PROCESING

Copyright : Les Contributeurs (voir ci-dessous)
Publié : 07 Apr 2011
ISBN :
Licence : GPL
Note : Une attention particulière a été apportée aux exemples et illustrations de ce manuel afin d'offrir une information complète et fiable. Nous n'offrons cependant aucune garantie et déclinons toute responsabilité de préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

PROCESSING

TABLE DES MATIÈRES

PRÉSENTATION **7**
1. INTRODUCTION 9
2. EXEMPLES D'UTILISATION 13
3. L'INSTALLATION DE PROCESSING 23
4. LES BASES DE PROCESSING 29

DESSINER **35**
5. L'ESPACE DE DESSIN 37
6. LES FORMES 41
7. LES COULEURS 49
8. LE TEXTE 55

DESSINER PLUS **59**
9. LES IMAGES 61
10. LES STYLES DE BORDURES 73
11. LA TYPOGRAPHIE 77
12. LES TRANSFORMATIONS 83

PROGRAMMER **97**
13. LES VARIABLES 99
14. LES CONDITIONS 103
15. LES RÉPÉTITIONS 107
16. LES LISTES 113
17. LES MÉTHODES 125
18. LES OBJETS 135
19. LES COMMENTAIRES 141

ANIMER **143**
20. LA MÉTHODE DRAW 145
21. LA LIGNE DE TEMPS 153
22. L'ANIMATION D'UN OBJET 159
23. L'ANIMATION DE PLUSIEURS OBJETS 165
24. LA LECTURE DU SON 173

SORTIR **183**
25. L'EXPORTATION 185
26. L'IMPRESSION 189
27. LA VIDÉO 195

INTERAGIR **199**
28. LES ÉVÉNEMENTS CLAVIER 201
29. LES ÉVÉNEMENTS SOURIS 207
30. L'ENTRÉE MICROPHONE 213
31. L'ENTRÉE VIDÉO 219

ANNEXES **223**
32. LES ASTUCES 225
33. LES ERREURS COURANTES 231
34. LES LIBRAIRIES EXTERNES 235
35. LA DOCUMENTATION EN LIGNE 241
36. PROCESSING DANS L'ENSEIGNEMENT 243
37. ARDUINO 247
38. A PROPOS DE CE MANUEL 257
39. GLOSSAIRE 261
40. LES AUTEURS 269
41. LICENCE 279

PRÉSENTATION

1. Introduction
2. Exemples d'utilisations
3. L'installation de Processing
4. Les bases de Processing

1. INTRODUCTION

Conçu par des artistes, pour des artistes, Processing est un des principaux environnements de création utilisant le code informatique pour générer des œuvres multimédias sur ordinateur. L'attrait de ce logiciel réside dans sa simplicité d'utilisation et dans la diversité de ses applications : image, son, applications sur Internet et sur téléphones mobiles, conception d'objets électroniques interactifs.

Processing fédère une forte communauté d'utilisateurs professionnels et amateurs : artistes, graphistes, vidéastes, typographes, architectes, web designers et designers en général. Il est également utilisé par des enseignants en arts qui souhaitent familiariser leurs étudiants avec les potentialités artistiques de la programmation, les concepteurs du logiciel l'ayant pensé dès l'origine comme un outil d'apprentissage.

DESSINER ET CRÉER AVEC DU CODE INFORMATIQUE

Logiciel de création multimédia, Processing possède la particularité d'utiliser des instructions informatiques pour dessiner, réaliser des animations en 2 ou 3 dimensions, créer des œuvres sonores et visuelles, concevoir des objets communicants qui interagissent avec leur environnement.

Pour un artiste habitué à utiliser à main levée des outils comme son crayon, son pinceau, sa souris ou sa tablette graphique, il peut sembler surprenant de créer des formes, des couleurs, des mouvements en saisissant seulement une suite d'instructions à l'aide de son clavier.

Ce mode d'expression artistique par le code utilise les caractéristiques propres à l'informatique (rapidité d'exécution, automatisation des actions et des répétitions, interaction, etc.) pour produire des créations originales qui n'auraient pas vu le jour autrement ou dont la réalisation, à l'aide de procédés plus classiques ou de logiciels plus complexes, aurait demandé davantage de temps.

Processing permet également de programmer des circuits électroniques qui interagissent avec le milieu qui les entoure. Connectés à des capteurs sonores, thermiques, de mouvement, ces circuits électroniques peu coûteux, dénommés micro-contrôleurs, peuvent en retour générer des images, actionner un bras articulé, envoyer des messages sur Internet... bien entendu en fonction du programme que vous aurez créé.

Comme nous le verrons dans ce manuel, en apprenant à programmer avec Processing, vous allez développer votre capacité d'expression et d'imagination.

UN FORMIDABLE ENVIRONNEMENT D'APPRENTISSAGE

Si Processing est simple d'utilisation, c'est qu'il a été conçu dès son origine pour servir à enseigner les bases de la programmation informatique dans un contexte visuel.

La vocation pédagogique de Processing en fait un excellent outil d'apprentissage de la programmation pour les non-programmeurs ou les programmeurs débutants. De nombreux enseignants l'utilisent pour initier leurs élèves et leurs étudiants aux concepts et aux bonnes pratiques de la programmation.

Plusieurs universités, écoles et centres d'artistes donnent des cours sur ce logiciel. Dans un contexte francophone, on peut mentionner notamment l'Université du Québec à Montréal (UQAM), l'Université Laval, la Société des arts technologiques - SAT (Canada-Québec), le Centre art sensitif - Mains d'oeuvres à Paris, l'Ecole d'Art d'Aix en Provence, l'association PING à Nantes (France), l'Ecole de Recherche Graphique (ERG) et IMAL à Bruxelles, la Haute Ecole Albert Jacquard (HEAJ) à Namur (Belgique), La Haute Ecole d'Art et de Design (HEAD) à Genève, La Haute Ecole d'Arts Appliqués de Lausanne - ECAL (Suisse).

UN LOGICIEL LIBRE ET GRATUIT

Processing est un logiciel libre et gratuit. Il fonctionne sur les plates-formes Windows, Linux, Mac (et sur toute autre plate-forme pouvant faire fonctionner des logiciels conçus en Java). Il existe également des versions pour téléphones portables et des dérivés pour circuits électroniques.

En tant que logiciel libre, Processing bénéficie de la générosité de nombreux programmeurs volontaires qui mettent à disposition des utilisateurs des morceaux de codes facilement réutilisables (dénommés en jargon informatique des librairies). Plus d'une centaine de librairies étendent ainsi les capacités du logiciel dans le domaine du son, de la vidéo, de l'interaction, etc.

UN PEU D'HISTOIRE

Processing a été conçu au laboratoire Aesthetics + Computation Group (ACG) du MIT Media Lab par Ben Fry et Casey Reas en 2001. Ce logiciel est plus ou moins le prolongement du projet *Design By Numbers*, créé par le directeur du laboratoire, l'artiste-programmeur John Maeda. Dans son livre présentant le langage de programmation qu'il a conçu, Maeda met en avant la simplicité et l'économie d'action dans la programmation d'images.

Plusieurs éléments de ce premier projet sont visibles dans l'environnement Processing : la simplicité de l'interface du logiciel, la priorité donnée à

l’expérimentation et l’apprentissage, ainsi que les nombreuses fonctions que les deux environnements partagent. Les concepteurs de Processing ne cachent pas cet héritage.

