 2, 2);
}
```

Figura 54. Transformacions i origen de coordenades (5)

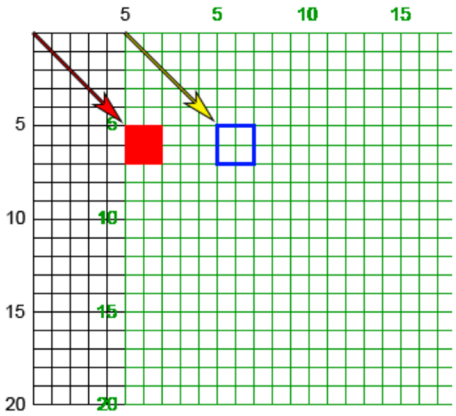


El quadrat s'ha dibuixat a la posició (10, 5), però fixeu-vos que nosaltres el continuem dibuixant a la posició (5, 5). En realitat en el moment que dibuixem el quadrat fem dues coses:

- Desplacem l'origen de coordenades a la posició (5, 0).

- Dibuixem el quadrat tenint en compte les noves coordenades de referència.

Figura 55. Transformacions i origen de coordenades (6)

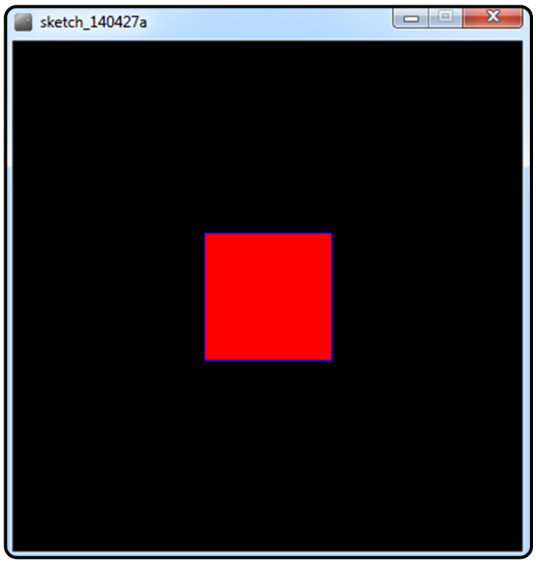


Amb aquesta imatge ho podem veure. El quadre vermell utilitza les coordenades (0, 0) originals (fletxa vermella), però per a dibuixar el blau primer hem desplaçat l'origen de coordenades (representades per la matriu de color verd) i després hem dibuixat el quadrat blau amb les noves coordenades de referència (fletxa groga).

Repetim el procés utilitzant la rotació, i comencem amb aquest codi:

```
void setup() {
   size(400, 400);
   frameRate (20);
}

void draw() {
   background(0, 0, 0);
   fill (255, 0, 0);
   stroke (255, 0, 0);
   rect (150, 150, 100, 100);

   noFill ();
   stroke (0, 0, 255);
   rect (150, 150, 100, 100);
}
```

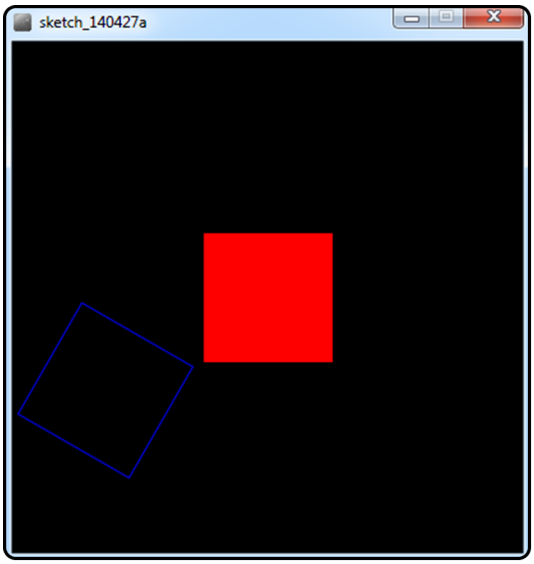
Figura 56. Transformacions i origen de coordenades (7)



Si rodem el quadre blau 30 graus sense tenir les coordenades en compte obtenim això:

```
noFill ();
stroke (0, 0, 255);
rotate (radians (30));
rect (150, 150, 100, 100);
```

Figura 57. Transformacions i origen de coordenades (8)

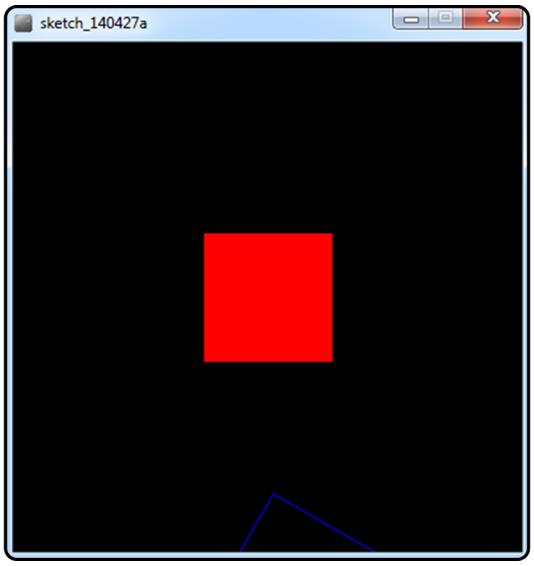


Per a solucionar-ho primer cal que desplacem l'origen de coordenades en el punt que volem fer servir com a centre de la rotació:

