INTRODUCTION À PROCESSING

2013-2014 | berenger.recoules@gmail.com

Contents

Introduction 5

Premier Programme 8

Les variables 10

définition 10

Différents types de variables : 10

La portée des variables 11

Exemple d'utilisation des variables 11

Variables globales de processing 12

L'aléatoire 13

random() 13

noise() 14

randomSeed() et noiseSeed() 15

Les boucles 17

for() 17

while() 17

Couleurs 19

Couleurs 19

Niveau de gris 19

Mode RGB 19

Mode HSB 19

La transparence 19

Primitives de dessin 21

Les instructions de dessin 21

Les primitves (formes) 21

Les vertices et coordonnées polaires (formes sur mesure) 21

Transformation de l'espace 25

translate() 25

rotate() ; 26

Coder ses propres fonctions 28

Interactions Souris et clavier 30

Souris 30

Les variables globales 30

Les fonctions 31

La fonction map() 33
Clavier 33
Dessiner du texte, utiliser des polices de caractère 34
Les Classes : Programmation Orientée Objet 36
Structure d'un classe 36
Construction d'un classe simple 36
Déclaration de variables 36
Constructeur : initialisation 37
Méthodes complémentaires : update() et draw() 37
Utilisation d'un classe simple 38
Les Tableaux 40
Emergence : Un programme interactif complexe 42
Les Librairies 44
Installation d'une librairie 44
ControlP5 pour la création de GUI 44
OSCP5 pour la communication avec d'autres programmes 47
Travailler avec les images 50
Charger et afficher une image 50
Acccéder aux pixels 51
Faire « sortir » les pixels en 3D 51
3D et audio-réactif avec Pure-Data 53
Trucs et astuces 57
IDE 57
Programmation 57
abréviation des opérations 57
Graphisme 57
Un blur très simple 57
Color Selector 57
In/Out 57
Sauvegarder une image 57
Redimensionner une image 58
Ressources 59

Introduction

Processing est un langage de programmation basé sur java et principalement destiné à la création graphique. Il est apparut en 2001, crée par deux artistes Ben Fry et Casey Reas, alors étudiants au MIT. Il reprend une partie des concepts de Design by Numbers l'environnement de programmation graphique developpé par John Maeda au sein du Média Lab du même MIT.

Le langage processing est du Java fortement simplifié par l'accès direct à de nombreuses primitives de dessin. Il a été crée spécialement dans le but de faciliter l'apprentissage des bases de la programmation objet, via la création graphique permettant ainsi d'obtenir des résultats valorisants très rapidement.

Processing seeks to ruin the careers of talented designers by tempting them away from their usual tools and into the world of programming and computation. Similarly, the project is designed to turn engineers and computer scientists to less gainful employment as artists and designers.

Au-delà du fait d'être un formidable outil de dessin et de prototypage rapide en terme de design d'interaction. Processing est un réel langage de programmation capable de réaliser n'importe quelle fonction.

Processing est un projet Open Source devenu collaboratif de par l'intérêt qu'il a suscité dès sa sortie. Il est ouvert au développements tiers par l'intégration d'un système de librairie ainsi qu'une documentation aidant au développement de librairies externes. Il existe aussi un système de « modes » permettant d'ajouter des fonction à l'IDE (Itegrated Developpement Environment) de processing comme le développement pour android (smartphones et tablettes), ou en javascript (web).

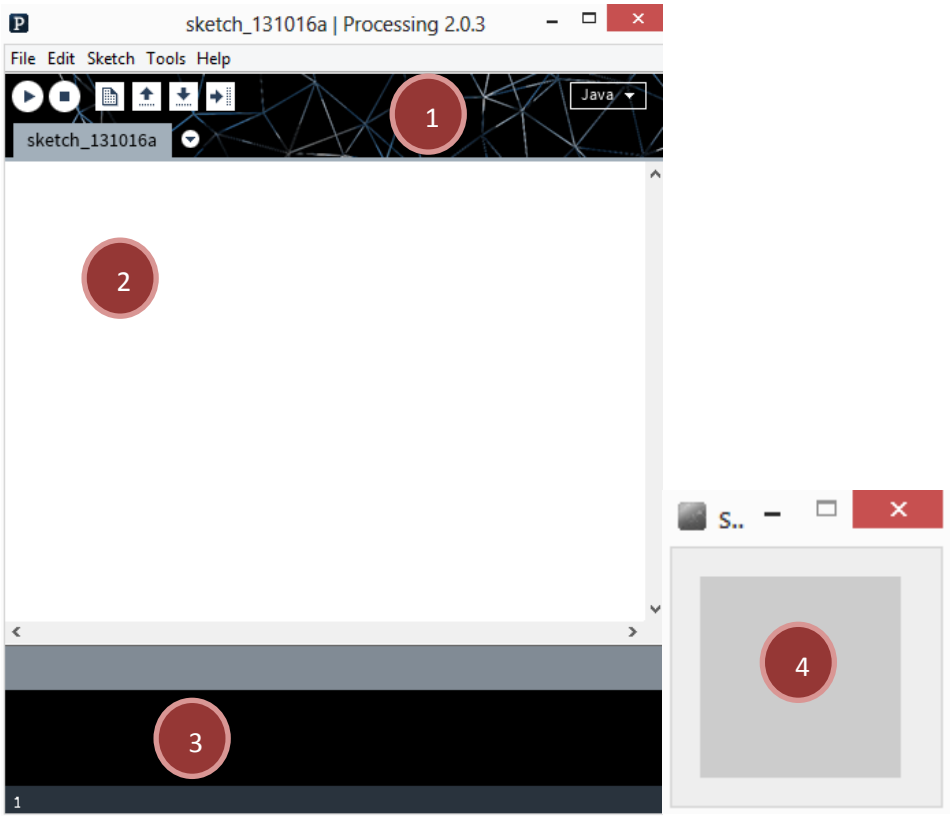
Processing existe aussi bien sous Linux, Windows que Mac OS. Le projet étant à but pédagogique et porté par une communauté très active il est excessivement bien documenté.

Note : Ce document ne se veut pas exhaustif, il présente les fonctions principales dont nous aurons l'utilité dans le cadre de ce cours. Processing étant très bien documenté il est très facile de trouver de nombreux tutoriels en ligne sur les différents aspects du programme. Se référer à la section « Ressources » pour plus d'informations.

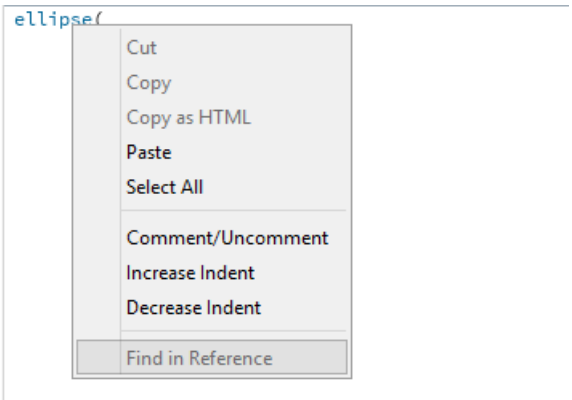
Note2 : Ce document a pour but de rappeler les informations vues en cours sous une autre formalisation, sa lecture doit donc s'accompagner des exemples de code construits pendant le module.

L'IDE de processing

L'ide est le programme que l'on lance pour écrire du code processing, il comporte différentes zones :



- La zone 1 correspond à la barre d'action, en haut à gauche se situe différents boutons : le premier « run » permet de lancer son programme, le second « stop » permet de le stopper. En dessous se trouve un système d'onglet, cela permet de mieux organiser son code quand les programmes deviennent plus complexes. Tout à droite se trouve un menu déroulant permettant de passer d'un mode à un autre (c'est-à-dire d'un developpement android à un developpement java classique par exemple).
- La zone 2 est un éditeur de texte classique permettant d'écrire son programme. Les mots clés du langage processing y apparaitront en surbrillance. Il est important de noter qu'à tout moment il est possible de consulter la documentation en ligne en effectuant un clic droit sur un mot clé et en sélectionnant « find in reference ».



- La zone 3 est la console qui renvoi les erreurs rencontrées par l'ordinateur lorsqu'il tente d'exécuter un programme. C'est aussi une zone d'information dans laquelle on peut choisir d'afficher des messages (à l'aide de la fonction println()).
- La zone 4 est la fenêtre d'exécution de notre programme.

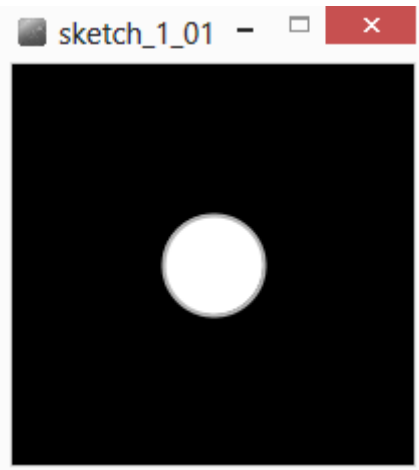
Processing fonctionne par défaut avec l'utilisation d'un « sketchbook ». C'est un dossier sur votre disque dur dans lequel seront stockés vos programmes et toutes les librairies, ainsi que les diférents modes que vous avez installé. Il est possible de modifier l'emplacement de ce dossier en allant dans :

File -> Preferences

Premier Programme

A tout moment il est possible d'insérer un commentaire dans son programme en utilisant ces deux caractères en début de ligne « // ». On peut aussi créer des commentaires sur plusieurs lignes en utilisant ces caractères : « /* » pour signaler le début d'un commentaire, et « */ » pour signaler la fin de ce même commentaire.

```
01.     // voici mon premier programme
02.     void setup(){
03.     size(200,200) ; // permet de spécifier la taille de la fenêtre.
04.     background(0) ; // utilisons un fond noir.
05.     }
06.     void draw(){
07.     background(0) ; // dessinons un fond noir.
08.     stroke(180) ; // on choisit de  dessiner un contour gris.
09.     strokeWeight(2) ; // ce contour aura un épaisseur de 2 pixels.
10.     fill(255) ; // le remplissage de notre dessin sera blanc.
        /* on dessine une ellipse située à 100 pixels du bord gauche, à
        100 pixels du bord haut, d'une largeur de 50 pixels et d'une
        hauteur de 50 pixels … un cercle donc ! */
11.     ellipse(100,100,50,50) ;
12.     }
```



Sketch_1_01.pde

Ce programme se compose de deux parties principales, appelées aussi « fonctions » qui se démarquent par l'utilisation de mots clés « void » ainsi que la paire d'accolade { } qui délimite les instructions exécutées lors de l'appel de la fonction.

- Les lignes 02 à 05 : présentent la fonction setup(). Cette fonction est appelée une seule fois au démarrage du programme, c'est un initialisation.

- Les lignes 06 à 12 : présentent la fonction draw(), qui est le cœur du programme. La suite d'instruction enfermée entre les accolades est exécutée en boucle, le plus rapidement possible.

Il existe des fonction spécifique au langage processing appelée primitives, on peut les appeler simplement en utilisant leur syntaxe spécifique. Chaque fonction indiquée en surbrillance possède une documentation en ligne, il est fortement conseillé de s'y référer pour savoir comment les utiliser.

A noter que par défaut nous utilisons un systèmes de coordonnées cartésiennes centré en haut à gauche de la fenêtre de dessin. Dans notre programme, le coin en haut à gauche a donc les coordonnées(0,0), le coin en bas à droite a donc les coordonnées (199,199)

Les variables

définition

Les variables correspondent à un espace utilisé dans la mémoire de l'ordinateur pour stocker une information de manière temporaire. Les variables peuvent être de différents types en fonction des données qu'elles doivent stocker.

Différents types de variables :

- int : permet de stocker des nombres entiers.
- float : permet de stocker des nombres flottants soit des nombres à virgules.
- string : permet de stocker des chaines de caractères, c'est-à-dire du texte.
- color : permet de stocker une couleur.

Si on écrit :

```
01. int a;
02. a = 5;
03. int b ;
04. b =3;
05. int result = a + b;
06. String operation = a +"+"+ b +"=";
07. println(operation);
08. println(result);
```

On crée une variable entière dont le nom est a, et on lui attribue la valeur 5. On crée une seconde variable entière dont le nom est b et on lui attribue la valeur 3. On crée ensuite un entier pour stocker le résultat que l'on obtient en additionnant les deux variables.

On crée ensuite une variable de type chaine de caractère pour inscrire l'opération effectuée puis son résultat dans la console. Une chaine de caractère doit être comprise entre deux " " pour être reconnue comme telle. Ici on compose une chaine de caractère complexe en utilisant le symbole « + », les différentes chaines de caractères (ou caractères simples) qui la compose sont accolées.

Il est important de noter que si jamais notre variable appelée « result » avait été de type String, la ligne :

03. **String** result = a+b ;

aurait renvoyé un tout autre résultat puisque les variables aurait été interprétée comme des String on aurait alors obtenu le juxtaposition des deux caractère soit « 53 ».

Certain type sont compatibles avec d'autres : on peut par exemple stocker un entier dans un flottant et des entiers ou des flottants dans des String. Cependant l'utilisation d'un flottant avec un type entier reverra nécessairement une erreur dans la console.

Il est aussi possible de stocker des données plus complexes comme des tableaux. Les tableaux servent à stocker des ensembles de données d'un type précis, on peut même y stocker des instances de classes… (nous verrons ceci dans la section dédiée aux classes).

La portée des variables

Un point essentiel réside dans la portée de ces variables (en anglais on parle de « scope »). D'une façon simplifiée : une variable sera accessible uniquement dans la fonction ou portion de code dans laquelle elle aura été définie. Une portion de code correspond à l'espace entre deux accolades « { } ».