La version actuelle de Processing est la version 1.2.1. Les exemples donnés dans ce manuel font référence à cette version du logiciel.

COMMENT UTILISER CE MANUEL

Production originale en français, ce manuel est destiné au public professionnel et amateur qui souhaite s'initier à l'utilisation de Processing. Il ne réclame aucune connaissance préalable de programmation. L'apprentissage de Processing y est fait pas à pas. Nous vous invitons à suivre l'ordre de succession des chapitres, surtout pour les tout premiers qui posent les bases de l'utilisation du logiciel.

La saisie des exemples de programmes proposés dans ce manuel peut constituer à elle seule une expérience formatrice, ne serait-ce qu'en apprenant à éviter de faire des erreurs de frappe. Si cette activité vous rebute ou si vous voulez rapidement expérimenter le résultat d'un exemple, copiez le code du programme depuis les pages web de l'ouvrage (consultables sur la partie francophone du site Flossmanuals à l'adresse http://fr.flossmanuals.net/processing/) pour ensuite le coller directement dans la fenêtre du logiciel.

Disponible en plusieurs formats numériques (html, pdf, ePub) ainsi qu'en version papier, ce manuel est publié sous licence GPLv2 : vous êtes autorisé à le lire et à le copier librement.

Ouvrage collectif, ce manuel est vivant : il évolue au fur et à mesure des contributions. Pour consulter la dernière version actualisée, nous vous invitons à visiter régulièrement le volet francophone de Flossmanuals sur le site http://fr.flossmanuals.net.

Le cœur du manuel d'environ 270 pages a été réalisé en 5 jours dans le cadre d'un Booksprint qui s'est tenu à Paris du 6 au 10 septembre 2010 à l'initiative et avec le soutien de l'Organisation internationale de la Francophonie (http://www.francophonie.org).

Vous consultez l'édition révisée et augmentée du 22 novembre 2010.

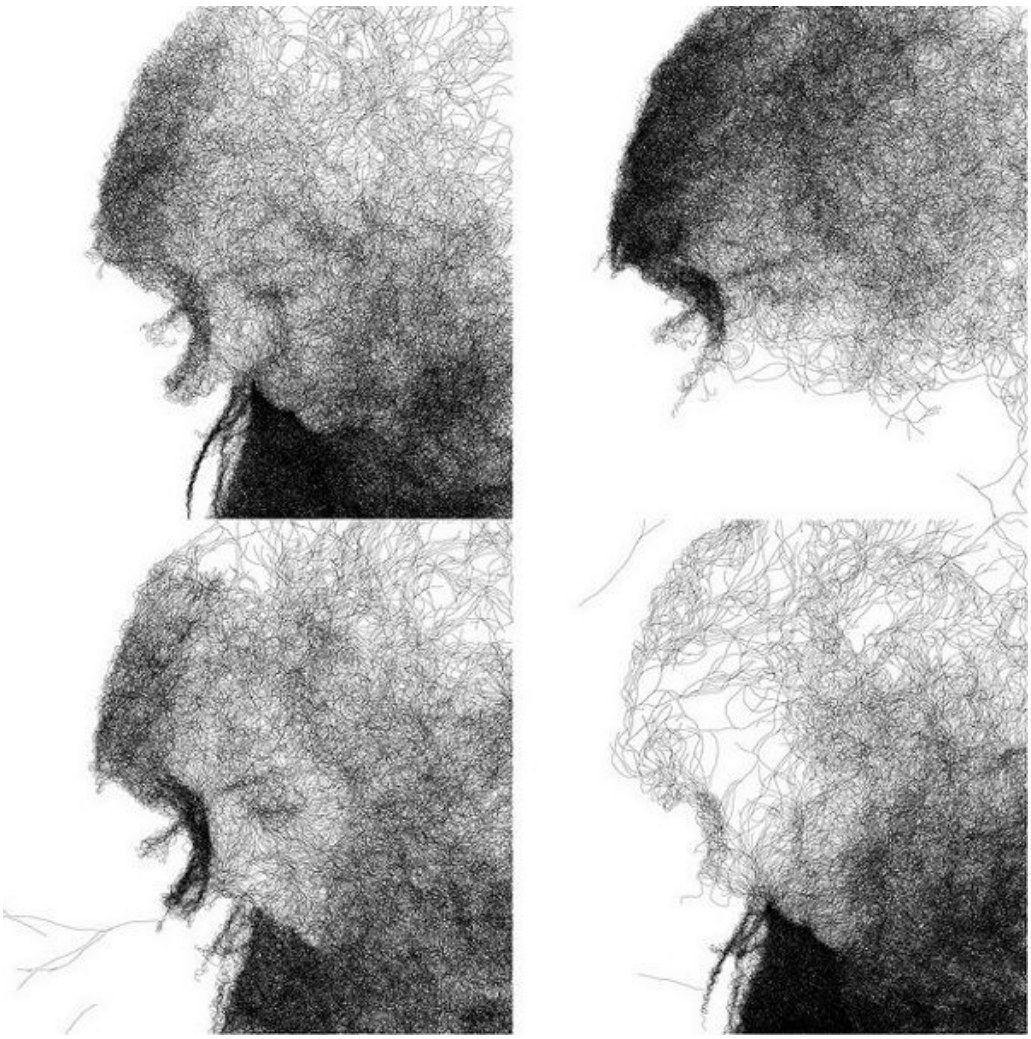
2. EXEMPLES D'UTILISATION

Logiciel de création multimédia, Processing permet de dessiner, réaliser des animations en 2 ou 3 dimensions, créer des œuvres sonores et visuelles, concevoir des objets électroniques qui interagissent avec leur environnement. Des dizaines de milliers d'artistes, de designers, d'architectes, de chercheurs et même d'entreprises l'utilisent pour réaliser des projets incroyables dans de multiples domaines :

- **publicités, génériques de films, vidéos clips, dessins animés**, Processing permettant de créer des effets visuels originaux ;
- **visualisation de données scientifiques** sous la forme d'images fixes ou animées, facilitant ainsi la représentation d'informations complexes dans de multiples secteurs professionnels (environnement, transports, économie, sciences humaines, etc.) ;
- **production musicale**, Processing permettant non seulement de lire mais aussi de transformer et de créer du son ;
- **spectacle vivant**, grâce aux nombreuses fonctions d'interaction offertes par Processing, il est possible de créer des performances de VJing, utiliser le mouvement des danseurs pour générer en temps réel des effets sonores et visuels dans un spectacle, réaliser des œuvres d'arts numériques interactives ;
- **Architecture**, Processing facilitant la représentation spatiale, il est utilisé dans des projets d'architecture pour automatiser le dessin de constructions en 2 ou 3 dimensions.

La suite de ce chapitre va vous présenter quelques exemples de travaux réalisés avec l'aide de Processing dans différents contextes.