```
noFill ();
stroke (0, 0, 255);
translate (150, 150);
rotate (radians (30));
```

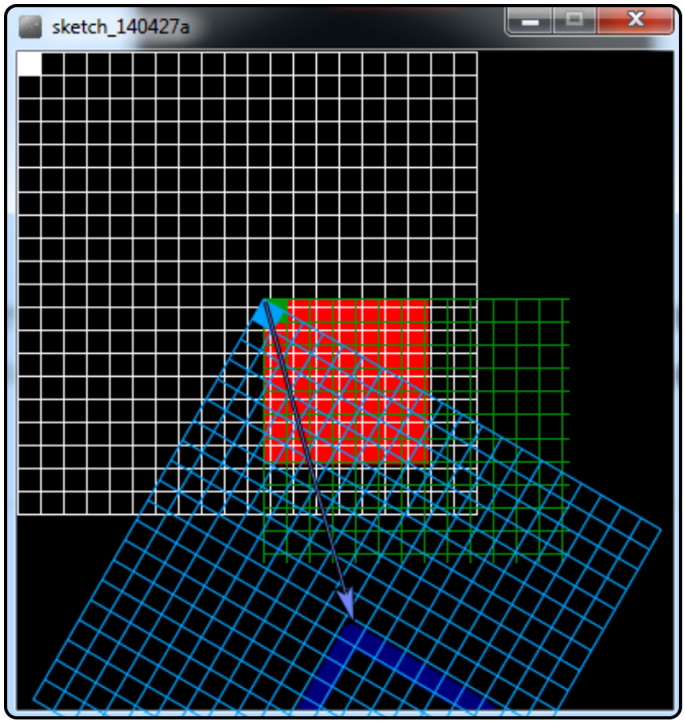
```
rect (150, 150, 100, 100);
```

Figura 58. Transformacions i origen de coordenades (9)



Però, que ha passat? Per què ens apareix el quadrat tan avall si només l'hem rodat? La resposta és com ha quedat situat l'origen de coordenades i on hem dibuixat el quadrat després:

Figura 59. Transformacions i origen de coordenades (10)

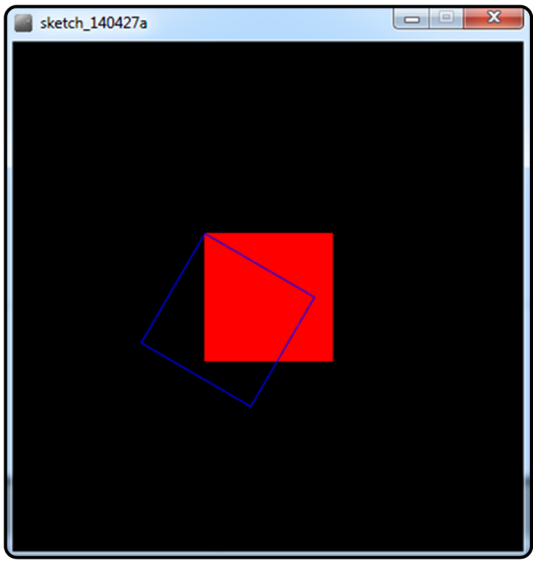


Primer de tot hem modificat les coordenades originals (color blanc) i les hem desplaçat a la nova posició (color verd), i després les hem rodat 30 graus (color blau).

Ara hem dibuixat el quadrat a la posició (150, 150) de les noves coordenades, però aquestes coordenades ja estaven en el lloc on volíem que estigués dibuixat el quadrat, i per tant el que caldrà fer és dibuixar-lo a les coordenades (0, 0):

```
noFill ();
stroke (0, 0, 255);
translate (150, 150);
rotate (radians (30));
rect (0, 0, 100, 100);
```

Figura 60. Transformacions i origen de coordenades (11)



Ara sí, per fi tenim el quadrat rodat de la manera que volíem.

Aquest sistema, que per a dibuixar un parell de quadrats sembla molt complex, fa que quan es treballa amb molts elements que depenen uns d'altres se simplifiqui força el treball. És per això que tant amb el Processing com amb qualsevol altre sistema de programació gràfica us trobeu amb solucions d'aquest estil.

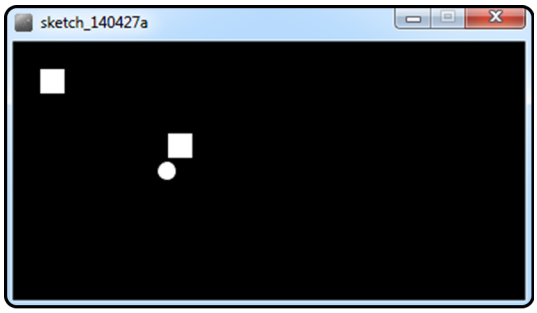
8.3. Àmbit de les transformacions

Quan treballem amb transformacions hem de tenir en compte que en el moment, que n'apliquem alguna, aquesta afectarà tots els elements que es dibuixin a partir d'aquest moment. Mirem aquest codi:

```
void setup() {
   size(400, 200);
   frameRate (20);
}

void draw() {
   background(0, 0, 0);
```

```
    rect (20, 20, 20, 20);
    translate (100,50);
    rect (20, 20, 20, 20);
    ellipse (20, 50, 15, 15);
}
```

Figura 61. Àmbit de les transformacions



Hem dibuixat un quadrat a la posició (20, 20) i després n'hem dibuixat un altre que volíem desplaçar 100 punts a la dreta i 50 punts avall, però el cercle que hem afegit després volíem que estigués a la posició (20, 50) i també ha aparegut desplaçat.

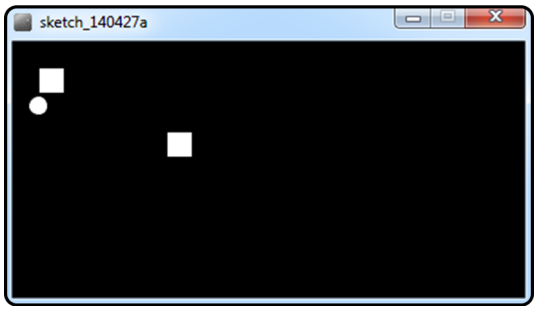
Això és perquè des del moment que hem aplicat la translació tot queda afectat, recordeu que en realitat hem modificat l'origen de coordenades. Per a solucionar això tenim diferents solucions.

8.3.1. Ordenació dels elements en funció de les transformacions

Aquest sistema es basa a aplicar i dibuixar els elements en funció de les transformacions que cal aplicar; en el nostre exemple el codi quedaria d'aquesta manera:

```
void draw() {
    background(0, 0, 0);
    rect (20, 20, 20, 20);
    ellipse (20, 50, 15, 15);
    translate (100,50);
    rect (20, 20, 20, 20);
}
```

Figura 62. Ordenació de les transformacions



Amb un exemple tan simple com aquest el resultat és correcte; el problema és que no sempre podrem intercanviar l'ordre en què pintem els elements (recordeu l'algorisme del pintor). També fa que no tinguem els elements que dibuixem per grups lògics que ens simplifiquin la interpretació del codi, sinó una agrupació en funció d'unes transformacions que s'han d'aplicar.

8.3.2. Invertir els efectes aplicats

Aquesta és una solució millor, ja que en comptes de reorganitzar el codi per adaptar-lo a les transformacions que apliquem desactivarem les transformacions que ja no volem utilitzar. Amb l'exemple es veurà més clar:

```
void draw() {
   background(0, 0, 0);
   rect (20, 20, 20, 20);
   translate (100,50);
   rect (20, 20, 20, 20);

   translate (-100, -50);
   ellipse (20, 50, 15, 15);
}
```

Tornem a dibuixar el cercle al final del codi, però en aquest cas just abans hem aplicat la translació inversa. Amb aquesta acció fem que l'origen de coordenades es torni a desplaçar a la seva situació original a la cantonada superior esquerra de la finestra, i per tant el cercle apareix on volíem.

Tot i que aquest sistema és molt millor que l'anterior, en casos en què hi ha múltiples transformacions i molts elements per dibuixar pot ser una mica difícil d'utilitzar, sobretot perquè cal fer un seguiment molt acurat de totes les transformacions que fem i després invertir-ne els efectes en l'ordre correcte.

8.3.3. La pila de transformacions

Per a solucionar la complexitat d'haver de recordar tots els efectes aplicats i l'ordre en què cal desfer-los tenim la pila de transformacions. La idea és que en aquesta pila podrem emmagatzemar diferents orígens de coordenades amb les seves transformacions associades i després recuperar-los.

Per a poder treballar amb la pila utilitzarem dues funcions:

- *pushMatrix*: desarà la situació actual a la pila, amb totes les transformacions que hi hagi aplicades.

- *PopMatrix*: recuperarà l'últim estat que havíem desat, de manera que deixarem d'aplicar les transformacions que teníem definides ara i tornarem a utilitzar les anteriors.

Vegem-ne un exemple:

```
void draw() {
   background(0, 0, 0);
   rect (20, 20, 20, 20);

   pushMatrix();
   translate (100,50);
   rect (20, 20, 20, 20);

   popMatrix();
   ellipse (20, 50, 15, 15);
}
```

Si executeu el codi veureu que s'estan dibuixant els tres elements tal com volíem. Primer dibuixem el quadrat a la posició (20, 20) i a continuació desem l'estat actual, que no té cap transformació aplicada, a la pila.

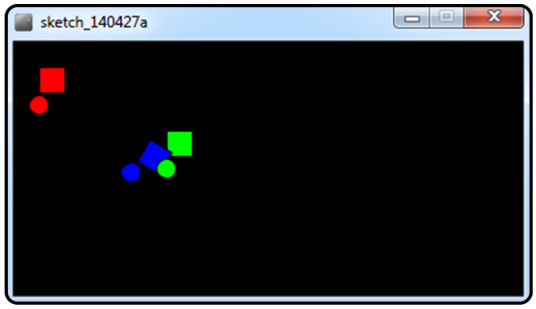
A continuació apliquem una translació i dibuixem el segon quadrat, que, com és natural, apareix desplaçat.

Ara és el moment en què volem dibuixar el cercle sense cap transformació, abans ens ha calgut desfer la translació però ara podem recuperar l'estat anterior amb el *popMatrix*. Utilitzant aquesta funció el que fem és recuperar l'estat que havíem desat (sense transformacions) i utilitzar-lo per a continuar dibuixant més elements.

A la pila podem anar posant i traient tants estats com vulguem, l'únic que hem de tenir en compte és que sempre s'ha de fer en ordre, és a dir, en fer el *popMatrix* sempre recuperarem l'últim que hi havíem posat. Mirem un exemple amb uns quants elements més:

```
void draw() {
   background(0, 0, 0);
   fill (255, 0, 0);
   rect (20, 20, 20, 20);

   pushMatrix();
   translate (100,50);
   fill (0, 255, 0);
   rect (20, 20, 20, 20);

   pushMatrix ();
   fill (0, 0, 255);
   rotate (radians(30));
   rect (20, 20, 20, 20);
   ellipse (20, 50, 15, 15);

   popMatrix();
   fill (0, 255, 0);
   ellipse (20, 50, 15, 15);

   popMatrix();
   fill (255, 0, 0);
   ellipse (20, 50, 15, 15);
}
```

Figura 63. Pila de transformacions



A part de dibuixar més elements, s'han codificat amb colors en funció de les transformacions que tenen aplicades per a poder veure millor com funciona la pila:

- Color vermell per a les figures sense cap transformació aplicada.
- Color verd per a les figures que només tenen la translació.

- Color blau per a les que tenen la translació i la rotació.

Si seguim el codi podem veure l'evolució següent:

1) Dibuixem el primer quadrat vermell sense cap transformació.

2) Desem l'estat a la pila (cap transformació).

a) Ens desplacem 100 punts a la dreta i 50 avall.

b) Dibuixem el quadrat verd.

c) Desem l'estat a la pila (translació).

- Apliquem una rotació de 30 , que se suma al desplaçament que ja teníem aplicat.
- Dibuixem el quadrat blau.
- Dibuixem el cercle blau.
- Recuperem l'últim estat que tenim a la pila.

d) Tornem a tenir únicament la translació aplicada.

e) Dibuixem el cercle verd.

f) Recuperem l'últim estat que tenim a la pila.

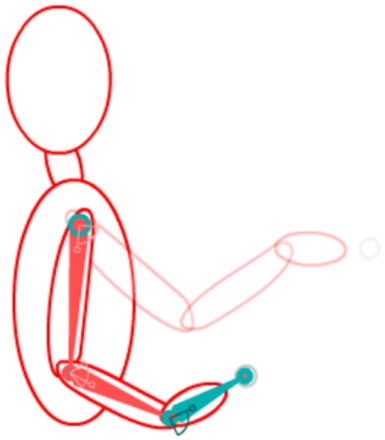
3) Tornem a la situació de no tenir cap transformació aplicada.

4) Dibuixem el cercle vermell.

Amb aquest sistema podem tenir un control més senzill de les transformacions aplicades fins i tot si el programa que s'està realitzant no és complex.

Un exemple en què és pràcticament obligatori utilitzar la pila de transformacions és amb les cadenes cinemàtiques (*kinematic chain*), que, per exemple, ens permetrien representar un braç.

Figura 64. Cadena cinemàtica



Aquest és l'exemple típic. Per a representar el braç tenim tres elements: braç, avantbraç i mà. Quan el braç gira, l'avantbraç i la mà també ho han de fer, i per tant s'ha d'aplicar una transformació que els afecti a tots, però si només es mou la mà, els altres no es veuen afectats.

Amb la pila de transformacions, de la mateixa manera que hem dibuixat els quadrats i cercles de diferents colors en funció de les transformacions aplicades, podríem controlar la posició a cada moment de tots els elements del braç i aniríem a un nivell o l'altre de la pila en funció del tros de braç que estiguéssim representant.

9. Biblioteques

El Processing és un llenguatge que ens ofereix l'oportunitat de crear una infinitat de programes per a solucionar qualsevol necessitat que puguem tenir. Naturalment, en funció de la complexitat del que vulguem obtenir el codi necessari també ho serà més o menys.

En alguns casos concrets, quan apareixen necessitats per a les quals el llenguatge no té una solució prou pràctica o directament no té cap solució, tenim disponibles les biblioteques.

Una biblioteca és una col·lecció de codi que afegeix funcionalitats que el llenguatge bàsic no té. Per exemple, el Processing ens permet dibuixar rectangles, però no té cap funció que dibuixi una poma. Una manera de solucionar això seria crear una biblioteca que disposés de la funció per a dibuixar una poma, i d'aquesta manera en comptes d'haver-la de dibuixar amb les primitives que té el Processing per defecte (línies, el·lipses, rectangles...) la podríem crear directament, potser amb una crida similar a aquesta:

```
apple (20, red);
```

Les biblioteques que utilitzem també estan programades en Processing, o Java, que és el llenguatge en què es basa, i per tant també ho hauríem pogut codificar nosaltres. L'avantatge de disposar de les biblioteques és que les podem fer accessibles a qualsevol programa que estiguem creant i que disposen d'una API (*application programming interface*) ben definida que ens informa de què podem fer i com ho hem de fer, de la mateixa manera que tenim la referència de Processing.

En aquest apartat farem la introducció bàsica a dues biblioteques que ens poden ser d'utilitat per a donar un valor afegit als nostres programes d'una manera més senzilla:

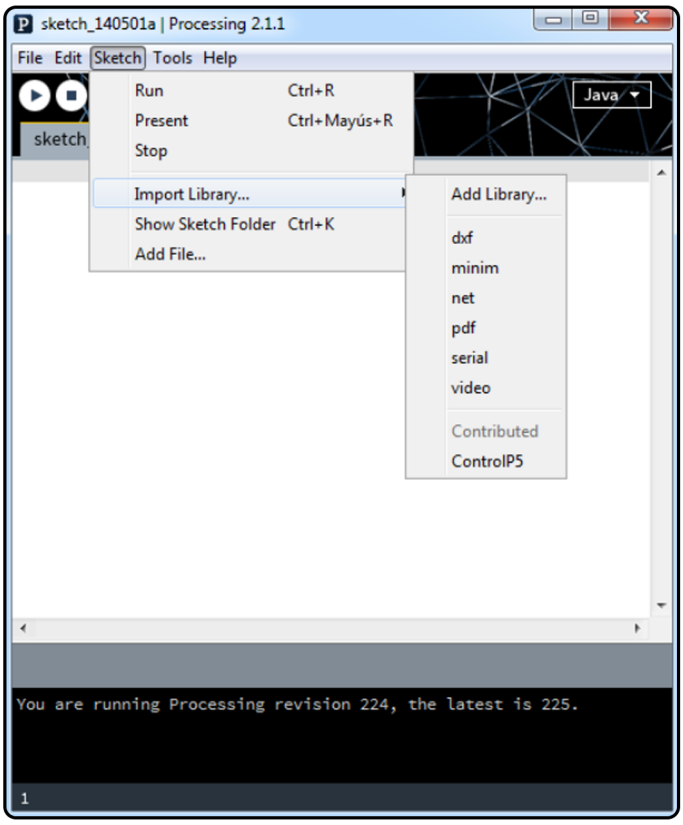
- *Minim*: és una biblioteca que per defecte ja està carregada i que ens permet treballar amb fitxers d'àudio.
- *ControlP5*: aquesta biblioteca no està carregada per defecte i ens permet crear interfícies gràfiques ràpidament amb botons, desplegables, etc.

Per saber quines biblioteques hi ha disponibles en podeu consultar la llista al web de Processing, on a més trobareu la descripció i com cal utilitzar-les.

9.1. Gestió de biblioteques

Per a saber quines biblioteques tenim disponibles podem anar al menú *Sketch*, *Import Library*, i trobarem una llista de les que hi ha instal·lades i podrem utilitzar.

Figura 65. Llista de biblioteques



A la imatge podeu veure que hi ha les biblioteques *Minim* i *ControlP5*, però si ho comproveu en el vostre PC no trobareu la *ControlP5*, i per tant serà necessari veure com s'instal·la.

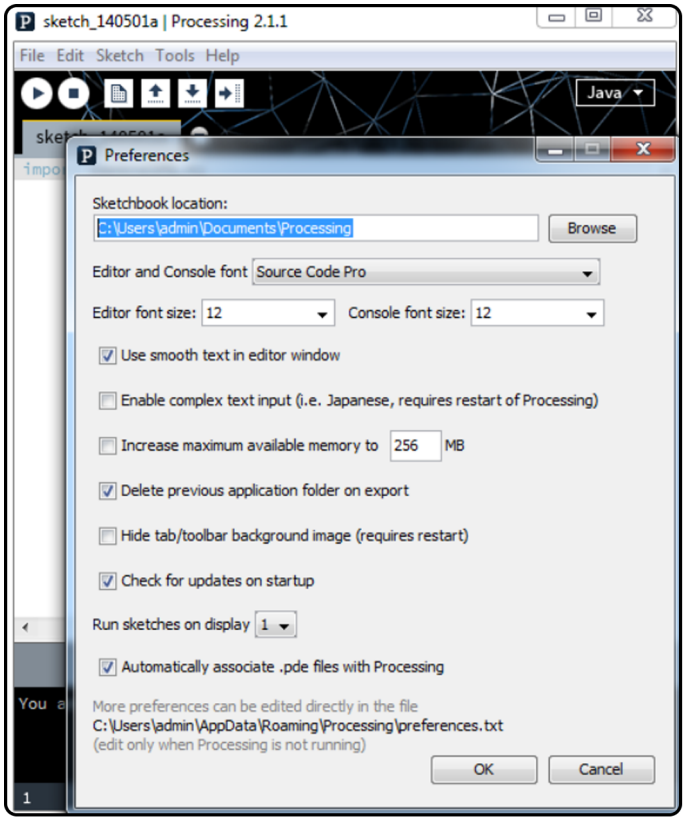
9.1.1. Instal·lació de la biblioteca *ControlP5*

Des de la plana web on hi ha la llista de totes les biblioteques, seleccionarem la *ControlP5* i saltarem directament al web oficial d'aquesta biblioteca. En aquest web, a part de trobar la biblioteca que cal instal·lar, també trobarem la referència per a poder utilitzar-la, exemples i molta documentació per a poder veure com funciona i el que podem arribar a fer.

De moment ens baixarem l'última versió disponible (el fitxer en format ZIP que hi ha a la zona de descàrregues), i descomprimirem el fitxer en un directori temporal. En fer-ho veureu que apareix un directori anomenat *controlP5*, que és la biblioteca, i un fitxer de text que explica com cal instal·lar-la.

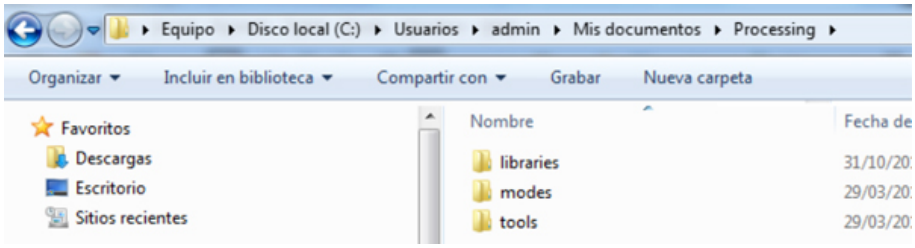
El procés és tan senzill com copiar el directori dins de la carpeta on el Processing busca les biblioteques, informació que podem trobar si anem a les preferències del Processing:

Figura 66. Instal·lació de biblioteques (1)



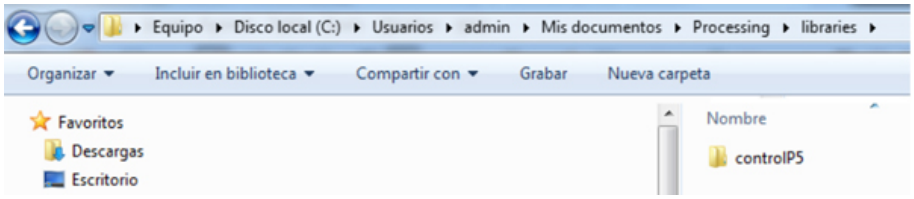
Copiem la ruta de l' *sketchbook location* i anem a l'explorador de fitxers:

Figura 67. Instal·lació de biblioteques (2)



Aquí trobarem el directori *libraries*, que conté les biblioteques que instal·larem nosaltres; entrem al directori i hi copiem el de la biblioteca que hem descomprimit:

Figura 68. Instal·lació de biblioteques (3)



Un cop fet això només cal reiniciar el programa perquè carregui les noves biblioteques i ja les podrem començar a utilitzar.

Per a utilitzar una biblioteca només cal anar a la llista de les que tenim disponibles i seleccionar-la, i veureu com automàticament s'afegirà un codi al vostre programa:

```
import controlP5.*;
```

Aquest és l'aspecte que té amb la biblioteca *controlP5*, i serveix per a indicar al programa que a part de les funcionalitats que té per defecte tindrà accés a d'altres que estan contingudes dins de la biblioteca que hem indicat.

9.2. Biblioteca *Minim*

Com hem dit, aquesta biblioteca ens permet treballar amb sons. A continuació veurem els passos bàsics per a poder-ne carregar i reproduir un.

Comencem amb aquest codi:

```
void setup()
{
   size(400, 200);
}

void draw()
{
```

El primer que farem serà indicar que volem utilitzar la biblioteca, i per tant anirem al menú *Sketch*, *Import Library* i seleccionarem la biblioteca *minim*. En fer-ho veureu que el codi queda d'aquesta manera:

```
import ddf.minim.spi.*;
import ddf.minim.signals.*;
import ddf.minim.*;
import ddf.minim.analysis.*;
import ddf.minim.ugens.*;
import ddf.minim.effects.*;

void setup()
{
   size(400, 200);
}

void draw()
{
```

Ha afegit una sèrie de crides *import* per a carregar totes les funcionalitats de la biblioteca, i a partir d'aquest punt ja podem començar a utilitzar-la.

La millor manera de saber què podem fer és consultar directament el web del creador de la biblioteca, on com hem dit trobarem tota la referència i exemples. En el nostre cas simplement carregarem un fitxer i el reproduirem; per a fer-ho necessitarem dues variables:

```
Minim minim;
AudioPlayer player;
```

La primera és la que utilitzarem per a accedir a les funcionalitats de la biblioteca, i la segona és la que ens permetrà carregar i reproduir els fitxers d'àudio (de manera similar a com ho fem amb les imatges). Ara les inicialitzarem:

```
void setup()
{
   size(400, 200);

   minim = new Minim(this);
   player = minim.loadFile("buzzer.mp3");
}
```

El més interessant d'aquest codi és la manera com carreguem el fitxer d'àudio, i el fet que abans d'utilitzar-lo cal que hàgim importat el fitxer *buzzer.mp3* a l'*sketch* tal com fem amb les imatges (recordeu: menú *sketch*, *add file*).

Ara que ja tenim el fitxer carregat només cal reproduir-lo:

```
void setup()
{
   size(400, 200);

   minim = new Minim(this);
   player = minim.loadFile("buzzer.mp3");

   player.play();
}
```

En executar el programa sonarà una sirena. Fixeu-vos que l'ordre de reproduir el so està dintre de la funció *setup*. Ho hem fet així perquè volem que el so es reprodueixi una sola vegada; si estigués dintre de la funció *draw* començaria una reproducció cada vegada que s'executés la funció *draw*. Proveu a modificar el codi i deixar-ho així:

```
void draw()
{
```

```
    player.play();
}
```

Quin mal de cap, no? Potser hem vist alguna cosa abans que ens serveixi per a millorar-ho:

```
void mousePressed() {
    player.play();
}
```

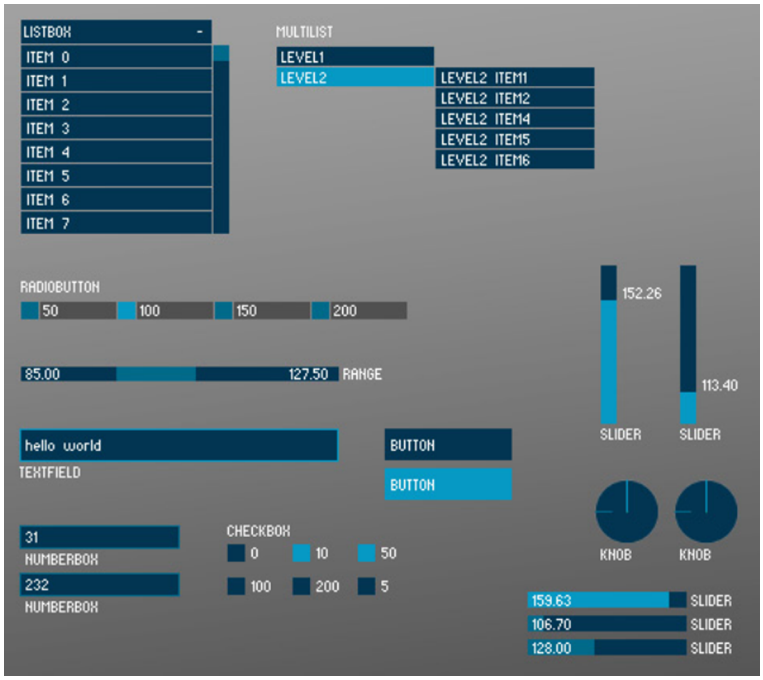
Molt millor, en deixar el codi així només sonarà l'alarma en prémer algun botó del ratolí.

Aquesta biblioteca té moltes més opcions, recordeu de fer una ullada a la referència oficial per a veure-les.

9.3. Biblioteca *ControlP5*

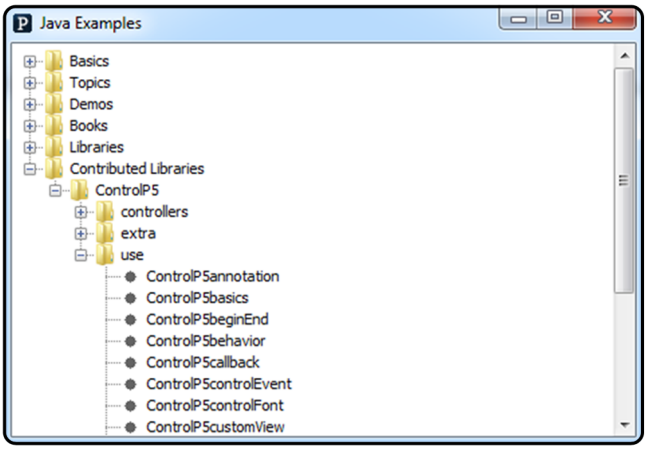
Aquesta biblioteca ens permet dissenyar interfícies gràfiques per als nostres programes d'una manera força senzilla. A continuació tenim una imatge en què podem veure els diferents controls que ens ofereix:

Figura 69. Controls de la biblioteca *ControlP5*



Una cosa molt interessant d'aquesta biblioteca és que té disponibles exemples bàsics de tots els seus elements, i ja els teniu instal·lats al vostre PC pel fet d'haver instal·lat la biblioteca. Aneu al menú *File*, *Examples*:

Figura 70. Exemples disponibles



En aquesta llista trobarem aquests codis d'exemple, i com podeu veure no solament són d'aquesta biblioteca, sinó que n'hi ha molts més.

A continuació veurem un programa bàsic en què afegirem un botó i que ens servirà per a veure l'estructura bàsica que podem utilitzar per a construir les nostres interfícies. Com sempre, començarem amb un codi bàsic:

```
void setup()
{
   size(400, 200);
}

void draw()
{
   background (0, 0, 0);
}
```

El pas següent és indicar que utilitzarem la biblioteca *ControlP5*:

```
import controlP5.*;
```

I ara ja la podem començar a utilitzar. Primer de tot definirem i inicialitzarem la variable que contindrà tots els elements de la interfície que vulguem afegir:

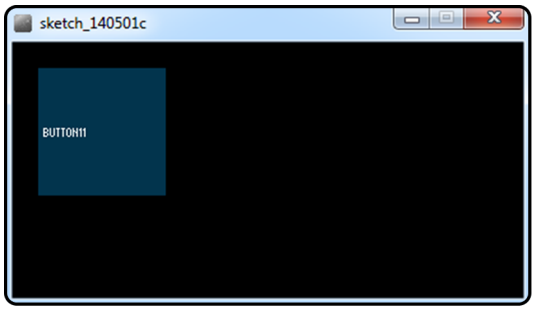
```
import controlP5.*;

ControlP5 userinterface;

void setup()
{
   size(400, 200);

   userinterface = new ControlP5 (this);
```

```
}
```

A partir d'ara utilitzarem la variable *userinterface* com a referència per a afegir i modificar la interfície que estem creant. Afegirem el botó:

```
void setup()
{
   size(400, 200);

   userinterface = new ControlP5 (this);

   userinterface.addButton ("button11", 1, 20, 20, 100, 100);
}
```

Figura 71. *ControlP5*: el primer botó



En executar el codi ens apareix el botó que acabem de definir, on hem indicat el nom, el valor associat que tindrà el botó (es pot considerar com un identificador), la posició (x, y) i la mida.

El següent que hem de fer és detectar quan han premut el botó, i per a fer-ho afegirem una funció nova:

```
void controlEvent(ControlEvent theEvent) {

   if(theEvent.isController()) {

      println("control event from : "+theEvent.controller().name()+", value :
                                    "+theEvent.controller().value());

      if(theEvent.controller().name().equals("button1")) {
         println ("BUTTON PRESSED");
      }
   }
}
```

Anem per parts:

1) Funció *controlEvent*: aquesta funció s'encarrega d'escoltar tots els esdeveniments o accions que es fan sobre els elements que pertanyen a l'usuari. Per exemple, si es mou una finestra, es desplaça el ratolí o cliquem el botó.

En el cas del ratolí teníem funcions específiques (per exemple, *mousePressed*) que feien que no fos necessari utilitzar aquesta funció, però en el cas de *controlP5* aquesta serà la millor manera de gestionar el que està passant.

2) Comprovació *if(theEvent.isController())*: amb aquesta comprovació filtrem tots els esdeveniments que arriben i només gestionem els que són de tipus *controller*, que són els associats als elements de la nostra interfície.

3) Informació que rebem: un cop que sabem que l'esdeveniment és de la interfície, ensenyem la informació del nom i el valor que té a la finestra de missatges del Processing. Això ho fem simplement per a poder fer un seguiment del que està passant mentre fem les proves.

4) Comprovar qui ha generat l'event: aquesta és la part important. Amb la comprovació *if(theEvent.controller().name().equals("button1"))* mirem quin dels elements que hem creat ha generat l'esdeveniment. En el nostre cas només tenim el botó, de manera que comprovem si qui ha generat l'esdeveniment es diu *button1*.

Si tinguéssim més elements afegiríem tants blocs de comprovació *if(theEvent.controller().name().equals("nom_element"))* com fossin necessaris.

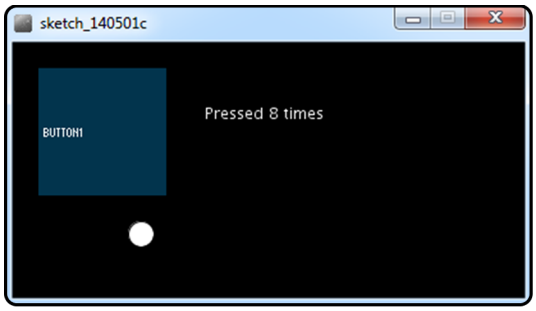
5) Acció per fer: dins de cada bloc associat a cada element hi haurà el codi que s'ha d'executar en utilitzar aquest element; en el nostre cas escrivim un missatge de text dient que hem premut el botó. Si el codi que s'ha d'executar és una mica complex serà molt recomanable crear una funció a part i cridar-la des d'aquí per simplificar la comprensió del codi i tenir-lo més ben organitzat.

Ara que tenim el botó creat i sabem quan el premem, afegim una mica de codi perquè passi alguna cosa més interessant:

```
import controlP5.*;

ControlP5 userinterface;

int ball_x;
int count;

void setup()
{
   size(400, 200);

   userinterface = new ControlP5 (this);
```

```
    userinterface.addButton ("button1", 1, 20, 20, 100, 100);

    ball_x = 20;
    count = 0;
}

void draw()
{
    background (0, 0, 0);

    text ("Pressed "+count+" times", 150, 60);
    ellipse (ball_x, 150, 20, 20);
}

void controlEvent(ControlEvent theEvent) {
    if(theEvent.isController()) {
        println("control event from : "+theEvent.controller().name()+", value :
                                        "+theEvent.controller().value());

        if(theEvent.controller().name().equals("button1")) {
            println ("BUTTON PRESSED");
            pressedButton1();
        }
    }
}

void pressedButton1(){
    ball_x = ball_x + 10;
    count = count + 1;
}
```

Figura 72. *ControlP5*: botó funcional



Aquest és un exemple molt bàsic en què podem veure l'estructura que utilitzarem per a treballar amb aquesta biblioteca. Tenint en compte que l'objectiu del *controlP5* és simplificar la creació d'interfícies, ens pot semblar que el codi que hem hagut d'utilitzar és força complex, i és cert, però és molt més senzill que haver de crear i gestionar aquests elements gràfics nosaltres mateixos.

El codi que hem creat utilitza les opcions més bàsiques d'un botó per a veure la interacció més bàsica que podem obtenir, i a partir d'aquí és molt interessant veure els exemples disponibles per a poder treure el màxim profit a aquesta biblioteca.

Per exemple, aquí teniu l'aspecte d'una pràctica feta amb el Processing per un company que ha cursat l'assignatura *Integració digital de continguts*, en què cada un dels elements de control s'ha elaborat amb aquesta biblioteca:

Figura 73. *ControlP5*: exemple pràctic