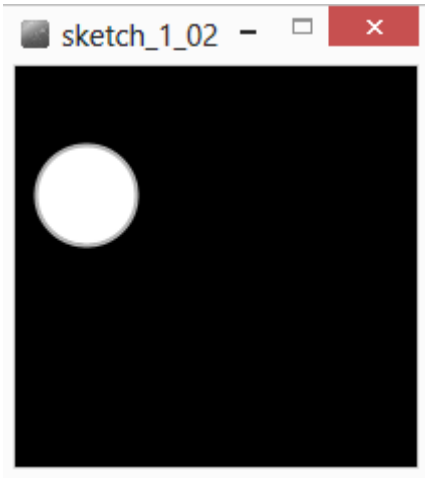
Cela signifie qui si je définie une variable dans le setup() de mon programme celle-ne sera accessible uniquement dans le setup().

Il est possible de définir des variables à l'extérieur des fonctions setup() et draw() : par exemple en tout début de programme, ces variables seront alors accessibles partout dans notre programme.

Exemple d'utilisation des variables

Habituellement on a tendance à créer les variables tout en haut de notre programme, on les initialise ensuite dans le setup() , puis on les utilise dans le draw() .

```
01.     // voici mon premier programme utilisant des variables
02.     color background_color ;
03.     int size ;
04.     float xpos,ypos ;
05.
06.     void setup(){
07.     size(200,200) ;
08.     background_color = color(0) ;
09.     size = 50 ;
10.     xpos = random(100) ;
11.     ypos = random(100) ;
12.     background(background_color ) ; // utilisons un fond noir.
13.     }
14.
15.     void draw(){
16.     background(background_color ) ;
17.     stroke(180) ;
18.     strokeWeight(2) ;
19.     fill(255) ;
        /* on dessine notre ellipse en utilisant nos variables*/
20.     ellipse(xpos,ypos,size,size) ;
21.     }
```



Sketch_1_02.pde

Variables globales de processing

Il existe dans processing des variables globales, qui sont donc accessibles partout dans processing, ces variables sont définies par défaut et gérée par processing lui-même, il faut mieux éviter d'utiliser leur nom pour définir ses propres variables.

C'est le cas entre autres de :

- width : (float) qui est associée par défaut à largeur de la fenêtre de dessin.
- height : (float)qui est associée par défaut à la hauteur de la fenêtre de dessin.

L'aléatoire

En informatique et en design génératif , l'aléatoire est très souvent utilisé pour obtenir des résultats présentant des variantes contraintes, c'est-à-dire pour obtenir plusieurs variations d'un même algorithme.

Il existe deux principales façon d'obtenir des nombres aléatoires ou plutôt pseudo-aléatoire, car il n'existe pas en informatique de méthode permettant d'obtenir un résultat réellement et statistiquement complètement aléatoire.

random()

La fonction random() renvoit donc des résultats aléatoires en fonction d'un argument qui sera spécifié entre les parenthèses. Ce nombre aléatoire sera de type float.

Par exemple :

```
01.     float nb_aleatoire = random(100) ;
02.     println(nb_aleatoire) ;
```

nous obtiendrons avec ce code un nombre aléatoire compris entre 0 et 100.

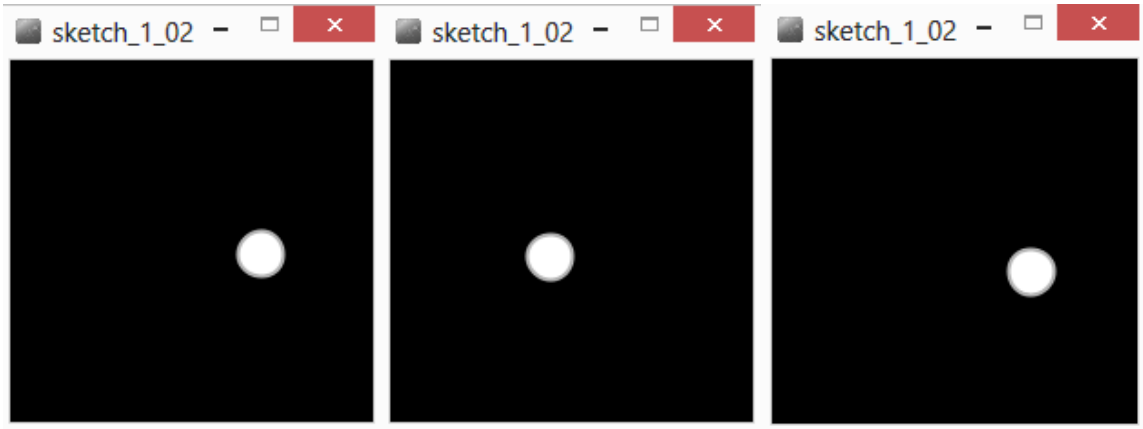
Il est aussi possible de spécifier un borne supérieur, ainsi qu'un borne inférieure, ainsi :

```
01.     float nb_aleatoire = random(20,50) ;
02.     println(nb_aleatoire) ;
```

renverra une valeur aléatoire comprise entre 20 et 50 ;

```
01.     /* voici mon premier programme utilisant des variables et de
   l'aléatoire */
02.     int size ;
03.     float xpos,ypos ;
04.
05.     void setup(){
06.     size(200,200) ;
07.     size = 25 ;
08.     xpos = random(0,width) ;
09.     ypos = random(0,height) ;
10.     background(0 ) ; // utilisons un fond noir.
11.     }
12.
13.     void draw(){
14.     background(0) ;
15.     stroke(180) ;
16.     strokeWeight(2) ;
17.     fill(255) ;
18.     // à chaque image calculée on définit une nouvelle position
19.     xpos = random(0,width) ;
20.     ypos = random(0,height) ;
        /* on dessine notre ellipse en utilisant nos variables*/
21.     ellipse(xpos,ypos,size,size) ;
22.     }
```

Ce programme va dessiner, à chaque image, un cercle positionné aléatoirement dans la fenêtre de dessin.



Sketch_1_03.pde

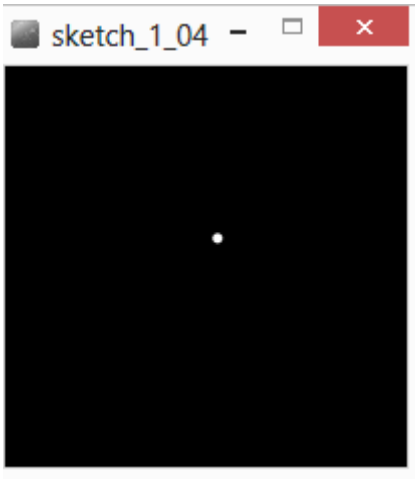
noise()

La fonction noise() est un peu particulière puisque elle permet de générer des suites de nombres très proches les uns des autres. Cela permet notament de créer des mouvement et des contours qui paraissent plus naturels.

Son usage est un peu plus compliqué car il faut lui fournir un argument « évolutif », celle-ci renvoi des valeurs comprises entre 0 et 1. Il faut donc souvent adapter le résultat obtenu en fonction de nos besoins.

```
01.      float noiseF; // facteur évolutif de notre fonction noise
02.      float xpos,ypos; // coordonnées de notre forme
03.
04.      void setup(){
05.        size(200,200);
06.        background(0);
07.        noStroke();
08.        fill(255);
09.        // on initialise notre facteur à une valeur aléatoire
10.        noiseF = random(500) ;
11.        // on place notre forme au centre de la fenetre.
12.        xpos = width/2;
13.        ypos = height/2;
14.      }
15.
16.      void draw(){
17.        background(0);
18.        /* on ajoute à notre position une valeur comprise entre
   -1 et 1, résultante de notre fonction noise. */
19.        xpos += noise(noiseF,10,20)*2-1;
20.        ypos += noise(noiseF,85,140)*2 -1;
```

```
21.         ellipse(xpos,ypos,5,5);
22.       // on incrémente notre facteur noise d'une petite valeur
23.         noiseF += 0.005;
24.       }
```



Sketch_1_04.pde

Ce programme dessine un cercle qui va se déplacer aléatoirement dans l'espace de dessin. Il est d'ailleurs fort probable qu'il en sorte, mais nous y reviendrons plus tard.

randomSeed() et noiseSeed()

Ces deux fonctions permettent de pouvoir retrouver un résultat qui a été obtenu avec des nombres aléatoires. Comme mentionné ci-dessus, les ordinateurs ne permettent pas d'avoir des générateurs de nombres complétement aléatoires, dans certain cas cela peut-être un avantage, notament quand il s'agit de pouvoir régénérer exactement la même image avec un algorithme qui utilise pourtant des nombres aléatoires.

Ces fonctions s'utilisent de la même façon :

```
01.       int seed ;
02.       seed = 123;



03.       randomSeed(seed);
04.       float a = random(500);
05.       println("seed" + seed +" : " + a);

06.       seed = 52;
07.       randomSeed(seed);
08.       a = random(500);
09.       println("seed" + seed +" : " + a);
10.
11.       seed = 123;
```

```
12.       randomSeed(seed);
13.       a = random(500);
14.       println("seed" + seed +" : " + a);
```

cf : Sketch_1_05.pde

Les boucles

C'est un des point primordiaux de la programmation objet, le boucles permette de répéter une action ou une suite d'instruction un nombre limité de fois. Il existe les boucles dites « for », et les boucles dites « while ».

Dans le cadre de ce document nous n'allons traiter que les boucles «for », les boucles « while » étant relativement rarement usitées.

for()

voici un exemple de boucle « for » :

```
01.      for (int i  = 0 ; i < 10 ; i=i+1){
02.            noStroke() ;
03.            fill(255) ;
04.            ellipse (10+i*10, 10+i*10, 5,5) ;
05.      }
```

Un boucle for se compose, de deux parties :

- Un bloc d'instructions à exécuter, situé entre les deux accolades.
- Entre les parenthèses ce sont les conditions d'exécution de la boucle qui sont séparées par des points virgules. D'abord, on définit un nombre entier appelé « i » que l'on initialise à 0, ensuite on précise que l'on exécutera le bloc d'instruction uniquement si « i » reste strictement inférieur à 10, puis on incrémente « i » de 1 en lui ajoutant la valeur 1.

Le code présenté ci-dessus permet donc de dessiner 10 cercles blancs de 5 pixels de diamètre, le premier étant situé en haut à gauche aux coordonnées (10,10), le dernier au coordonnées (100,100).(100 = 10 + 9*10).

Comme nous le verrons ces boucles sont très utilisées en programmation objet car elles permettent d'itérer une série d'instructions sur les éléments d'un tableau.

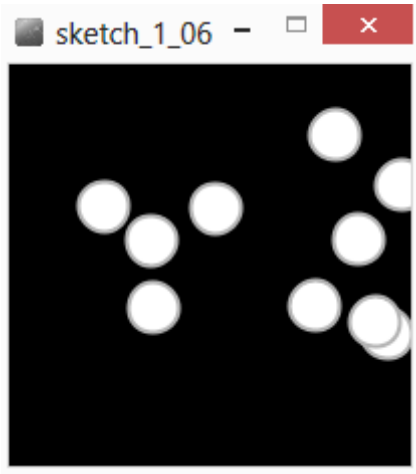
while()

La boucle while() est moins usitée en programmation objet, mais très commune en électronique, elle permet de réaliser un bloc d'instruction tant qu'une condition est vérifiée.

```
01.      int i =0 ;
02.      while (i < width){
03.            noStroke();
04.            fill(255);
05.            ellipse(i,height/2,5,5);
06.      i+=5;
07.      }
```

Ici, nous avons un variable initialisée à zéro, tant que cette variable est inférieure à la largeur de notre fenêtre de dessin, nous dessinons une ellipse blanche, de 5 pixels de diamètre placée en abscisses à la valeur de « i » et en ordonnées au milieu de notre fenêtre de dessin.

```
01.      /* voici mon premier programme utilisant des variables et de
   l'aléatoire et une boucle for*/
02.      int size ;
03.      float xpos,ypos ;
04.
05.      void setup(){
06.      size(200,200) ;
07.      size = 25 ;
08.      xpos = random(0,width) ;
09.      ypos = random(0,height) ;
10.      background(0 ) ; // utilisons un fond noir.
11.      }
12.
13.      void draw(){
14.      background(0) ;
15.      stroke(180) ;
16.      strokeWeight(2) ;
17.      fill(255) ;
18.      for (int i = 0 ; i <10 ; i++){
19.      // à chaque image calculée on définit une nouvelle position
20.      xpos = random(0,width) ;
21.      ypos = random(0,height) ;
22.      ellipse(xpos,ypos,size,size) ;
23.      }
24.      }
```



Sketch_1_06.pde

Couleurs

Couleurs

Niveau de gris

Dans processing il existe trois mode principaux de couleurs. Le premier mode est celui que nous avons déjà utilisé dans les exemples précédents : le mode grayscale ou niveau de gris. Dans ce mode il s'agit de signifier un nombre entre 0 et 255 ; 0 étant le noir et 255 le blanc.

```
01.      stroke(0) ;
02.      fill(180) ;
03.      ellipse(width/2,height/2,50,50) ;
```

Le code va donc dessiner un cercle gris souris avec un contour noir.

Mode RGB

Le mode de couleur par défaut de processing est le mode RGB (« Red Green Blue »), pour créer des couleurs il s'agit alors de spécifier les niveau de rouge, vert et de bleu que l'on souhaite entre 0 et 255 ;

```
01.      noStroke() ;
02.      fill(255,0,0) ; // Rouge
03.      ellipse(width/5,height/2,10,10) ;
04.      fill(0,255,0);// Vert
05.      ellipse(width*2/5,height/2,10,10);
06.      fill(0,0,255);// Bleu
07.      ellipse(width*3/5,height/2,10,10);
08.      fill(255,100,100);// Rose
09.      ellipse(width*4/5,height/2,10,10);
```

Mode HSB

Le mode HSB correspond à la specification de niveau de Hue (teinte), Saturation (contraste), Brightness (luminosité). Lorsque l'on définie le mode HSB on spécifie généralement l'étendue de la plage que doivent occuper ces valeurs.

```
01.      colorMode(HSB,360,100,100) ;
```

précise que l'on va spécifier les valeurs de teinte entre 0 et 360, les autres valeurs seron spécifiées entre 0 et 100.