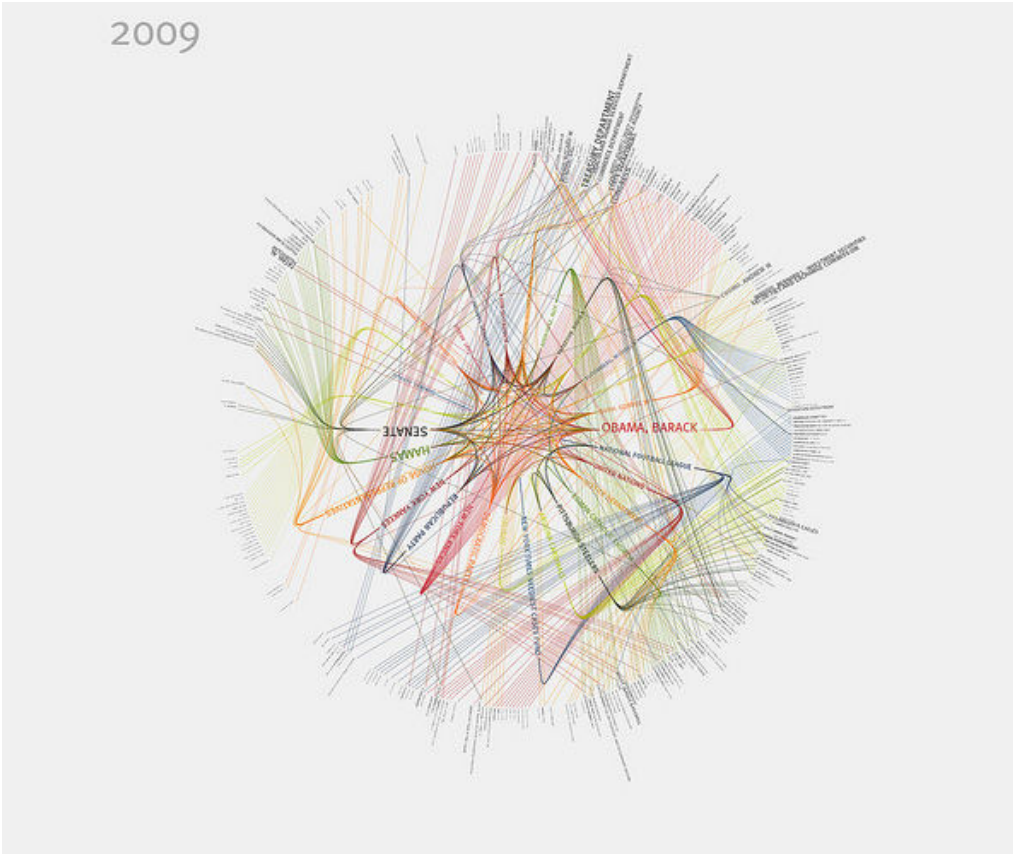
MYCELIUM



Des traits apparaissent sur l'écran, se ramifient pour finalement constituer un visage. A partir d'une image d'origine, ce programme d'animation conçu avec Processing simule le développement du mycelium, la partie végétative des champignons, et utilise ce procédé à des fins esthétiques.

Réalisé en 2010 par Ryan Alexander : http://onecm.com/projects/mycelium/

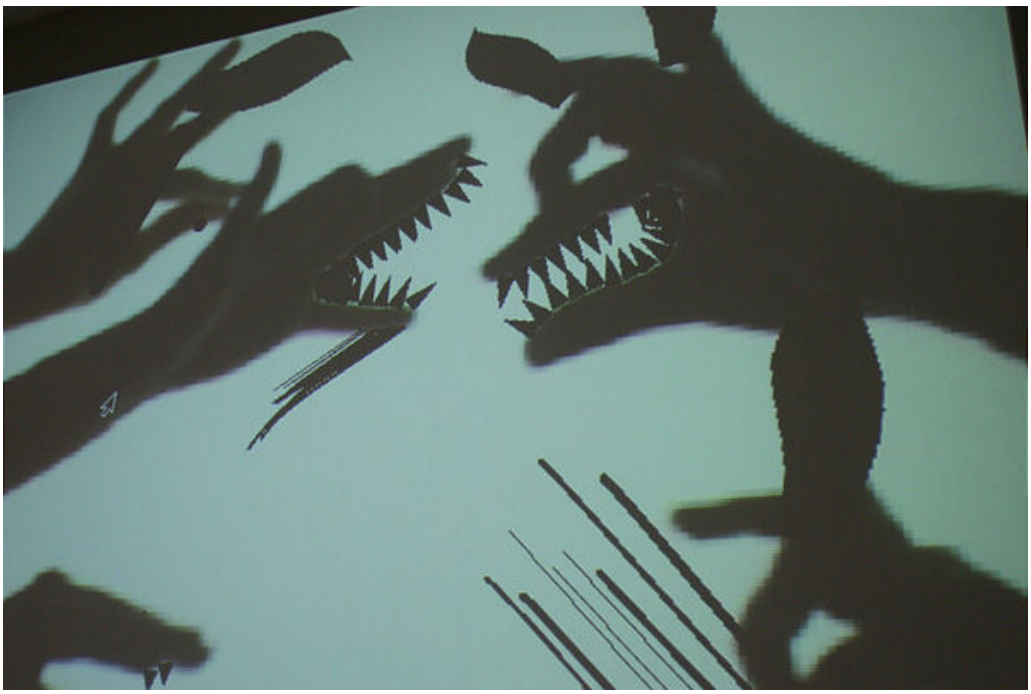
NYTIMES 365/360



NY Times est un système de visualisation de données basé sur les flux d'information du site internet New York Times. Utilisant les actualités du journal comme source de données, le programme rend visible sous la forme d'un graphe le degré d'importance des mots employés et leur relation entre eux. Au final, on obtient une série d'images en haute résolution destinée à l'impression.

Réalisé en 2009 par Jer Thrope:
http://blog.blprnt.com/blog/blprnt/7-days-of-source-day-2-nytimes-36536

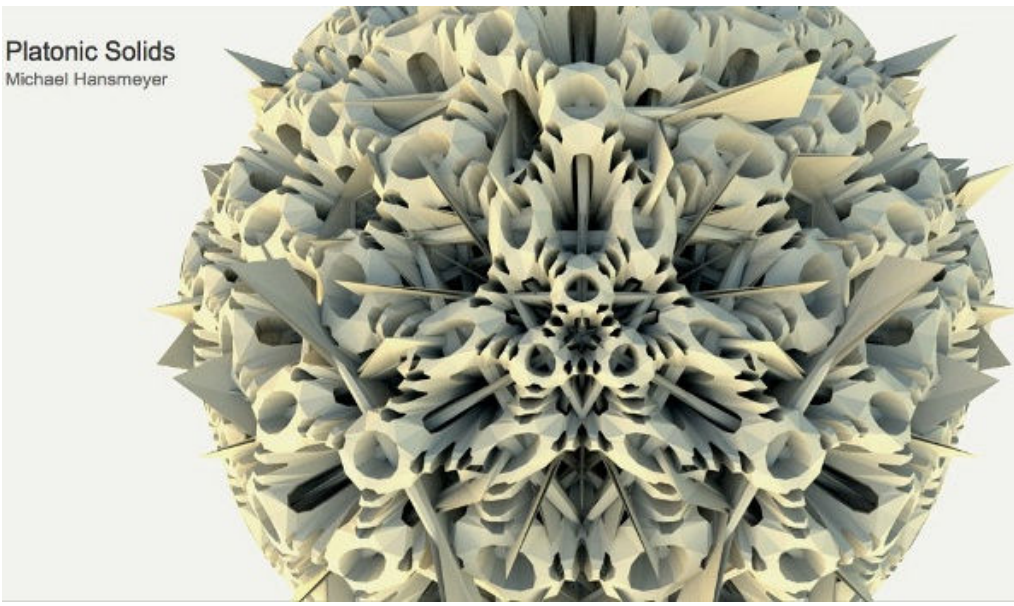
SHADOW MONSTERS



Shadow Monster est une installation interactive qui fonctionne sur le principe des ombres chinoises. Le public qui passe devant l'installation voit son ombre se transformer en monstre, des excroissances étranges poussant instantanément sur les extrémités du corps. Sur cette image, des personnes se sont amusées à réaliser des figures animales avec l'ombre portée de leurs mains, l'installation interactive prenant de son côté l'initiative d'y rajouter des dents, des touffes de cheveux, des antennes ou des yeux.

Réalisé en 2005 par Philip Worthington:
http://worthersoriginal.com/viki/#page=shadowmonsters

PLATONIC SOLIDS



Création de formes géométriques complexes s'inspirant de l'ouvrage *Kunstformen der Natur* (Formes d'art de la nature) du scientifique et talentueux dessinateur du XIXe siècle, Ernst Haeckel.

Réalisé par Michael Hansmeyer :
http://www.michael-hansmeyer.com/html/solids/pos.html

CHAMP D'OZONE

Champ d'Ozone est une installation artistique présentée en 2007 à Paris dans le cadre de l'exposition "Airs de Paris". Commandée par le Centre d'art et de culture Georges Pompidou, elle a été conçue en collaboration avec Airparif, un organisme de surveillance de la qualité de l'air en Ile-de-France. Cette œuvre interactive projette sur la surface d'une vitre des nuages virtuels dont la couleur varie constamment selon le niveau de pollution atmosphérique à Paris.

Réalisé en 2007 par HeHe: http://hehe.org.free.fr/hehe/champsdozone/

COP15 GENERATIVE IDENTITY

L'identité graphique de la Conférence des Nations-Unies sur le climat à été réalisée par un studio londonien avec Processing. Les créatifs de cette agence de communication ont conçu un outil pour générer un logo animé basé sur des principes d'interaction de force illustrant ainsi la complexité des échanges lors de cette conférence.

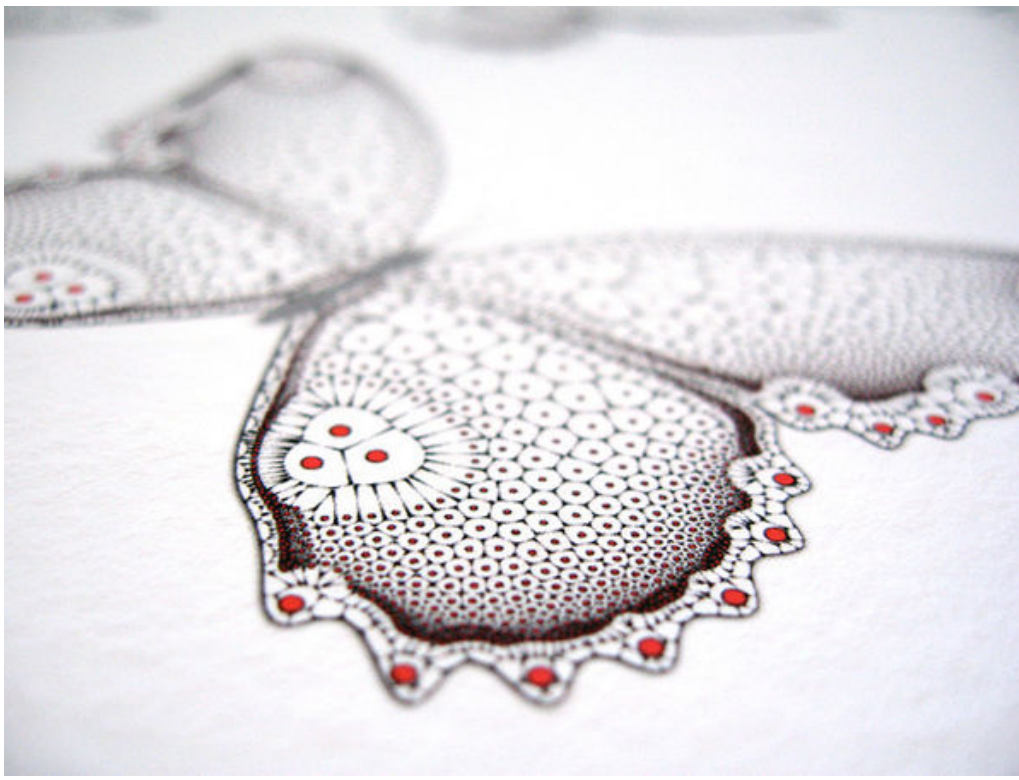
Réalisée en 2009 par le studio okdeluxe à Londres:
http://www.okdeluxe.co.uk/cop15/

BODY NAVIGATION

Body navigation est une performance scénique. Elle a été réalisée lors d'une collaboration entre la chorégraphe Tina Tarpgaard et le développeur Jonas Jongejan. Les danseurs sont filmés par une caméra infrarouge qui repère leurs déplacements et des visuels générés par Processing sont instantanément projetés autour d'eux.

Réalisé en 2008 par Jonas Jongejan et Ole Kristensen :
http://3xw.ole.kristensen.name/works/body-navigation/

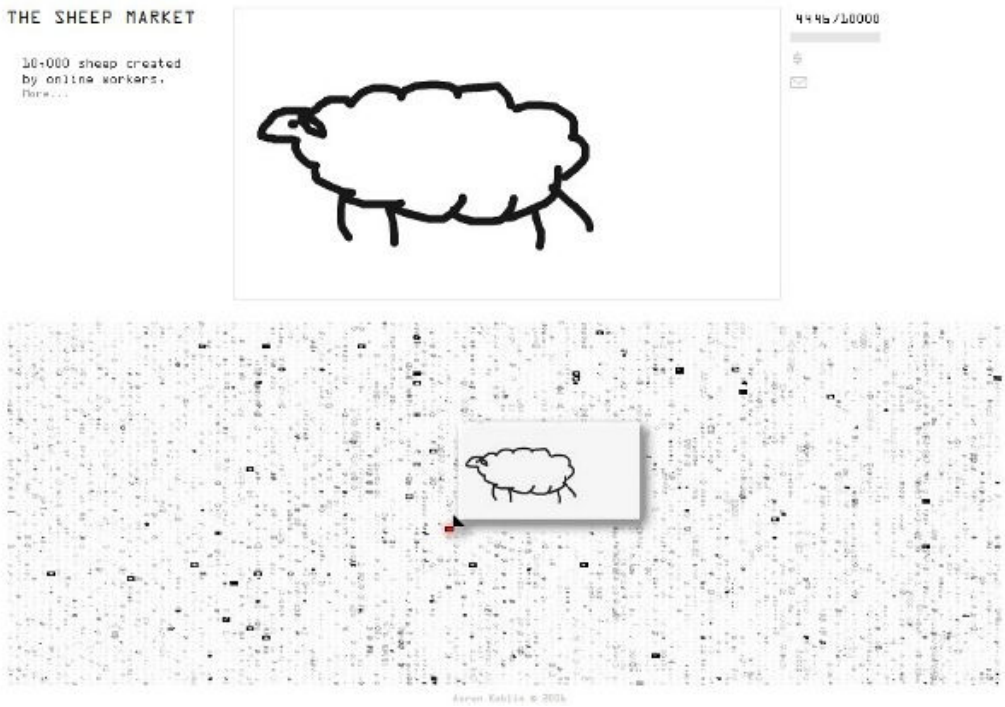
FLIGHT 404



Robert Hodgin explore les possibilités expressives du code, souvent en associant son et algorithme. Il génère de nombreux films avec Processing qu'il distribue sur Internet. Ces expérimentations sont ensuite réutilisées dans divers projets, comme le visualiseur actuellement installé dans iTunes, ou ces papillons générés avec des diagrammes de Voronoi pour une exposition personnelle.