Ce mode rend très facile la création de dégradés.

```
01.      size(360,100) ;
02.      colorMode(HSB,360,100,100) ;
03.      for (int i =0 ; i <width ; i++){
04.         stroke(i,100,100);
05.         line(i,0,i,height);
06.      }
```

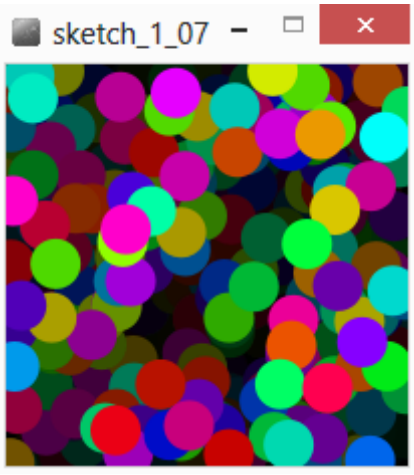
La transparence

Un des avantages de processing pour le dessin est le fait qu'il rende très simple l'usage de la trasparence. En effet il suffit d'ajouter un quatrième paramètre à nos couleurs quelque soit le mode

choisit. Ce dernier paramètre réglera le niveau de transparence à spécifier entre 0 et 255 ; 0 étant complétement transparent (soit invisible) et 255 étant entièrement opaque.

Dans la création graphique la superposition de plusieurs couches transparentes donne des effets de textures souvent intéressants. L'utilisation de la transparence dans certain cas facilite l'émergence de motifs complexes.

```
01.      /* voici mon premier programme utilisant des variables et de
   l'aléatoire et une boucle for et de couleurs !*/
02.      int size ;
03.      float xpos,ypos ;
04.
05.      void setup(){
06.      size(200,200) ;
07.      size = 25 ;
08.      xpos = random(0,width) ;
09.      ypos = random(0,height) ;
10.      colorMode(HSB,360,100,100) ;
11.      background(0 ) ;
12.      }
13.
14.      void draw(){
15.      // blur « maison »
16.      fill(0,20) ;
17.      rect(0,0,width,height) ;
18.      noStroke() ;
19.      for (int i = 0 ; i <10 ; i++){
20.      xpos = random(0,width) ;
21.      ypos = random(0,height) ;
22.      fill(random(360),100,100) ;
23.      ellipse(xpos,ypos,size,size) ;
24.      }
25.      }
```



Sketch_1_07.pde

Primitives de dessin

A partir de maintenant nous allons abandoner notre programme fil-rouge, qui nous a déjà appris beaucoup de choses, pour nous pencher vers d'autres rendus graphiques plus riches.

Les instructions de dessin

Nous avons déjà vu la majeure partie des instructions de dessins dans les programmes précèdents, mais opérons tout de même à un petit rappel :

- stroke() ; (color) permet de définir la couleur du trait de dessin.
- strokeWeigth() ; (float) permet de définir l'épaisseur de ce trait.
- noStroke() ; autorise à ne pas dessiner de contour.
- fill() ; (color) permet de définir la couleur de remplissage d'une forme.
- noFill() ; autorise à ne pas coloriser une forme.

Concenant les instruction de lignes, il existe aussi les fonction strokeCap() ;(String) et strokeJoint() ; (String) dont je vous invite à consulter la documentation en ligne.

Une autre instruction est importante , il s'agit de smooth() ; qui permet de modifier les paramètres de l'anti-aliasing de processing. On peut lui attribuer les valeurs de 2, 4 ou 8. Cela permet d'avoir des lignes fines plus précises à haute résolution.

Les primitves (formes)

Nous avons pour l'instant principalement utilisé des ellipses pour nos code. Mais processing regorge d'une bonne quantité de primtives pour dessiner différentes formes géométriques.

- ellipse(x-coord, y-coord, width,heigth) ; permet donc de dessiner une ellipse
- line(x1-coord, y1-coord, x2-coord, y2-coord) ; permet de dessiner une ligne entre les points (x1,y1) et (x2,y2) ;
- rect(x-coord,y-coord,width,height) ; permet de dessiner un rectangle, on peut lui adjoindre jusqu'à quatre autres paramètres pour spécifier l'arrondi de chaque angle.
- quad(x1, y1, x2, y2, x3, y3, x4, y4) ; permet de dessiner un quadrilatère.
- triangle(x1, y1, x2, y2, x3, y3) ; permet de spécifier un triangle.

Il existe souvent différent modes pour dessiner ces formes, je vous conseille donc de regarder les documentation de rectMode() (String) et ellipseMode() (String) par exemple. On peut par exemple choisir de dessiner à partir d'un coin (CORNER - par défaut lorsque l'on dessine un rectangle), ou à partir du centre de nôtre forme (CENTER – par défaut pour l'ellipse).

Les vertices et coordonnées polaires (formes sur mesure)

Si cela ne vous suffisait pas il existe d'autres possibilités pour créer des formes. Les fonctions beginShape(), endShape() et vertex() vont nous y aider.

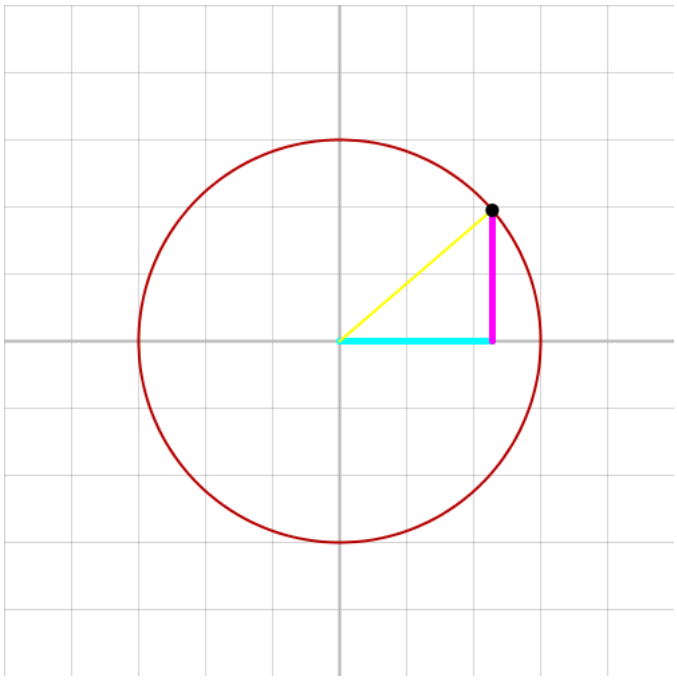
Un vertex n'est en fait ni plus ni moins qu'un couple de coordonnées, conjugé à beginShape() et enShape(), il permet de créer des ensembles de points à relier entre eux qui peuvent alors créer des

formes complexes. Nous allons nous intéresser à la façon dont il est possible de dessiner un cercle à l'aide de ces fonctions.

Pour rappel, voici un cercle trigonométrique :

Alors que les coordonnées cartésiennes utilisent l'abscisse et l'ordonnée d'un point pour le placer dans le plan, les coordonnées polaires utilisent le rayon et l'angle pour définir un point de l'espace. En terme de code informatique il est donc assez facilement imaginable de tracer un cercle à l'aide d'un boucle « for » permettant de parcourir les 360° (ou 2*PI pour ceux qui préfèrent les radians), cependant il nous faut un moyen de passer des coordonnées polaires aux coordonnées cartésiennes (processing et la fonction vertex() demandent en effet un couple de coordonées).

Le cercle trigo nous permet de nous souvenir simplement de ces formules : le point noir sur le cercle à pour coordonées (x,y) dans un repère cartésien et (r,theta) en coordonnées polaires.



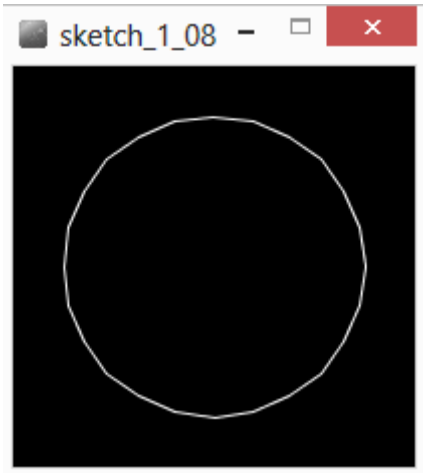
La projection bleue sur l'axe des abscisses nous donne la coordonnée x et correspond à un facteur près au cosinus de l'angle. La projection fuchsia sur l'axe des ordonnées nous fournit la coordonnée y qui est aussi proportionnelle à l'angle, mais cette fois au sinus :

X = rayon * cos (angle)
Y = rayon* sin(angle)

beginShape() va vous permettre de signaler à Processing que vous allez commencer à dessiner une forme, il attendra donc un instruction de fin de dessin qui sera endShape() et aussi une liste de vertex() entre les deux pour définir les différents points à relier entre eux.

```
01.      float xpos,ypos ;
02.      int radius; // le rayon de notre cercle
03.      float step; // l'incrément de l'angle qui va parcourir une
   rotation de 360°
04.
05.      void setup() {
```

```
06.         size(800, 600);
07.         background(0);
08.         xpos = width/2;
09.         ypos = height/2;
10.         radius = 300;
11.         step = PI/12;
12.       }
13.
14.       void draw() {
15.         background(0);
16.         noFill();
17.         stroke(255);
18.
19.         beginShape();// on démarre notre forme
20.         for ( float angle = 0 ; angle <TWO_PI ; angle += step) {
21.           // on applique la formule vue précédement
22.           float ex = xpos + radius *cos(angle);
23.           float wy = ypos + radius *sin(angle);
24.       // on ajoute un vertex
25.           vertex(ex, wy);
26.         }
27.         endShape(CLOSE);// on arête notre forme et on la ferme
28.
29.       }
```



Sketch_1_08.pde

Il est important de noter que ces objets sont très puissant et peuvent être manipulés avec différentes options pour obtenir des résultats divers.

Par exemple : beginShape(), peut prendre un argument qui affectera la façon dont la forme sera dessinée. beginsShape(POINTS) dessinera des points, beginShape(LINES) dessinera des lignes, beginShape(TRIANGLES) constituera des triangles etc.

Pour plus d'informations, il peut-ére utile de se référer à le documentation de Processing concernant beginShape(). Il peut-être intéressant aussi de regarder les fonction beginContour() et endContour

Concernant les vertices, il existe aussi plusieurs type de fonctions permettant de les utiliser : curveVertex(), bezierVertex(), quadraticVertex() sont d'autres façons de définir des vertices demandant plus ou moins d'arguments et donc plus ou moins simples à mettre en œuvre.

Enfin pour ceux qui persiste dans l'utilisation de processing, il est intéressant de regarder le foncionnement de l'objet PShape, qui permet de créer des formes complexes, de les stocker puis de les manipuler plus facilement.

Transformation de l'espace

C'est un des points primordiaux de processing, il faut savoir se repérer dans un espace 2D et savoir utiliser différents systèmes de coordonnées, pour pouvoir se faciliter la vie.

Il existe deux types de transformation de l'espace :

- translate() ;
- rotate() ;

translate()

translate() ; permet d'opérer une translation, ce qu'il est primordial de comprendre c'est que l'on n'opère pas cette translation sur les formes que l'on dessine, mais plutôt sur notre espace de dessin. C'est comme si l'on gardait notre crayon au même endroit et que l'on déplaçait la feuille.

Par exemple :

```
01.      ellipse(width/2,height/2,25,25) ;
02.      translate(50,0) ;
03.      ellipse(width/2,height/2,25,25) ;
```

dessinera deux cercles côte à côte séparé dont les centre seront séparés de 50 px. Ce programme est équivalent à : (c'est une question de style)

```
01.      translate(width/2,height/2) ;
02.      ellipse(0,0,25,25) ;
03.      translate(50,0) ;
04.      ellipse(0,0,25,25) ;
```

remarquez bien que les translations s'enchaînent, il est possible cependant de replacer la feuille à sa position par défaut en utilisant pushMatrix() et popMatrix(). Bien que cela puisse paraître compliqué il suffit de comprendre que si l'on dessine avec translate() il est parfois plus simple d'encadre chaque forme que l'on dessine de ces deux fonctions comme dans l'exemple ci-dessous :

```
01.      pushMatrix() ;
02.      translate(width/2,height/2) ;
03.      ellipse(0,0,25,25) ;
04.      popMatrix() ;
05.      pushMatrix() ;
06.      translate(width/2+50,height/2) ;
07.      ellipse(0,0,25,25) ;
08.      popMatrix() ;
```

translate en réalité déplace notre repère de dessin, par défaut le centre de notre repère (le point de coordonnées (0,0)) est situé en haut à gauchede notre fenêtre. En utilisant translate(xpos,ypos) ; nous déplaçons ce centre au point de coordonnées (xpos,ypos).

Ré-écrivons notre programme fil-rouge avec des tranlate().

```
01.      /* voici mon premier programme utilisant des variables et de
         l'aléatoire et une boucle for et de couleurs ! et des translate(),
         pushMatrix(), popMatrix()*/
02.      int size ;
03.      float xpos,ypos ;
```

```
04.
05.       void setup(){
06.       size(200,200) ;
07.       size = 25 ;
08.       xpos = random(0,width) ;
09.       ypos = random(0,height) ;
10.       colorMode(HSB,360,100,100) ;
11.       background(0 ) ;
12.       }
13.
14.       void draw(){
15.       // blur « maison »
16.       fill(0,20) ;
17.       rect(0,0,width,height) ;
18.       noStroke() ;
19.       for (int i = 0 ; i <10 ; i++){
20.       xpos = random(0,width) ;
21.       ypos = random(0,height) ;
22.       fill(random(360),100,100) ;
23.       pushMatrix() ; // on replace la feuille à chaque image
24.       translate(xpos,ypos) ;// on se déplace
25.       ellipse(0,0,size,size) ; // on dessine
26.       popMatrix() ; // objet parent de pushMatrix
27.       }
28.       }
```

popMatrix() permet en fait de replace la feuille pour dessiner éventuellement d'autres choses après. Ces deux objets doivent impérativement être utilisés conjointement, l'un sans l'autre renverra une erreur…

rotate() ;

rotate() fonctionne de la même façon que translate(), il faut lui fournir un angle en radians, une fonction radians(angle) permet de convertir un angle spécifié en degrés en une mesure radian facilement.