Réalisé en 2007 par Robert Hodgin: http://flight404.com/

THE SHEEP MARKET



Dans le cadre de ce projet artistique, les internautes, jeunes et moins jeunes, ont été invités à dessiner un mouton, référence non dénuée d'humour au *Petit Prince* d'Antoine de Saint-Exupéry. Ces illustrations ont été regroupées dans une immense base de données accessible en ligne. Processing permet de parcourir les 10 000 dessins et de visualiser leur tracé.

Réalisé en 2008 par Aaron Koblin : http://www.thesheepmarket.com

Les images des réalisations présentées dans ce chapitre sont la propriété de leurs auteurs respectifs.

3. L'INSTALLATION DE PROCESSING

Processing étant écrit en Java, il fonctionne en principe sous toute plate-forme supportant au moins la version 5 (également appelée 1.5) de ce langage. Les plates-formes les mieux supportées sont néanmoins Microsoft Windows (Windows XP et Windows 7), GNU/Linux et Mac OS X.

L'installation proprement dite de Processing est assez simple et dépend de la plate-forme que vous utilisez. Dans tous les cas, allez sur la page de téléchargement http://processing.org/download/ et cliquez sur le nom de votre plate-forme.

↓ Linux ↓ Windows
↓ Mac OSX ↓ Windows (Without Java)*

Pour la suite de ce chapitre, dans les noms de fichiers et dossiers, le xxx fait référence à la version de Processing utilisée.

SOUS WINDOWS

Pour Windows, il est préférable de ne pas utiliser l'option *Without JAVA* qui nécessitera d'installer séparément Java. En cliquant sur Windows dans la page de téléchargement du site de Processing, vous allez télécharger une archive *processing-xxx.zip* . Une fois le téléchargement achevé, décompressez l'archive et placez le dossier Processing extrait de l'archive dans le dossier C:\Program Files\.

Allez ensuite dans le dossier C:\Program Files\Processing et exécutez le fichier *processing.exe* en cliquant dessus.

SOUS MAC OS X

La version Mac OS X est téléchargée sous la forme d'une image disque (fichier .dmg). Par défaut ce fichier s'affiche sur le bureau et la fenêtre de l'image disque s'ouvre automatiquement. Au sein de cette fenêtre, il suffit de prendre l'icône Processing et de la glisser sur l'icône du dossier Applications.

Pour exécuter le programme que vous venez d'installer, rendez-vous dans votre dossier *Applications* et double cliquez sur l'icône Processing.

SOUS GNU/LINUX

Après avoir cliqué sur *Linux* pour télécharger le fichier *processing-xxx.tgz* correspondant à cette plate-forme, il suffit de suivre pas à pas la procédure d'installation décrite ci-après.

Enregistrer le fichier sur votre ordinateur

Il est préférable d'enregistrer (ou copier) le fichier téléchargé dans votre répertoire personnel (exemple : /home/MonDossierPersonnel).

Extraire les contenus du fichier

Cette opération peut être réalisée de 2 manières :

- en utilisant votre souris (notamment si vous utilisez la distribution Ubuntu - gnome) : effectuez un clic droit sur le fichier pour faire apparaître le menu contextuel puis cliquez sur *Extraire ici* (*Extract here*).
- pour les plus experts, saisissez et exécutez dans un terminal la commande suivante :

```
tar -zxvf processing-xxx.tgz
```

Dans les deux cas, un dossier *processing-xxx* sera créé dans votre répertoire personnel, le xxx faisant référence version du logiciel. Ainsi dans le cas de la version 1.2.1 de Processing le nom du dossier s'appellera *processing-1.2.1* et le chemin pour accéder à ce répertoire sera /home/MonDossierPersonnel/processing-1.2.1.

Installer Processing

- Pour installer Processing, en mode graphique, il suffit de se placer dans le dossier créé et d'effectuer un double clic.
- Si par contre vous utilisez un terminal, saisissez la commande :

```
cd /home/VotreDossierPersonnel/Processing-xxx/
```

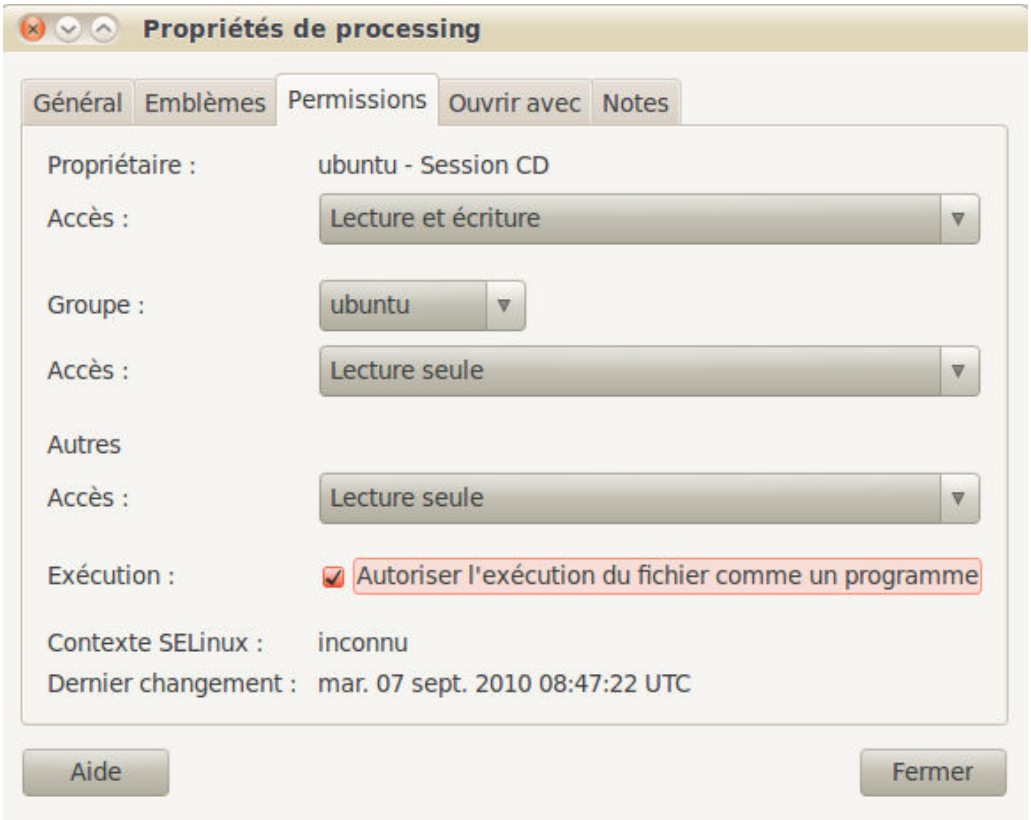
Autoriser l'exécution de Processing sur votre ordinateur

Après l'installation du logiciel, il faut s'assurer que le fichier *processing* contenu dans le dossier du programme (*/home/MonDossierPersonnel/Processing*-xxx) est bien exécutable. Ce n'est pas le cas par défaut.

- Dans un terminal, exécutez la commande suivante :

```
chmod +x processing
```

- En mode graphique, effectuez un clic droit sur le fichier *processing* puis dans le menu contextuel qui apparaît, cliquez sur *propriétés*. La boîte de dialogue suivante s'affichera :



Cliquez sur l'onglet *Permissions* puis cocher la case *Autoriser l'exécution du fichier comme un programme*.

Lancer Processing

- Pour démarrer le programme Processing dans le terminal (en étant toujours dans le dossier du programme, exemple /home/MonDossierPersonnel/processing-1.2.1/), lancer la commande :

```
./processing
```

- En mode graphique, faites un double clic sur le fichier *processing* puis dans la boîte de dialogue qui apparaît, cliquez sur le bouton *Lancer*.

Installer Java, si besoin

Pour fonctionner, Processing requiert Java (au moins la version 5, également appelée 1.5) originaire de Sun Microsystems.

Si Java n'est pas installé, vous pourrez télécharger et installer openjdk en vous rendant sur le site http://openjdk.java.net/, ou mieux, en utilisant le système de paquetage de votre distribution de GNU/Linux.

Sous Ubuntu, exécuter la commande suivante (notez bien que cette étape est préalable à l'installation de JRE présentée un peu plus loin) :

```
sudo apt-get install openjdk-6-jdk
```

Ensuite, il se peut que vous ayez à changer la version de Java qui est proposée par défaut. Pour vous en assurer, saisissez la commande suivante :

```
sudo update-alternatives --config java
```

Une liste de choix devrait alors apparaître. Entrez le chiffre qui se trouve sur la même ligne que le chemin qui contient le mot «sun», et appuyer sur entrée. Dans l'exemple présenté ci-dessous, la version présélectionnée est identifiée par un astérisque situé au début d'une des lignes. Il faudrait donc saisir le chiffre 4 pour sélectionner la bonne version.

```
Il existe 4 choix pour l'alternative java (qui fournit /usr/bin/java).
  Sélection   Chemin                                    Priorité  État
------------------------------------------------------------
* 0            /usr/lib/jvm/java-6-openjdk/jre/bin/java   1061      mode automatique
  1            /usr/bin/gij-4.4                           1044      mode manuel
  2            /usr/lib/jvm/java-6-cacao/jre/bin/java     1059      mode manuel
  3            /usr/lib/jvm/java-6-openjdk/jre/bin/java   1061      mode manuel
  4            /usr/lib/jvm/java-6-sun/jre/bin/java       63        mode manuel
Appuyez sur <Entrée> pour conserver la valeur par défaut[*] ou choisissez le numéro
sélectionné :
```

4. LES BASES DE PROCESSING

Processing propose à la fois un environnement de création complet et un ensemble de fonctionnalités supplémentaires qui viennent enrichir les possibilités du logiciel. Cet environnement permet d'écrire des programmes (appelés sketchs dans Processing), de les convertir en fichiers autonomes, de les publier ainsi que d'identifier et de corriger les erreurs. Il contient les fonctions essentielles à la programmation tout en étant simple d'utilisation.

Processing est basé sur le langage Java. C'est cette syntaxe qui sera utilisée lorsque vous allez programmer. Processing vous facilite la maîtrise de ce langage en se chargeant de manière transparente des opérations relativement complexes comme gérer les fenêtres, le son, la vidéo, la 3D et bien d'autres choses encore. Ce logiciel propose une large palette de fonctionnalités prédéfinies qui simplifie la conception de programmes créatifs, notamment pour les personnes ne maîtrisant pas les notions complexes de programmation et de mathématiques.

Ce chapitre vous présente les bases de l'interface de Processing et les notions minimales de la syntaxe Java à connaître pour bien débuter.

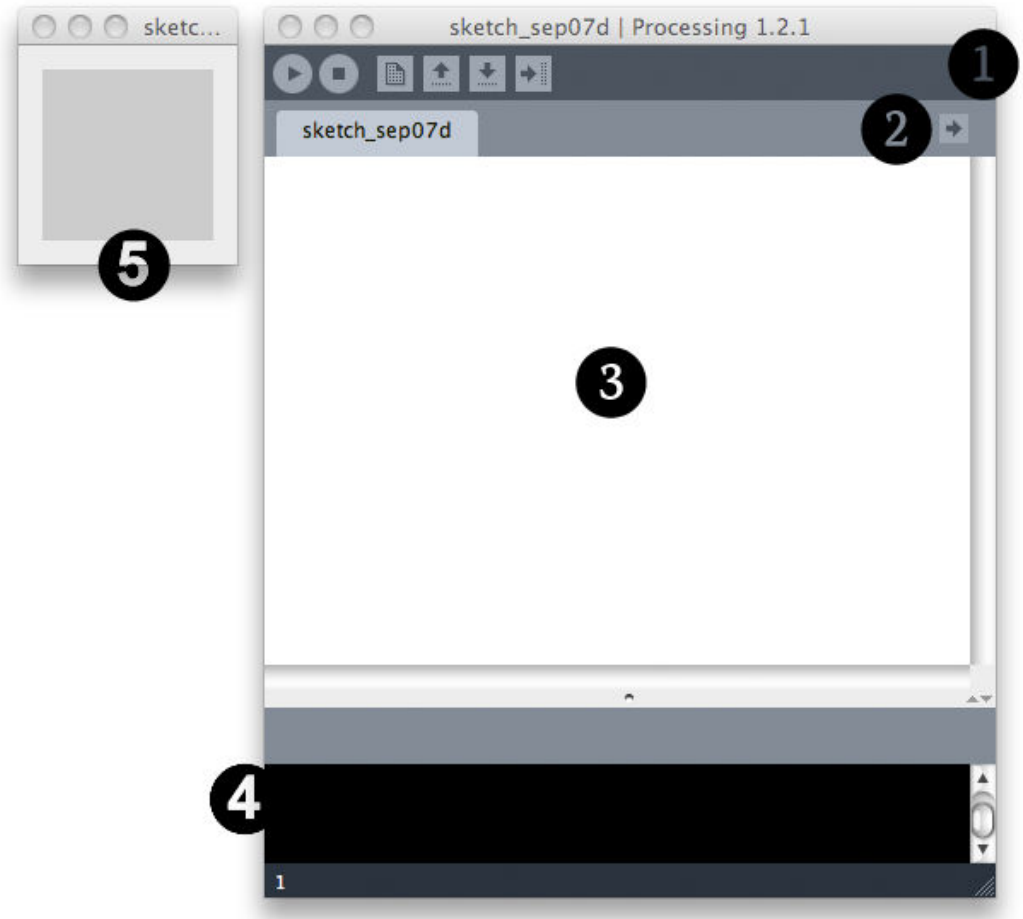
A ce jour, l'interface de Processing est disponible uniquement en anglais. Malgré cela, son utilisation reste simple et il est très probable que les versions à venir proposeront une traduction en plusieurs langues, dont le français. Les nombreux atouts de ce logiciel libre le rendent suffisamment incontournable pour commencer à apprendre à l'utiliser sans plus attendre.