Il existe des variante de rotate que sont rotateX(), rotateY(), rotateZ() sur lesquelles nous ne épancheront pas.

En utilisant rotate(), il est nécessaire encore une fois de bien pense au système de coordonnées. Si par exemple je dessine un cercle au mileu tout en haut de ma zone de dessin :

```
01.       ellipse(width/2,0,50,50) ;
```

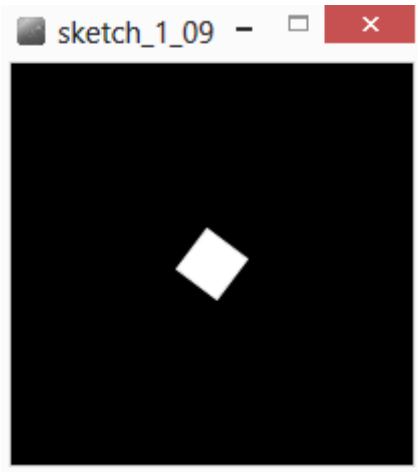
je peux faire en sorte que ce cercle se retrouve au centre en opérant un rotation de 45° ou PI/4 radians :

```
01. rotate(radians(45)) ;
02. ellipse(width/2,0,50,50) ;
```

Cela commence intéressant lorsque l'on combine translate() et rotate() ensemble, je peux par exemple très facilement faire tourner un carré sur lui-même.

```
01.       float angle ;
```

```
02.     void setup(){
03.       angle = 0 ;
04.       rectMode(CENTER) ;// rappelez vous les modes !
05.     }
06.     void draw() {
07.       background(0) ;
08.       stroke(255) ;
09.       pushMatrix();// on s'assure d'avoir un repère bien à nous
10.       translate(width/2,height/2) ;// on déplace notre repère
11.       rotate(angle);// on le fait tourner
12.       rect(0,0,25,25);// on dessine un rectangle
13.       popMatrix();
14.       angle += PI/24;// on incrémente l'angle
15.     }
```



Sketch_1_09.pde

Si vous intervertissez rotate() et translate() l'effet ne sera plus du tout le même, idem si l'on oublie d'utiliser pushMatrix() et popMatrix().

Coder ses propres fonctions

A partir de ces transformations simples nous allons créer une texture mouvante en quelques lignes de code. Le principe est simple nous allons utiliser le principe du carré tournant autour de son point supérieur gauche, mais nous allons créer une grille de carrés sur toute la surface de l'écran.

Pour simplifier le code nous allons écrire une fonction. Cette fonction aura pour objectif de dessiner un carré situé à des coordonnées spécifiques avec un angle de rotation propre. Cette fonction étant une fonction de dessin, elle sera de type « void », et elle acceptera les 3 paramètres sus-cités , on la déclarera de cette façon :

```
01.      void draw_rect(float xpos, float ypos, float rotation) {
02.      // suite d'instructions à écrire
03.      }
```

Pour utiliser cette fonction il suffit alors de l' « appeler » :

```
01.      draw_rect(50,50,PI/2) ;
```

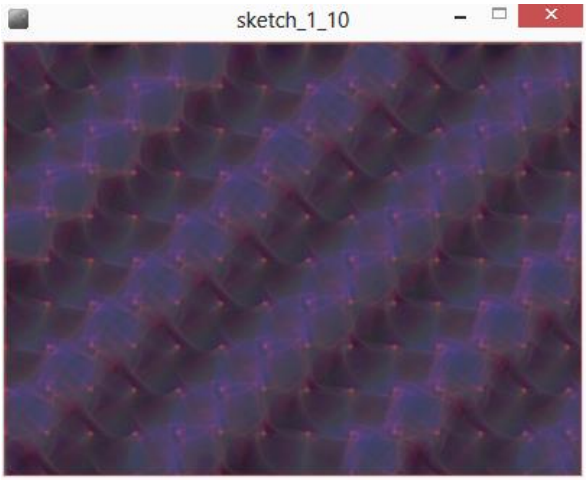
On dessinera ainsi un rectangle au point de coordonnées (50,50), tourné d'un angle de PI/2 radians, si le code écrit à l'intérieur de notre fonction est le bon. Heureusement nous avons appris à précédemment à faire exactement cela :

```
01.   void draw_rect(float xpos, float ypos, float rotation) {
02.   pushMatrix();
03.   fill(100,100,255,2);
04.   stroke(255,100,100,5);
05.   strokeWeight(2);
06.   translate(xpos, ypos);
07.   rotate(rotation);
08.   rect(0, 0, 35, 35);
09.   popMatrix();
10.   }
```

Nous avons maintenant ce qu'il nous faut, il ne nous reste maintenant plus qu'à écrire notre code autour… Pour animer notre image, nous allons utiliser une variable pour stocker l'angle auquel seront dessinés nos carrés, à chaque image (donc à chaque répétition de la boucle draw) nous allons augmenter la valeur de cet angle. Une petite astuce pour obtenir un rendu plus « ondulant » sera d'attribuer des valeurs d'angles différentes en fonction de la position du carré dans la grille !

```
01.      float angle = 0; // un variable pour stocker un angle
02.
03.      void setup() {
04.        size(400, 300);
05.        background(0);
06.      }
07.
08.      void draw() {
09.        fill(0,5);
10.        rect(0,0,width,height); // encore un blur
11.
12.        // une double boucle pour parcourir une grille
13.        // on crée des case de 30px X 30 px
14.        for (int i= 0 ; i <= width+30 ; i+=30) {
15.          for (int j=0; j <= height; j+=30){
```

```
16.           // on dessine un rectangle aux coordonnées (i,j)
17.           draw_rect(i, j, angle + i + j);
18.           }
19.         }
20.         // on incrémente notre angle
21.         angle += 0.05;
22.       }
23.
24.       void draw_rect(float xpos, float ypos, float rotation) {
25.         pushMatrix();
26.         fill(100,100,255,2);
27.         stroke(255,100,100,5);
28.         strokeWeight(2);
29.         translate(xpos, ypos);
30.         rotate(rotation);
31.         rect(0, 0, 35, 35);
32.         popMatrix();
33.       }
```



Sketch_1_10.pde

Interactions Souris et clavier

Processing nous donne accès à des fonctions bien pratiques pour créer de l'interaction avec nos programmes, notament à travers l'utilisation de fonction spécifiques permettant d'intercepter les événements provenant de notre souris ou de notre clavier.

Souris

Les évenements provenant de la souris peuvent être captés de diverses façon. Il est par exemple possible de connaitre la position de la souris à tout moment, de savoir quel bouton est activé etc.

Les variables globales

Processing met à notre disposition différentes variables globales nous permettant de connaitre l'état de notre souris, ainsi :

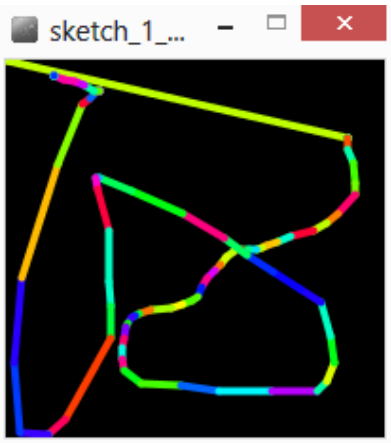
- mouseX et mouseY , nous permettent de connaitre les coordonnées de la position de la souris dans notre fenêtre à tout moment.
- mousePressed : nous renvoi un booléen : TRUE si la souris est cliquée, FALSE sinon.
- mouseButton : nous permet de connaitre l'identité du bouton qui a été cliqué

```
01.      void setup() {
02.      }
03.
04.      void draw() {
05.        println("mouse position :", mouseX, mouseY);
06.        println("mouse pressed :", mousePressed);
07.
08.        if (mousePressed && mouseButton==LEFT) {
09.          println("bouton gauche cliqué");
10.        }
11.        else if (mousePressed && mouseButton == RIGHT) {
12.          println("bouton droit cliqué");
13.        }
14.      }
```

Sketch_1_11.pde

Il existe aussi les variables pmouseX et pmouseY qui permette de connaitre la position de la souris à l'image précédente., combinées à mouseX et mouseY, il devient assez facile de calculer la vitesse de déplacement de la souris :

```
01.      void setup() {
02.        size(200, 200);
03.        background(0);
04.        strokeWeight(4);
05.        frameRate(15);
06.        colorMode(HSB, 360, 100, 100);
07.      }
08.      void draw() {
09.        noStroke();
10.        fill(0, 25);
11.        rect(0, 0, width, height);
12.        stroke(random(360), 100, 100);
13.        line(pmouseX, pmouseY, mouseX, mouseY);
14.      }
```



Sketch_1_12.pde

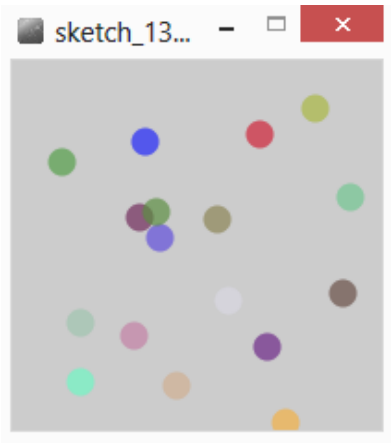
Les fonctions

Il existe aussi un certain nombre de fonctions permettant d'exectuer un bloc de code en fonction d'un événement souris :

- mousePressed() : lorsque l'on appuie sur un bouton.
- mouseReleased() : lorsque l'on relâche un bouton.
- mouseClicked() : lorsque l'on appuie puis que l'on relâche un bouton.
- mouseMoved() : lorsque l'on déplace la souris.
- mouseDragged() : lorsque l'on déplace la souris alors qu'un bouton est appuyé.
- mouseWheel() : lorsque l'on active la molette.

Toutes ces fonctions s'utilisent de la même façon :

```
01.      void setup() {
02.        size(200, 200);
03.      }
04.
05.      void draw() {
06.      }
07.
08.      void mousePressed() {
09.        noStroke();
10.        fill(random(255),random(255),random(255),random(100, 180));
11.        ellipse(mouseX, mouseY, 15, 15);
12.      }
```



sketch_1_13.pde

Un petit point cependant sur la fonction mouseWheel(),avec laquelle on peut obtenir le sens de déplacement de la molette, par le biais d'un MouseEvent.

```
01.      float hue=180;
02.
03.      void setup() {
04.        size(200, 200);
05.        colorMode(HSB,360,100,100);
06.      }
07.
08.      void draw() {
09.        background(hue,100,100);
10.      }
11.
12.      void mouseWheel(MouseEvent event) {
13.        float amt = event.getCount();
14.        hue +=amt;
15.      }
```

Sketch_1_14.pde

La fonction map()

La fonction map() peut s'avérer être très utile dans le cadre de divers programmes. Elle permet d'échelonner des valeurs d'un intervalle à un autre. Imaginons par exemple qu'en fonction de la position de la souris nous voulions changer la taille d'un cercle.

Nous voulons que lorsque la souris est tout à gauche de l'écran, notre cercle soit petit (disons un diamètre de 5) et que lorsque nous sommes tout à droite notre cercle soir gros (un diamètre de 100). Nous savons déjà que lorsque notre souris est tout à gauche la variable mouseX aura pour valeur 0, lorsqu'elle est tout à droite sa valeur est de « width » (la variable globale contenant la largeur de notre fenêtre de dessin). La fonction map nous permettra alors de transformer la position de notre souris (comprise entre 0 et width) en une valeur comprise entre 5 et 100.

```
01.   float diam ;
02.   diam = map(mouseX,0,width,5,100) ;
```

Clavier

De la même façon, les évènements claviers peuvent être interceptés. Il existes les variables :

- key
- keyCode

qui retournent les valeurs de la dernière touche de clavier enfoncée ; il existe aussi des fonctions similaires à celles de la souris :

- keyPressed()
- keyReleased()
- keyTyped()

keyTyped() va ignorer l'enfoncement des touches CTRL, ALT etc.

En fonction des machines, des systèmes d'exploitation ces fonctions et variables ne renvoient pas systématiquement les même valeurs. Attention donc pour le développement sur plusieurs plateformes.(Pour débugger il est conseillé d'avoir recours à des println pour vérifier les valeurs dans la console). Le chapitre suivant concernant le texte et les polices de caractère vous donnera un exemple d'utilisation de ces fonction.

Dessiner du texte, utiliser des polices de caractère

Le programme suivant va permettre de stocker les lettres tapées sur notre clavier dans une variable de type String et va dessiner ensuite ces lettres de manière aléatoire sur l'écran. La fonction keyPressed() va gérer l'ensemble des interactions avec le clavier.

Pour dessiner du texte nous utilisons à la ligne 22 la fonction :

```
01.      text (string, int, int) ;
```

permettant de dessiner une chaine de caractère en spécifiant les coordonnées auxquelles la dessiner. Le rendu du texte est affecté par la fonction fill() comme n'importe quelle forme géométrique, mais ne réponds pas à la fonction stroke().

Deux autres fonctions permettent de modifier l'apparence du rendu du texte :

```
01.      textSize(float) ;
```

utilisée à la ligne 20, permet d'en changer la taille, et

```
01.      textFont(PFont) ;
```

permet de changer la police à la ligne21. Le type PFont est un objet permettant de charger une police externe chargée dans un fichier « *.vlw ».