L'INTERFACE

L'interface d'utilisation de Processing est composée de deux fenêtres distinctes : la fenêtre principale dans laquelle vous allez créer votre projet et la fenêtre de visualisation dans laquelle vos créations (dessins, animations, vidéos) apparaissent.

On trouve plus précisément les éléments suivants dans l'interface :

1. Barre d'actions
2. Barre d'onglets
3. Zone d'édition (pour y saisir votre programme)
4. Console (destinée aux tests et messages d'erreur)
5. Fenêtre de visualisation (espace de dessin)
6. Barre de menu (non représentée sur la copie d'écran)



Barre d'actions

Bouton "Run" : exécute votre sketch (votre programme).

Bouton "Stop" : arrête l'exécution de votre sketch.

Bouton "New" : Crée un nouveau sketch.

Bouton "Open" : ouvre un sketch existant.

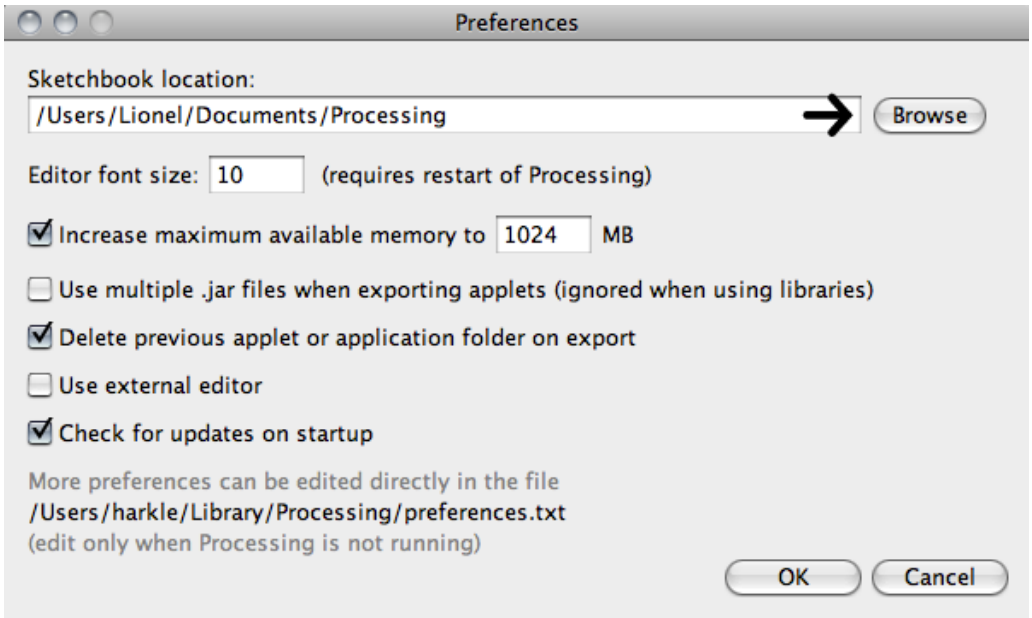
Bouton "Save" : sauvegarde le sketch en cours.

Bouton "Export" : exporte le sketch pour le web.

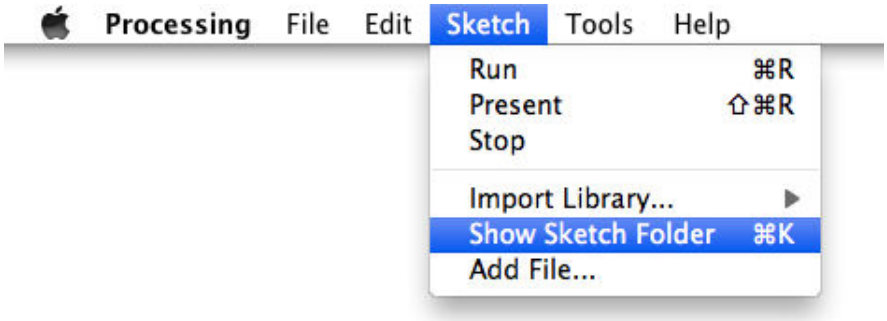
Le dossier de travail

C'est le dossier dans lequel seront enregistrés les sketchs et les librairies (des modules externes proposant des fonctionnalités supplémentaires). Par défaut ce dossier se nomme *Processing* et se trouve dans *Documents* (sous Mac) ou *Mes Documents* (sous Windows). Sous GNU/Linux, il est dans votre dossier personnel sous le nom de *sketchbook*.

Pour modifier ce dossier, allez dans le menu *Files › Preferences*. Dans la boîte de dialogue qui apparaît, cliquez sur *Browse* (parcourir) pour choisir le dossier qui vous convient.



A tout moment, pour savoir quel est votre dossier de travail, sélectionnez, dans le menu *Sketch › Show Sketch Folder*. Cette option est également accessible via le raccourci ctrl-k sous Windows/Linux ou cmd-k sur Mac :



BASES DU LANGAGE

Processing utilise le langage Java pour créer vos programmes. Ce langage, qui va être lu par votre ordinateur après avoir cliqué sur le bouton de lancement du sketch, possède un certain nombre de règles de syntaxe qui si elles ne sont pas respectées empêcheront l'exécution correcte du programme. Il y a aussi un certain nombre de concepts de base nécessaires à connaître.

Majuscules et minuscules

Processing est sensible à la casse, il fait la différence entre les majuscules et les minuscules : **l**ibre est différent de **L**ibre !

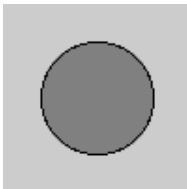
Le point virgule

A la fin de chaque instruction (dessiner un cercle, effectuer un calcul, etc.), vous devez mettre un « ; » afin de signaler à l'ordinateur la fin de l'instruction. Dans l'exemple ci-dessous, on utilise les caractères « // » pour insérer un commentaire qui sera ignoré lors de l'exécution (la présence de commentaires dans votre programme facilite sa compréhension ultérieure).

```
//Dessine un cercle
ellipse(10,10, 10, 10);

//Crée une variable
int chiffre = 10 + 23;
```

Appels de méthodes

Processing propose un grand nombre de fonctionnalités prédéfinies appelées méthodes : dessiner un rectangle, définir une couleur, calculer une racine carrée, etc. Ces méthodes ont chacune un nom spécifique. Pour faire appel à elles, il suffit de taper leur nom en respectant les majuscules et minuscules et de coller des parenthèses après le nom : parfois on doit préciser certaines valeurs à l'intérieur des parenthèses (couleur, position, taille, etc.). L'exemple ci-dessous affiche un cercle gris.



```
fill(128);
ellipse(50, 50, 60, 60);
```

Affichage dans la console

La console (la zone 4 dans le schéma de l'interface présenté en début de ce chapitre) permet d'afficher du texte brut à des fins de test et de débogage (correction d'erreurs). Pour afficher quelque chose dans cette zone, il faut utiliser la méthode `println();`

```
println("Salut tout le monde!");
```

```
Salut tout le monde!
```

```
println(1000);
```

```
1000
```

Par extension, le terme console sert également à désigner la bande grise juste au dessus de la zone noire de test/débogage : Processing y affiche un certain nombre de messages, notamment pour signaler des erreurs.