On utilise donc une variable « font » de type PFont pour charger nos polices à la volée en fonction des touches enfoncées.

Concernant les interaction clavier, on utilise une technique de « castage » pour forcer le type de donnée que l'on va obtenir de la variable key. A la ligne 24, on s'assure que notre variable k sera bien un caractère avant de l'insérer dans notre chaine de caractères principale, à la ligne 25 on s'assure d'obtenir un entier pour avoir l'identifiant de la touche. Ensuite à l'aide de tests, nous définissons les action à effectuer.

```
01.   String buff = "         ";
02.   PFont font;
03.
04.   void setup() {
05.     size(400, 300);
06.     background(255);
07.     colorMode(HSB);
08.     smooth();
09.     font =loadFont("Mosaicleaf-48.vlw");
10.   }
11.
12.   void draw() {
13.     fill (255, 35);
14.     rect (0, 0, width, height);
15.     char k = buff.charAt(floor(random(buff.length()-1)));
16.     fill(random(255), 255, 255);
17.     textSize(random(48, 100));
18.     textFont(font, 48);
19.     text(k, random(0, width), random(0, height));
```

```
20.   }
21.
22.   void keyPressed() {
23.
24.     char k = (char)key;
25.     int nkey = (int) key;
26.
27.
28.     if (nkey == 48) {
29.       font = loadFont("Mosaicleaf-48.vlw");
30.     }
31.     else if (nkey == 49) {
32.       font =loadFont("Bauhaus93-48.vlw");
33.     }
34.     else if (nkey == 50) {
35.       font =loadFont("HarlowSolid-48.vlw");
36.     }
37.     else if (nkey==51) {
38.       font =loadFont("Magneto-Bold-48.vlw");
39.     }
40.     else if (nkey == 51 || nkey ==52 || nkey ==53 ||nkey ==54
41.       || nkey ==55 ||nkey == 56 || nkey == 57) {
42.     }
43.     else if (key == BACKSPACE) {
44.       buff = "  ";
45.     }
46.     else {
47.       buff=k+buff;
48.     }
49.   }
```

Sketch_1_15.pde

Il existe un outil permettant de construire des fonts au format .vlw et donc utilizable dans processing à partir des polices installées sur le système. Il suffit de cliquer sur le menu « Tools -> Create Font ».

Les Classes : Programmation Orientée Objet

Les classes sont un des concepts centraux de JAVA, elles nous permettent de créer des objets ensuite manipulables par du code. Souvent une classe permet d'encapsuler un certain nombre de concepts ensembles d'un façon générique, et permet d'améliorer la lisibilité de notre code.

Cela peut paraître un peu barbare mais c'est en réalité relativement simple : un classe est la description théorique d'un objet. Par exemple dans ce chapitre nous allons créer un classe « Mover », cette classe permettra de créer un objet (représenté graphiquement par un cercle) qui se déplacera dans notre fenêtre de dessin et rebondira contre les bords.

Structure d'un classe

Sa structure ressemble furieusement à la structure d'un programme processing. D'abord nous déclarerons des variables qui pourront être utilisées dans le code de notre classe. Ensuite il nous faudra une fonction pour initialiser ces variables , jusque là nous appelions ça la fonction « setup() », dans un classe cette fonction s'appelle un constructeur. Après cela nous aurons une ribambelle de fonctions qui seont appelées à chaque image calculée, qui nous permettra soit de dessiner quelquechose soit de modéliser un comportement physique, biologique , une interaction avec l'utilisateur etc.

```
01.      class Mover {
02.      // déclaration de variables
03.      // constructeur
04.      Mover(){
05.      // intialisation des variables
06.      }
07.      void update(){
08.      // faire des calculs
09.      }
10.      void draw(){
11.      // dessiner quelquechose
12.      }
13.      }
```

Construction d'un classe simple

Déclaration de variables

Pour construire notre classe Mover, nous allons utiliser un nouveau type de variable, le PVector. Cette variable est bien sûr l'équivalent d'un vecteur mathématique, son utilisation nous simplifiera grandement la vie pour l'implémentation du comportement physique que nous souhaitons. (rappelons tout de même qu'un vecteur n'est ni plus ni moins qu'un couple de coordonées).

Mover aura donc besoin pour fonctionner de deux vecteurs : un vecteur définissant la position de notre objet, et un vecteur définissant sa vitesse. Nous n'aurons donc que deux variables à déclarer :

```
01.      PVector loc, vel;
```

Constructeur : initialisation

Pour initialiser ces variales nous allons utiliser le même type de technique que lorsque nous avions écrit des fonctions. Nous allons nous attacher à pouvoir passer des arguments à notre objet. Cela signifie que lors de la création de l'objet, nous devrons nous même spécifier certaines valeurs, qui seront propre à cet objet crée. La classe elle n'a que faire de ses valeurs, elle ne les manipule que comme des valeurs symboliques. Sans surprise le constructeur resemblera donc à ça :

```
01.      Mover(PVector loc, PVector vel) {
02.          this.loc = loc;
03.          this.vel = vel;
04.        }
```

Il est important de bien comprendre ici le fonctionnement du mot clé « this » qui peut créer une confusion.

Dans notre classe nous avons déclarer deux PVector : 'loc' et 'vel'. Nous avons fait de même entre les parenthèse de notre constructeur pour pouvoir passer des valeurs. Rappelez vous de la portée des variables (vue au tout début de ce document). Dans le cadre de notre constructeur nous avons donc deux fois, deux variables qui portent le même nom ; il faut donc impérativement être capable de les différencier.

Le « this » sert à cela. Lorsque vous êtes dans cette situation le fait d'utiliser « this. » permet de signifier à notre programme que l'on parle de la variable de la classe, celle qui a été déclarée avant notre fonction. Nous allons donc toujours avoir :

```
01.      this.maVariable = maVariable ;
```

Autrement dit on attribue à la variable qui est utilisée dans notre classe, la valeur que l'on spécifie en argument de notre fonction.

Méthodes complémentaires : update() et draw()

En physique, il existe un lien entre la position, la vitesse et l'acceleration. Si l'on dérive l'accéleration par rapport au temps on obtient la vitesse, si l'on dérive cette vitesse on obtient la position. Et inversement si on intégre la position par rapport au temps on obtient la vitesse et si on intègre la vitesse on obtient la position (vérifiez si vous ne me croyez pas !). Pour nous cela signifie que pour calculer la position de notre objet à l'image suivante , il suffit d'ajouter la vitesse à notre position actuelle ! un petit tour rapide sur la page d'aide de PVector nous apprend qu'il existe une méthode « add() » pour ajouter deux objets PVector. La fonction update() de notre classe contiendra donc très certainement cette ligne de code :

```
01.      loc.add(vel);
```

Il ne nous reste plus qu'à dessiner quelquechose … et ça nous savons déjà le faire.

```
01.      class Mover {
02.
03.        PVector loc, vel;
04.
05.        Mover(PVector loc, PVector vel) {
```

```
06.          this.loc = loc;
07.          this.vel = vel;
08.        }
09.
10.        void update() {
11.          loc.add(vel);
12.        }
13.
14.        void draw() {
15.          pushStyle();
16.          noStroke();
17.          fill(255, 100);
18.          ellipse(loc.x, loc.y, 25, 25);
19.        }
20.
21.      }
```

Voici donc notre classe quasi-complétée. Il nous reste à utiliser des tests pour savoir si nos objets sortent de l'écran ; si c'est le cas il faut qu'elle rebondisse !

```
01.      if (loc.x < 0 || loc.x> width) { // trop à gauche ou ('||')
   trop à droite
02.            vel.x = -vel.x; // on inverse sa vitesse en abscisse
03.          }
04.      // meme schema pour les collision en haut et en bas avec
   l'ordonnée
05.          if (loc.y<0 || loc.y > height) {
06.            vel.y = -vel.y;
07.          }
```

Utilisation d'un classe simple

Maintenant notre classe écrite nous allons pouvoir l'utiliser. Vous trouvez peut-être que pour l'intant c'est beaucoup de code pour pas grand-chose, mais la magie de la programmation objet va commencer à opérer.

Notre classe écrite nous pouvons maintenant créer des objets que nous pourrons manipuler. Nous pouvons par exemple maintenant déclarer un nouvel objet Mover comme n'importe quel autre type de processing (int, float, string…).

```
01.      Mover mov ;
```

Va déclarer un nouvel objet, pour l'instant notre objet n'existe cependant toujours pas, nous disons juste à notre ordinateur que nous allons le créer. Nous allons le créer dans le setup() de notre programme, mais nous avons d'abord besoin de créer deux PVector que nous passerons en argument : un pour la position de notre objet, un pour sa vitesse. Ensuite nous pourrons créer un nouvel objet de type Mover en utilisant la syntaxe :

mov = new Mover(monVecteur1, mon Vecteur2) ;

```
01.      void setup() {
02.
03.        size (800, 600);
04.
05.        PVector initLoc = new PVector(width/2, height/2);
06.        PVector initAcc = new PVector(1.05, -2.25);
```

```
07.         mov = new Mover (initLoc, initAcc);
08.       }
```

Remarquez, que la façon dont nous initialisons les PVector et notre objet Mover est identique. C'est normal puisque l'objet PVector est lui-même un classe, qui est déjà codée pour nous.

Notre objet est donc crée, il ne nous reste plus qu'à la manipuler et à l'afficher. Cette étape se passera donc dans le draw()

```
01.       void draw() {
02.         background(180); // fond noir
03.         mov.update(); // calculer la position
04.         mov.draw();  // afficher notre forme
05.       }
```

Sketch_2_01.pde

Les Tableaux

Les tableaux sont un type d'objets complexes, ils nous permettent de stocker un grand nombre d'éléments de n'importe quel type float, string … ou même une classe que nous venons de créer. C'est précisément ce que nous allons faire. Cela nous permettra de traiter un maximum d'objet avec un minimum de lignes de code.

Pour créer un tableau nous devons connaitre le nombre d'éléments que nous allons stocker, il nous faudra donc d'abord définir une variable pour définir la taille de notre tableau. Ensuite un tableau se crée en utilisant des crochets '[' et ']'. Il nous faut d'abord définir le type d'objets que va contenir le tableau, puis ouvrir et fermer des crochets pour signifier que c'est un tableau, donner un nom à cet objet puis l'initialiser à l'aide du mot clé « new ». Pour créer un tableau de 100 objets Move, il suffit donc d'écrire ceci :

```
01.      int num = 100 ; //
02.      Mover[] movs = new Mover[num];
```

Pour créer un tableau de 100 flottants :

```
01.      Float[] flottants = new float[100] ;
```

Maintenant que nous avons crée notre tableau, il nous faut initialiser les éléments qui le compose (dans le cas d'une classe) ou leur attribuer une valeur (dans le cas de flottants) pour cela nous allons utiliser une boucle for pour parcourir l'ensemble de ses éléments. Pour accéder à un élément précis, on utilise le nom du tableau et entre crochet l'index de l'élément auquel on veut accéder :

```
01.      flottants[5] = 10 ;
```

va attribuer la valeur 10 à l'index 5 de notre tableau de flottants.

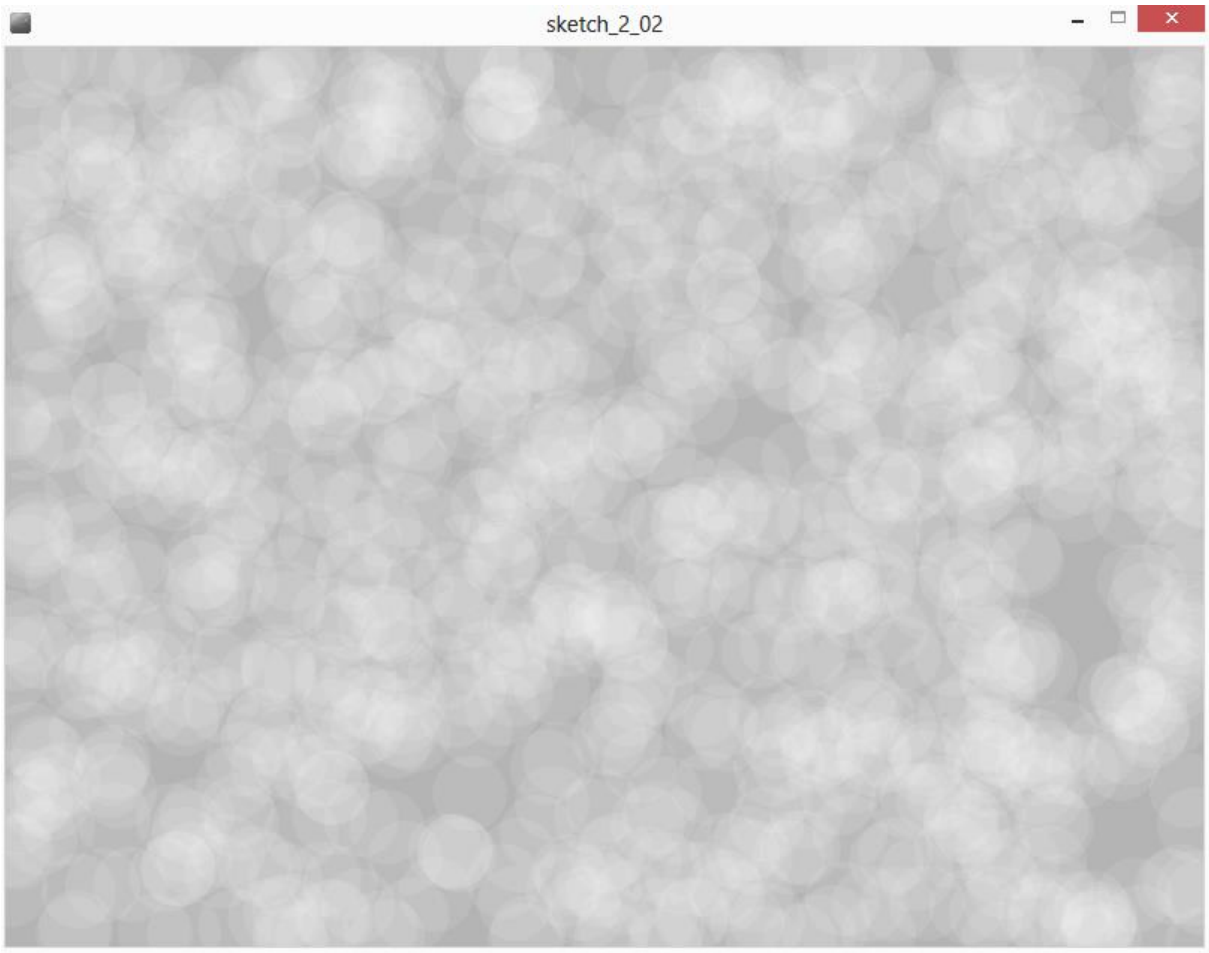
Pour initialiser notre tableau d'objets Mover , nous allons parcourir l'ensemble du tableau à l'aide d'une boucle, et à chaque index nous allons stocker un nouvel objet en appelant le constructeur de notre classe.

```
01.      void setup() {
02.        size (800, 600);
03.        for (int i = 0 ; i < num ; i++) {
04.          PVector initLoc = new PVector(random(5, width-5),
   random(5, height-5));
05.          PVector initVel = new PVector(random(-1,1), random(-
   1,1));
06.          movs[i] = new Mover (initLoc, initVel);
07.        }
08.      }
```

Enfin il ne nous reste plus qu'à utiliser nos objets dans le draw(), encore à l'aide d'un boucle for.

```
01.      void draw() {
02.
03.        background(180);
```

```
04.
05.
06.         for (int i = 0 ; i < num ; i++) {
07.           movs[i].update();
08.           movs[i].draw();
09.         }
10.
11.
12.       }
```



Sketch_2_02.pde

Nous avons donc maintenant 1000 objets Mover qui agissent indépendamment les uns des autres, et rebondissent sur les bords de notre fenêtre de dessin.

Emergence : Un programme interactif complexe

En repartant du programme précédent, nous allons nous attacher à représenter les choses d'un manière différente. Nous verrons qu'en changeant un peu de perspective nous arriverons à des résultats différents et graphiquement plus intéressants.

L'émergence en terme graphique peut être définie comme l'apparition de structures graphiques complexes et ordonnées à partir d'action simples.

Nous avons donc toujours un programme dessinant un certain nombre de particules qui se déplacent et rebondissent contre les bords de notre fenêtre de dessin. Au lieu de dessiner chaque particule individuellement, nous allons plutôt dessiner un lien entre deux de ces particules uniquement si la distance qui les sépare est inférieure à une certaine valeur.

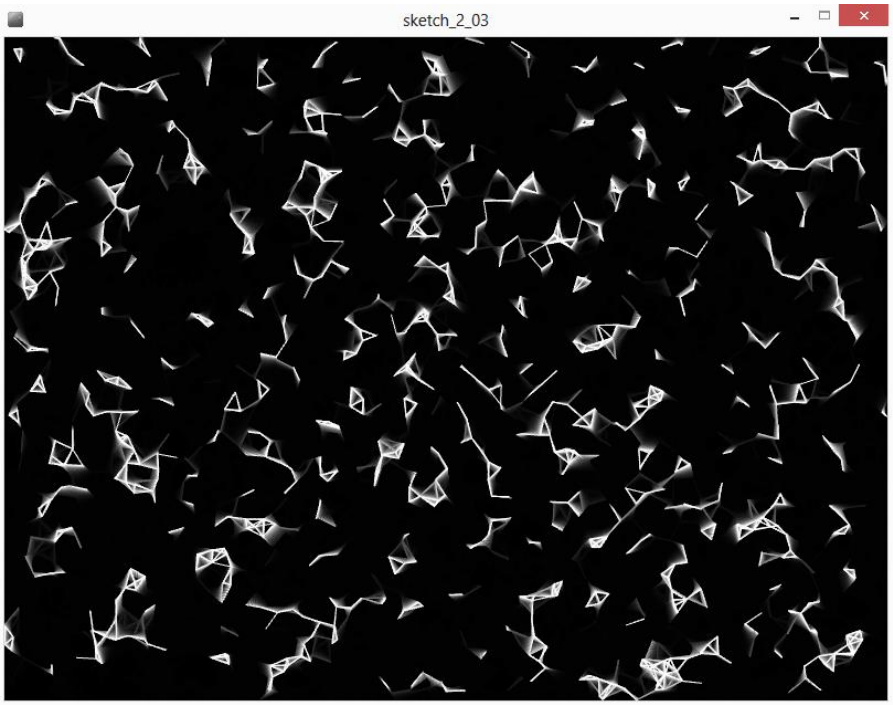
Cette valeur sera un variable nommée « treshold », elle sera ajusté en fonction de la position de la souris à l'aide de la fonction map() à la ligne 21.Nous n'utilisons plus la fonction draw() de notre classe Mover, elle a donc disparu.

Pour dessiner ce lien nous allons devoir utiliser une double boucle pour parcourir deux fois notre tableau de movers, en faisant attention à gérer le cas où l'on fait référence au même objet : à la ligne 27, nous vérifions que i est bien différent (« != ») de j.

Nous utilisons la fonction dist() pour calculer la distance entre les coordonnées des deux objets, puis si cette valeur est inférieur à notre treshold, nous dessinons une ligne utilisant ces mêmes coordonées aux lignes 28,29 et 30. Et voilà !

```
01.   int num = 1000;
02.   Mover[] movs = new Mover[num];
03.
04.   void setup() {
05.     background(0);
06.     size (800, 600, P2D);
07.     for (int i = 0 ; i < num ; i++) {
08.       PVector initLoc = new PVector(random(5, width-5), random(5,
   height-5));
09.       PVector initVel = new PVector(random(-1, 1), random(-1, 1));
10.       movs[i] = new Mover (initLoc, initVel);
11.     }
12.   }
13.
14.
15.   void draw() {
16.     noStroke();
17.     fill(0, 50);
18.     rect(0, 0, width, height);
19.
20.     stroke(255);
21.     float treshold = map(mouseX, 0, width, 0, 50);
22.
23.     for (int i = 0 ; i < num ; i++) {
24.       movs[i].update();
25.
26.       for (int j = 0 ; j < num ; j++) {
27.         if (i!=j) {
```

```
28.           float dist = dist(movs[i].loc.x, movs[i].loc.y,
   movs[j].loc.x, movs[j].loc.y);
29.           if (dist < treshold) {
30.             line(movs[i].loc.x, movs[i].loc.y, movs[j].loc.x,
   movs[j].loc.y);
31.           }
32.         }
33.       }
34.     }
35.   }
36.
37.   class Mover {
38.
39.     PVector loc, vel;
40.
41.     Mover(PVector loc, PVector vel) {
42.       this.loc = loc;
43.       this.vel = vel;
44.     }
45.
46.     void update() {
47.       loc.add(vel);
48.       if (loc.x < 0 || loc.x> width) {
49.         vel.x = -vel.x;
50.       }
51.       if (loc.y<0 || loc.y > height) {
52.         vel.y = -vel.y;
53.       }
54.     }
55.   }
```



Sketch_2_03.pde

Les Librairies

Un des grands avantages de Processing est sa vibrante communauté d'utilisateurs, qui écrit des tutoriels, qui documentent et partagent leurs travaux. Les développeurs de Processing ont voulu permettre aux utilisateurs développer leurs propres librairies et de les intégrer dans Processing.

Il existe un grand nombre de librairies pour faire beaucoup de choses différentes :

http://processing.org/reference/libraries/

Il en existe pour tout un tas d'applications : pour animer, pour contrôler de la vidéo, pour exporter ou importer des formats de données particuliers, pour faire de la 3d, de la typographie ou encore pour faire parler son ordinateur…

Nous allons nous intéresser principalement à deux librairies, une permettant de créer des boutons et des slider pour contrôler nos sketch processing : controlP5. L'autre permettant de faire communiquer deux programmes entre eux. Ces deux librairies ont été codées par Andreas Schlegel (http://www.sojamo.de/code/), un grand merci à lui !

Installation d'une librairie

Depuis la version 2.0 de processing, il existe un outil permettant d'installer facilement des librairies, c'est le « library manager » accessible depuis le menu « sketch -> Import library -> Add library ».

Cet utilitaire permet de naviguer parmi les librairie disponibles de les installer ou de les supprimer.

Si toute fois l'utilitaire ne fonctionnait pas bien, il existe un dossier spécifique dans notre sketchbook appelé « libraries » qui stocke toutes nos librairies installées. Pour rappel l'emplacement du sketchbook est modifiable dans « file -> Preferences ». Pour installer une librairie, il suffit de télécharger la librairie, de dézipper l'archive et de placer le dossier dans votre dossier « libraries »

Généralement une librairie est composée d'un dossier principal contenant quatre sous-dossier :

- /examples/
- /library/
- /reference/
- /src/

Dans le dossier /library/ vous devez normalement trouver un fichier *.jar portant le même nom que votre dossier racine. Si c'est bien le cas votre librairie sera alors reconnue et utilisable.

Généralement lorsque vous installez une librairie elle est fournie avec un certains nombres d'exemples censé expliquer son fonctionnement. On y accède via le menu File->Examples, il faut ensuite naviguer jusqu'au menu déroulant intitulé « Contributed Libraries », puis trouver le dossier correspondant à la librairie installée.

ControlP5 pour la création de GUI

ControlP5 est une librairie permettant des créer des GUI (« General User Interface »), c'est-à-dire des boutons et des glissières permettant de contrôler certains paramètres de notre programme.

En naviguant jusqu'à l'aide fournie avec la librairie, vous constaterez qu'elle est bien réelle et extensive (peut-être même un peu trop).

Le dossier /controllers/, présente l'ensemble des éléments de GUI implémentés : allant du bouton, à la liste en accordéon, en passant par les doubles sliders (ControlP5range) et autres surface de type pad XY (ControlP5slider2D).

Les dossiers/extra/ et/use/, introduisent quelques notions plus avancées, et notamment l'utilisation d'une fenêtre externe pour y placer les éléments de GUI. C'est l'exemple auquel nous allons nous intéresser.

```
01.   /**
02.    * ControlP5 Controlframe
03.    * by Andreas Schlegel, 2012
04.    * www.sojamo.de/libraries/controlp5
05.    *
06.    */
07.   import java.awt.Frame;
08.   import java.awt.BorderLayout;
09.   import controlP5.*;
10.
11.   private ControlP5 cp5;
12.
13.   ControlFrame cf;
14.
15.   int def;
16.
17.   void setup() {
18.     size(400, 400);
19.     cp5 = new ControlP5(this);
20.
21.     cf = addControlFrame("extra", 200, 200);
22.   }
23.
24.   void draw() {
25.     background(def);
26.   }
27.
28.   ControlFrame addControlFrame(String theName, int theWidth, int
   theHeight) {
29.     Frame f = new Frame(theName);
30.     ControlFrame p = new ControlFrame(this, theWidth, theHeight);
31.     f.add(p);
32.     p.init();
33.     f.setTitle(theName);
34.     f.setSize(p.w, p.h);
35.     f.setLocation(100, 100);
36.     f.setResizable(false);
37.     f.setVisible(true);
38.     return p;
39.   }
```

Dans cette première partie, on commence par importer les librairies nécessaires. On utilise ControlP5, déclarée en ligne 9, mais aussi deux classes provenant directement de Java.awt (AWT = Abstract Window Toolkit), ce qui nous permettra de créer une seconde fenêtre pour notre programme. Nous auront aussi besoin d'une classe que nous adapterons suivant nos besoins, elle est donnée ci-dessous.

Après avoir appelé nos librairies, nous créons une instance de ControlP5 qui s'appelera « cp5 » à la ligne 11, qui sera initialisée à la ligne 19 dans le setup().

Ensuite nous créons un nouveau ControlFrame appelé « cf » à la ligne 13. ControlFrame signifie fenêtre de contrôle et sera donc notre seconde fenêtre dans laquelle nous placerons notre interface. Notez bien que lors de son initialisation à la ligne 21, nous appelons une fonction spécifique qui commence à la ligne 28 et s'achève à la ligne 39. Cette fonction va appeler les fonctions java que l'on a importé pour créer une fenêtre portant le nom que l'on aura spécifié en premier argument, et les dimensions en deuxième et troisième arguments. Cette fonction n'a pas besoin d'être modifiée et peut rester la même à chaque utilisation.

Dans cette fonction à la ligne 30, on appelle le constructeur de la classe ControlFrame décrite ci-dessous. Cela permet créer la fenêtre de manière et de créer un lien entre les deux fenêtre, cela sort du cadre de cette introduction (« this » et « Object parent » y sont pour beaucoup). Il faut s'attarder sur la méthode setup() de cette nouvelle classe(lignes 47 à 53), pour comprendre comment ajouter des contrôleurs dans notre nouvelle fenêtre.

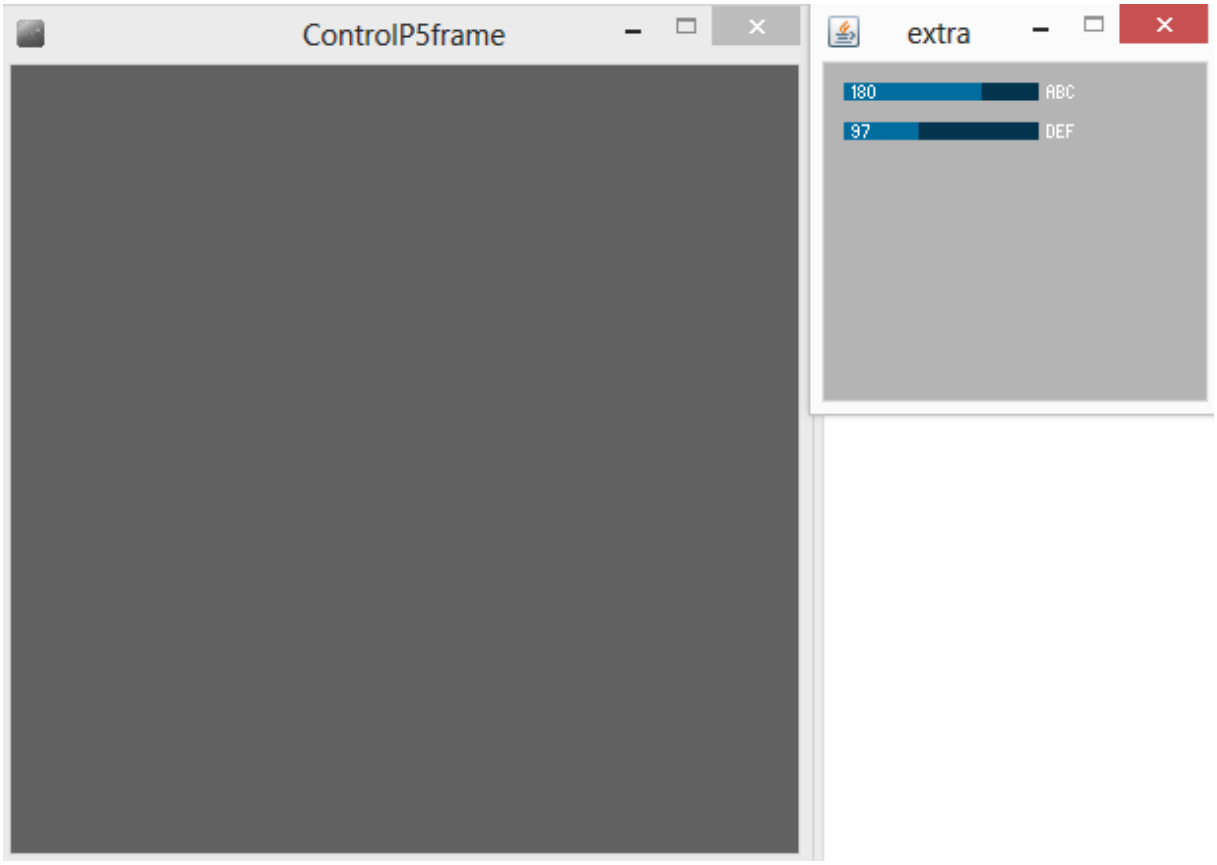
```
40.   public class ControlFrame extends PApplet {
41.
42.     int w, h;
43.     int abc = 100;
44.     ControlP5 cp5;
45.     Object parent;
46.
47.     public void setup() {
48.       size(w, h);
49.       frameRate(25);
50.       cp5 = new ControlP5(this);
51.       cp5.addSlider("abc").setRange(0, 255).setPosition(10, 10);
52.       cp5.addSlider("def").plugTo(parent, "def").setRange(0,
   255).setPosition(10, 30);
53.     }
54.
55.     public void draw() {
56.       background(abc);
57.     }
58.
59.     private ControlFrame() {
60.     }
61.
62.     public ControlFrame(Object theParent, int theWidth, int
   theHeight) {
63.       parent = theParent;
64.       w = theWidth;
65.       h = theHeight;
66.     }
67.
68.
69.     public ControlP5 control() {
70.       return cp5;
71.     }
72.   }
```

En premier lieu notre classe fait appel à une nouvelle instance de controlP5 rattachée à cette nouvelle fenêtre (ligne 50). A la ligne 51, on crée un slider appelé « abc », qui va sortir des valeurs

comprises entre 0 et 255, et qui sera situé à la position (10,10) dans la nouvelle fenêtre. De part son nom, les valeurs du slider « abc », seront directement stockées dans la variable du même nom.

A la ligne 52 on crée un second slider, cette fois appelé « def », qui va sortir le même type de valeurs, à la différence que ces données seront envoyées à notre fenêtre parente, et stockées dans notre variable de type entier « def ».

D'un façon générale, il est plus clair d'utiliser systématiquement fonction « .plugTo(this,myVar) » pour associer la valeur d'un gui à une variable. Dans notre exemple, on utilise « parent », pour spécifier qu'il s'agit d'une autre fenêtre que l'on a préalablement définit, mais l'utilisation de « this » permet de le faire aussi lorsque contrôles et dessins se passent dans la même fenêtre.



File->Examples->ContributerLibrairies-> ControlP5->Extra->ControlP5frame.pde

ControlP5 est une librairie très aboutie avec beaucoup de fonctionnalités, mais sa mise en œuvre peut-être parfois un peu lourde. S'il s'agit de faire des tests, le mode « Tweak » peut s'avérer être une bonne alternative.

OSCP5 pour la communication avec d'autres programmes

OSCP5 est un support pour processing de la fameuse librairie de communication entre différents paradigmes de programmation. OSC est présent dans quasiment tous les langages c'est donc un classique à connaitre et à utiliser sans modération.

Les exemples *sketch_3_01_OSC_Receive.pde* et s*ketch_3_01_OSC_Send.pde* sont donc à utiliser conjointement. Le premier programme recevra des informations du second, et changera sa couleur

de fond en fonction de la valeur reçue. Le second programme, enverra une valeur aléatoire lorsque l'on clique sur sa fenêtre.

Pour effectuer cela, il important de pouvoir spécifier une adresse ip pour pouvoir envoyer un message a un endroit précis et un numéro de port.

```
01.   //Receiver
02.   import oscP5.*;
03.   import netP5.*;
04.
05.   OscP5 oscP5;
06.
07.   float receivedValue;
08.
09.   void setup() {
10.     size(400, 400);
11.     frameRate(25);
12.
13.     receivedValue =0;
14.
15.     oscP5 = new OscP5(this, 1234);
16.   }
17.
18.   void draw() {
19.     background(receivedValue);
20.   }
21.
22.   void oscEvent(OscMessage theOscMessage) {
23.     if (theOscMessage.checkAddrPattern("/test")==true) {
24.       float firstValue = theOscMessage.get(0).floatValue();
25.       String secondValue = theOscMessage.get(1).stringValue();
26.       receivedValue = firstValue;
27.     }
28.   }}
```

Les lignes 2 et 3 permettent d'importer les objets nécessaires. La ligne 5 crée une instance d'OSCP5 La ligne 15, située dans le setup est primordiale pour le bon fonctionnement de notre programme, le second argument fourni (ici « 1234 ») est nôtre numéro de port, cette ligne signifie donc que nous écouterons toutes les informations entrantes sur cette machine (« this ») transitant par le port « 1234 ». Lorsqu'une telle information sera reçue, elle activera la fonction présente de la ligne 22 à la ligne 28.

Un message OSC est avant tout un préfixe « /test » ici, permettant de trier les messages arrivant puis un tableau remplit de variable de différents types, il est alors toujours plus pratique de savoir exactement ce que l'on va recevoir. Ici, comme vous pourrez le voire ci-après notre message est , d'un préfixe « /test » puis est composé : d'un float aléatoire, et d'un string constant. Pour accéder à l'élément « n » du message on utilise « theOscMessage.get(n) », puis on utilise l'accesseur approprié au type de valeur que l'on reçoit (« .floatValue() » pour un float).

La ligne 23 reçoit donc une valeur flottante, la ligne 25 stocke cette valeur dans une variable qui va servir à spécifier la couleur de fond de notre fenêtre.

```
01.   // Sender
```

```
02.   import oscP5.*;
03.   import netP5.*;
04.
05.   OscP5 oscP5;
06.   NetAddress myRemoteLocation;
07.
08.   void setup() {
09.     size(400, 400);
10.     frameRate(25);
11.
12.     oscP5 = new OscP5(this, 12000);
13.     myRemoteLocation = new NetAddress("127.0.0.1", 1234);
14.   }
15.
16.
17.   void draw() {
18.     background(0);
19.   }
20.
21.   void mousePressed() {
22.
23.     OscMessage myMessage = new OscMessage("/test");
24.
25.     myMessage.add(random(255));
26.     myMessage.add("hello!");
27.
28.     oscP5.send(myMessage, myRemoteLocation);
29.   }
```

Jusqu'à la ligne 6, rien de nouveau sous le soleil. A la ligne 6, apparait cependant un nouvel objet propre à OSCP5 : un objet de type NetAdress, c'est en fait un couple composé d'une adresse ip et d'un numéro de port, comme vous pouvez le constater à la ligne 13

Le reste du programme tient dans la fonction mousePressed() (lignes 21 à 29). A la ligne 23 on crée un nouveau message avec un préfixe spécifique. A la ligne 25 on lui ajoute une première donnée (un nombre aléatoire), puis une seconde à la ligne 26. La ligne 28, envoit notre message à l'adresse que nous avons spécifié dans le setup.

N'oubliez pas de faire attention aux adresses ip et au numéros de ports lorsque vus utilisez OSCP5, n'hésitez pas non plus à faire des print pour être bien sûrs de recevoir vos messages !

Le Chapitre suivant vous donnera un exemple permettant de de commander Processing à l'aide d'une analyse audio faite dans Pure Data.

Travailler avec les images

Processing utilise un classe pour travailler avec les images, pour les manipuler nous avons recours à l'objet « PImage ».

Charger et afficher une image

Pour charger et afficher une image dans Processing, il faut d'abord s'assurer qu'elle soit d'un type accepté par processing à savoir : `.gif`, `.jpg`, `.tga`, ou `.png`.

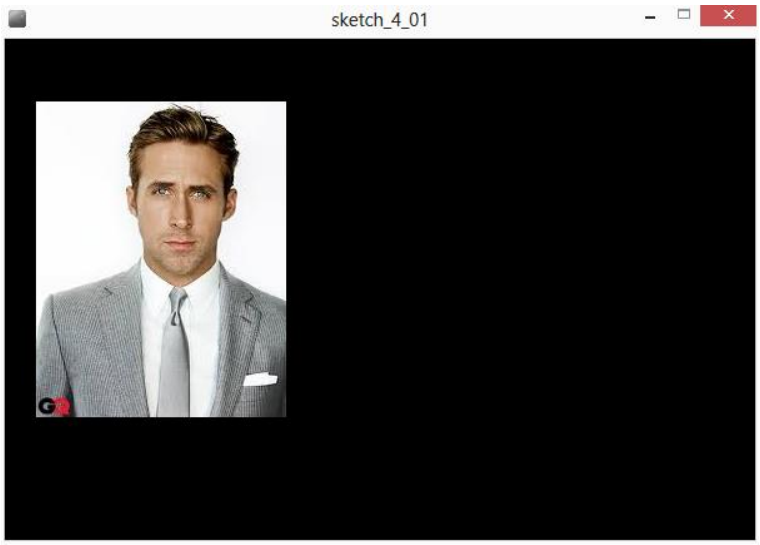
Il faut ensuite s'assurer qu'elle soit visible par le programme sur lequel on travaille. Pour cela il est préférable de sauvegarder le sketch, puis de glissr l'image à ajouter au sketch sur le fenêtre de processing.

Dans le dossier du sketch en question (menu : Sketch -> Show sketch floder) apparaitra alors un dossier nommé « data », qui contiendra votre image.

Une fois ces opération effectuées vous pouvez charger votre image comme ceci :

```
01.   PImage img;
02.
03.   void setup(){
04.
05.     img = loadImage("visage.jpg");
06.     size(600,400);
07.     println(img.width, img.height);
08.
09.   }
10.
11.   void draw(){
12.     background(0);
13.     image(img,0,0);
14.   }
```

A la ligne 1 on crée un nouvel objet PImage appelé « img », on l'initialise à la ligne 5 en chargeant l'image présente dans le dossier data. En suite à la ligne 13, on affiche cette image, au point de coordonnées (0,0).



Sketch_4_01.pde

Acccéder aux pixels

Une fois l'image chargée, il est possible de faire énormément de manipulations, il existe beaucoup d'exemples dans la documentation en ligne de processing a propos des filtres que l'on peut appliquer. Nous allons plutôt nous intéresser à la manipulation de pixels.

Il existe une fonction appelée « loadPixels() » qui permet de charger automatiquement les couleurs des pixels d'une image dans un tableau qui se nommera pixels[].

Le code suivant permet de lire la teinte du pixel d'une image pré-chargée à l'endroit précis du curseur de la souris :

```
01.   PImage img ;
02.
03.   void setup(){
04.     background(0);
05.     img = loadImage("ville.jpg");
06.     size(img.width,img.height,P3D);
07.
08.   }
09.
10.
11.   void draw(){
12.     background(0);
13.     image(img,0,0);
14.
15.     img.loadPixels();
16.     int mousePos = mouseX + mouseY*width;
17.     println(hue(img.pixels[mousePos]));
18.
19.   }
```

Skecth_4_02.pde

Jusqu'à la ligne 14, normalement tout va bien. La ligne 15 va appeler la fonction loadPixels() sur l'image que nous avons chargée, cela aura pour effet de nous permettre d'utiliser le tableau de pixels à la ligne 17 (« img.pixels[] »). Nous utilisons directement la fonction hue pour connaitre la teinte du pixel en question (se référer à l'usage de la fonction hue(myColor) dans la documentation en ligne). Il faut cependant bien noter qu'à la ligne 16 nous convertissons les coordonnées de la souris en index linéaire dans un tableau.

En effet, pixels[] est un tableau, à titre chaque valeur est stockée à un index précis, mais cet index est à une seule dimension, alors que les coordonnée de la souris sont en 2D.

Faire « sortir » les pixels en 3D

A partir de maintenant il devient très facilement possible d'effectuer tout un tas d'effets artistiques animés, en se basant sur les données des pixels. Dans le programme suivant nous allons nous attacher à déplacer les pixels en fonction de leur luminosité (et de la position de la souris, pour accentuer ou diminuer l'effet).

Afin d'avoir un programme plus rapide, nous allons avoir un setup() un peu plus long que d'habitude. En effet pour ne pas avoir à appeler la fonction loadPixels() en permanence, nous allons le faire une

seule fois dans le setup de notre programme. Cela nous permettra de remplir des tableaux pour mieux organiser nos données et pouvoir ainsi les dessiner plus simplement. Comme précédemment nous allons parcourir chaque pixel de l'image à l'aide d'une double boucle for, et nous allons stocker dans des tableaux de même dimension, les coorodonnées en x dans le tableau xC[], les coordonnées en y dans le tableau yC[] et la couleur de chaque pixel dans le tableau pColor[].

```
01.   PImage img ;
02.
03.   int [] xC;
04.   int [] yC;
05.   int [] pColor;
06.
07.   void setup() {
08.     size(500, 400, P3D);
09.     background(0);
10.     img = loadImage("image_200x100.jpg");
11.     img.loadPixels();
12.
13.     xC = new int[img.pixels.length];
14.     yC = new int[img.pixels.length];
15.     pColor = new int[img.pixels.length];
16.
17.     for (int i =0 ; i < img.width ; i++) {
18.       for (int j = 0 ; j < img.height; j++) {
19.         int loc = i + j*img.width;
20.         xC[loc] =i;
21.         yC[loc] =j;
22.         pColor[loc] = img.pixels[loc];
23.       }
24.     }
25.   }
```

Notez bien l'apparition du mode P3D dans l'instruction de taille de la fenêtre. Maintenant que nous avons ces informations, nous allons parcourir nos tableaux à chaque image et pour chaque valeur de l'index : créer un rectangle de la même couleur que le pixel d'origine et d'un taille de 1px X 1px, bref nous allons recréer l'image avec nos propres objets. La seule différence étant que nous allons ajouter une composante de translation en z, permettant de faire ressortir les pixels.

Pour obtenir l'effet souhaité, nous allons cependant le faire en deux fois. L'objectif est d'avoir l'image intacte quand la souris est à gauche de l'écran et « éclatée » quand la souris est à droite. A la ligne04 ci-dessous, nous allons donc définir une variable valant 0 quand la souris est à gauche et 60 quand la souris est à droite, ce que nous avions déjà fait précédemment.

Ensuite nous créons une boucle pour parcourir nos tableaux, nous stockons la luminosité de chaque pixel dans une variable à la ligne 07, puis nous nous apprêtons à dessiner avec des translations (usage du pushMatrix() …). La ligne 11 contient le positionnement de chaque forme : on décale de 150 pixels vers la droite et de 150 pixels vers le bas pour mieux centrer, puis on utilise les coordonnées stockées, en dernier paramètre on multiplie nos deux variable (position de la souris et luminosité) pour faire en sorte que les pixels les plus lumineux ressortent le plus.

```
01.   void draw() {
02.     background(0);
```

```
03.
04.     float push_z = map(mouseX, 0, width, 0, 60);
05.
06.     for (int i = 0 ; i < xC.length ; i++) {
07.       float br = brightness(pColor[i]);
08.       pushMatrix();
09.       noStroke();
10.       fill(pColor[i]);
11.       translate(150+ xC[i]+5, 150+yC[i]+5, push_z*br/20);
12.       rect(0, 0, 1, 1);
13.       popMatrix();
14.     }
15.   }
```

Sketch_4_03.pde

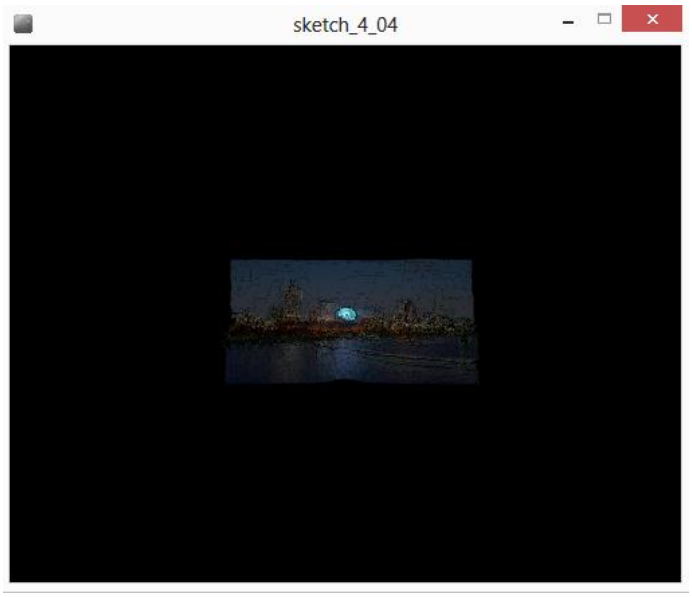
3D et audio-réactif avec Pure-Data

Pour compléter ce petit projet nous allons ajouter une partie audio-réactive, le temps pour Pure-Data et OSC de se rappeler à nos bons souvenirs…

Pour faire simple nous n'allons plus utiliser la souris pour contrôler la quantité de déplacement de nos pixels, mais plutôt le niveau sonore ambiant dans la pièce dans laquelle nous travaillons. Nous allons faire l'analyse audio dans Pure-Data puis envoyer les données via OSC à Processing.

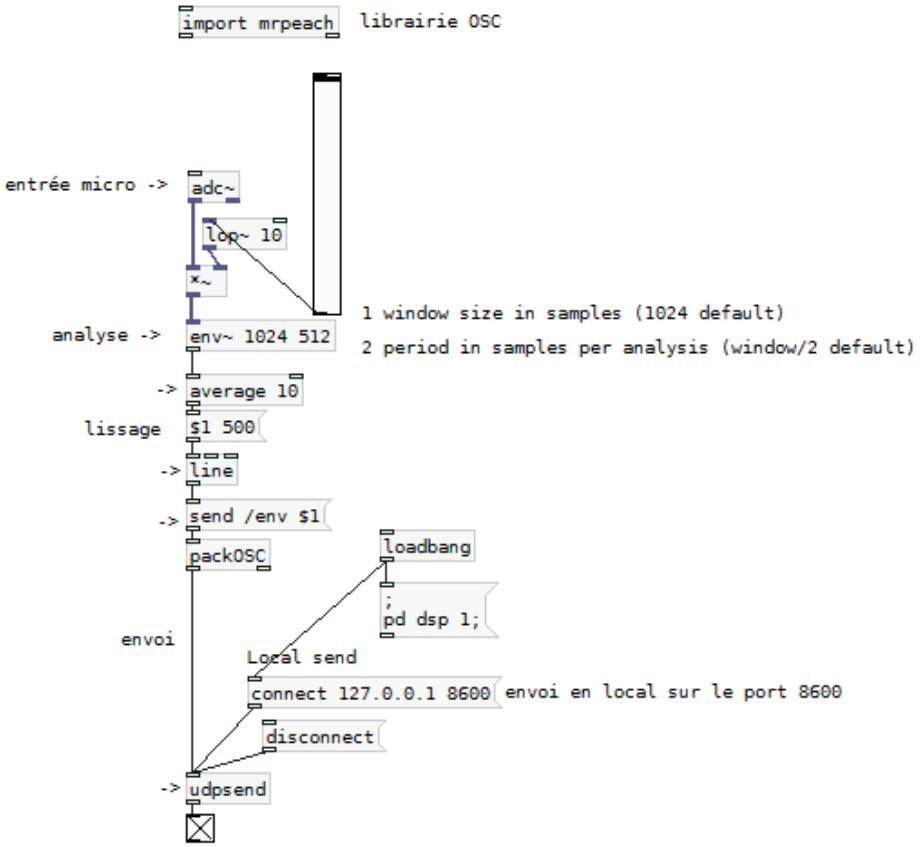
En ce qui concerne le code Processing, nous prenons le programme précédent, auquel nous apportons quelques modifications, notamment pour ajouter la librairie OSCP5 et ajuster la réception des messages (un seul float avec le préfixe « /env »). Il faut aussi pense à la ligne 40 à changer la valeur de push_z en fonction des données que l'on reçoit.

```
01.  import oscP5.*;
02.  import netP5.*;
03.
04.  OscP5 oscP5;
05.
06.  float r_env = 0 ;
07.
08.  PImage img ;
09.
10.  int [] xC;
11.  int [] yC;
12.  int [] pColor;
13.
14.  void setup() {
15.    size(500, 400, P3D);
16.    background(0);
17.    img = loadImage("image_200x100.jpg");
18.    img.loadPixels();
19.
20.     oscP5 = new OscP5(this, 8600);
21.
22.    xC = new int[img.pixels.length];
23.    yC = new int[img.pixels.length];
24.    pColor = new int[img.pixels.length];
25.
26.    for (int i =0 ; i < img.width ; i++) {
27.      for (int j = 0 ; j < img.height; j++) {
28.        int loc = i + j*img.width;
29.        xC[loc] =i;
30.        yC[loc] =j;
31.        pColor[loc] = img.pixels[loc];
32.      }
33.    }
34.  }
35.
36.
37.  void draw() {
38.    background(0);
39.
40.    float push_z = map(r_env, 45, 90, 0, 60);
41.
42.    for (int i = 0 ; i < xC.length ; i++) {
43.      float br = brightness(pColor[i]);
44.      pushMatrix();
45.      noStroke();
46.      fill(pColor[i]);
47.      translate(150+ xC[i]+5, 150+yC[i]+5, push_z*br/20);
48.      rect(0, 0, 1, 1);
49.      popMatrix();
50.    }
51.  }
52.
53.
54.  void oscEvent(OscMessage theOscMessage) {
55.    if (theOscMessage.checkAddrPattern("/env")==true) {
56.      float firstValue = theOscMessage.get(0).floatValue();
57.
58.      r_env = firstValue;
59.    }
60.  }
```



Sketch_4_04.pde

Lepatch Pure-Data est quand à lui très simple : il utilise la librairie « mrpeach » qui permet l'utilisation de boites spécifiques à l'envoi de messages OSC.



Pd-analysis.pd

Trucs et astuces

Inspiré de la rubrique de Amnon sur son wordpress :

http://amnonp5.wordpress.com/2012/01/28/25-life-saving-tips-for-processing/

IDE

Ctrl + T : permet de formater le texte de notre code en le ré-indentant en fonction des accolades.

File -> Preferences : emplacement du sketchbook

File -> Examples : exemples de programmes classés selon différentes catégories, la documentation des librairies est aussi disponible sous cet onglet.

Programmation

abréviation des opérations

```
01.     i = i+1 ;
```

est équivalent à

```
01.     i+=1 ;
```

Graphisme

Un blur très simple

Au lieu d'effacer le fond à chaque image en utilisant

```
01.     background (maCouleur) ;
```

Il est très simple de créer un effet de « blur » en utilisant de la transparence.

```
01.     Fill(0,20) ;
02.     Rect(0,0,width,height) ;
```

Color Selector

Utilisez l'outil Color Selector pour spécifier plus facilement vos couleurs dans les different modes.

Tools -> Color Selector.

Il existe aussi le mode Tweak qui est souvent plus simple que de créer un GUI complet.

In/Out

Sauvegarder une image

Pour sauvegarder un image on peut utiliser la fonction saveFrame(), on peut la coupler avec une interaction clavier, ainsi qu'une condition pour que la sauvegarde s'effectue lorsqu'on appuie sur la touche S. Le must est de composer une chaîne de caractère pour que chaque fichier ait un nom unique.

```
01.     // function d'interception des évenements clavier
```

```
02.      void keyPressed() {
03.      // condition pour identifier si la touche est un “s”
04.      if (key == 's' || key == 'S') {
05.      /* composition d’une string comportant le nom du sketch puis
         des informations de temps à l’aide de functions de processing pour
         récupérer des événements temporels*/
06.      String name = "monSketch-"+year()+"-"+month()+"-"+day()+"-
         "+hour()+"h"+minute()+"m"+second()+"s.png"
07.      saveFrame(name);
08.      }
```

}

Redimensionner une image

Il suffit de changer la taille de la fenêtre du programme pour sauvegarder une image au nouvelles dimensions.

```
01.   PImage img ;
02.
03.   void setup(){
04.     size(200,100,P3D);
05.     background(0);
06.     img = loadImage("ville.jpg");
07.     img.resize(width,height);
08.   }
09.
10.
11.   void draw(){
12.     background(0);
13.     image(img,0,0);
14.     saveFrame("image_"+width+"x"+height+".jpg");
15.     noLoop();
16.   }
```

Ressources

Site officiel : http://processing.org/

La référence de l'API processing : http://processing.org/reference/

Le Wiki(parfois la référence n'est pas complète) : http://wiki.processing.org/w/Main_Page

Les tutoriaux officiels : http://processing.org/tutorials/

Plus à propos de processing : http://en.wikipedia.org/wiki/Processing_(programming_language)

Initiation (français) : http://fr.flossmanuals.net/processing/

Ressources diverses : http://codelab.fr/39

Forum dédié à Processing (français) : http://codelab.fr/processing

Forum officiel de Processing (anglais) : http://forum.processing.org/two/

Ce forum est tout neuf et ne contient pas encore les archives des années précédentes, l'ancien forum est accessible à cette adresse(http://forum.processing.org/one/) et reste bien utile notamment pour ses archives.

Tutoriel (français) : http://www.ecole-art-aix.fr/rubrique81.html

D'autres tutoriels en français : http://tutoprocessing.com/tutos/

Vidéos de fun programming : http://funprogramming.org/

Vidéos de Daniel Shiffmann :

Computer programming for total beginner : https://vimeo.com/channels/introcompmedia

Nature of Code : https://vimeo.com/channels/natureofcode