Opérations arithmétiques

Processing permet d'effectuer des calculs mathématiques. Tout au long de l'utilisation de cet environnement, vous serez amené à calculer des valeurs. Ne vous inquiétez pas : l'ordinateur le fera pour vous. Les opérations d'addition, soustraction, multiplication et division peuvent être combinées. Il est également possible d'utiliser des parenthèses pour définir l'ordre des opérations.

Attention aux nombres à virgule ! Dans Processing, les unités sont séparées des décimales par un point et non par une virgule.

Voici quelques exemples d'opérations arithmétiques :

```
println(10 + 5);
println(10 + 5 * 3); // 5*3 (soit 15) puis additionne 10
println((10 + 5) * 3); // 10+5 (soit 15) puis multiplie 15 par 3
println(10.4 + 9.2);
```

Cette suite d'instructions saisie dans la fenêtre d'édition de Processing va produire dans la console le résultat suivant :

```
15
25
45
19.599998
```

Certaines opérations arithmétiques peuvent être contractées. Par exemple, `i++` donne le même résultat que `i = i + 1`. Et x+=10 donne le même résultat que x=x+10.

Maintenant que vous connaissez les bases de l'interface de Processing, il ne vous reste plus qu'à apprendre à écrire du code qui fonctionne pour dessiner et créer ce que vous souhaitez. C'est l'objet des prochains chapitres.

DESSINER

5. L'espace de dessin
6. Les formes
7. Les couleurs
8. Le texte

5. L'ESPACE DE DESSIN

L'espace de dessin constitue l'espace de représentation proprement dit. Cette fenêtre de visualisation affichera vos réalisations dans Processing en 2 ou 3 dimensions.

Cet espace est créé par l'instruction `size()` qui prend deux arguments : `size(largeur,hauteur);`.

Par exemple, dans la fenêtre d'édition du logiciel Processing, saisissez la commande suivante :

Puis cliquez sur le bouton **run**, votre fenêtre de visualisation se créé :

Amusez-vous à changer les dimensions de cette fenêtre en modifiant les valeurs entre parenthèses pour en voir le résultat.

Par défaut :

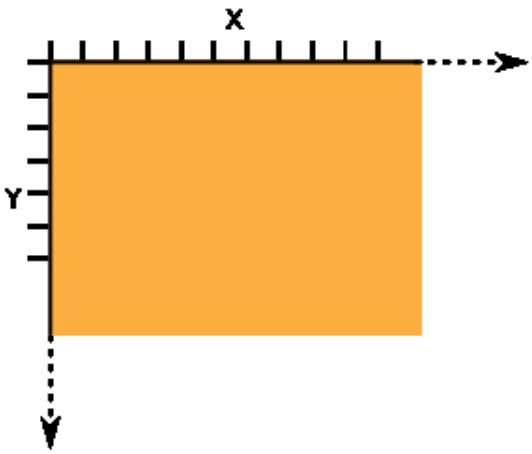
```
size();
```

...affichera une fenêtre de 100 pixels sur 100 pixels.

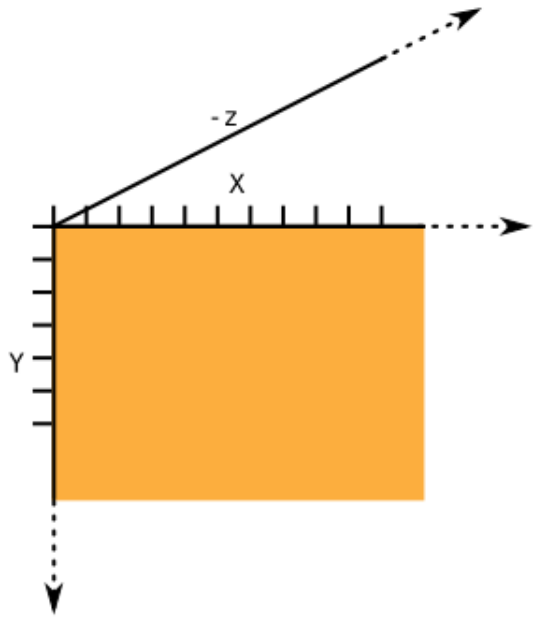
Félicitations, vous avez créé votre première fenêtre de visualisation !

COORDONNÉES DANS L'ESPACE

Quand on travaille en 2 dimensions (2D), on utilise deux axes de coordonnées x et y correspondant respectivement à la largeur (axe horizontal) et à la hauteur (axe vertical) d'une situation. Par convention de la mesure de l'espace, le coin en haut à gauche correspond aux valeurs x=0 et y=0. Les valeurs x sont croissantes vers la droite et les valeurs y sont croissantes vers le bas, contrairement à notre habitude du plan cartésien. Ces valeurs x et y peuvent s'étendre théoriquement à l'infini, même si, en réalité, les contraintes de la taille de votre fenêtre vont délimiter la taille maximale d'une surface de création visible. C'est donc dans cet espace que nous allons dessiner.



Quand on travaille en 3 dimensions (3D), en plus des deux axes de coordonnées, on a un troisième axe de coordonnées Z, exprimant la profondeur :



Dans ce cas précis, on utilise la commande `size` avec un troisième paramètre indiquant que l'on travaille dans un espace en 3D `size(100, 100, P3D);`

CONNAÎTRE LA TAILLE DE L'ESPACE DE DESSIN

Au sein d'un programme, on peut connaître à tout moment la taille de l'espace de dessin utilisé au moyen des mots-clés `width` et `height`. Ces instructions sont très utiles lorsque l'on souhaite notamment dessiner des formes qui puissent s'adapter ultérieurement aux éventuels changements de dimension de la fenêtre de visualisation.

6. LES FORMES

Beaucoup de formes prédéfinies sont fournies par Processing. En voici les principales :

LE POINT

Pour commencer à dessiner, nous allons partir d'un point. Sur l'écran, un point est l'équivalent d'un pixel localisé dans la fenêtre de visualisation par deux axes de coordonnées, x et y correspondant respectivement à la largeur (axe horizontal) et à la hauteur (axe vertical) de l'espace de dessin. En suivant ce principe, la création d'un point dans Processing s'effectue à l'aide de l'instruction `point(x,y)`. Dans cet exemple, le point est très petit. Il est placé au centre de la fenêtre de visualisation.

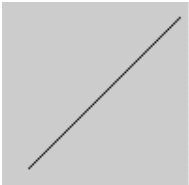
```
point(50, 50);
```


Notez que le cadre de la fenêtre de visualisation (l'espace de dessin) a une dimension de 100x100, ce qui explique que le point soit situé en plein milieu. Si on le dessinait en dehors du cadre (hors champ), avec par exemple l'instruction `size(150,150)`, on ne le verrait pas.

LA LIGNE

Par définition, une ligne (AB) est constituée par une infinité de points entre un point de départ A et un point d'arrivée B. Pour la construire, nous allons nous intéresser uniquement aux coordonnées x et y de A et de B. Ainsi, si par exemple dans la fenêtre par défaut, le point A se situe dans la région en bas à gauche de votre fenêtre, et que le point B se situe en haut à droite, les instructions suivantes, peuvent dessiner cette ligne sous la forme `line(xA,yA,xB,yB)` :

```
line(15, 90, 95, 10);
```

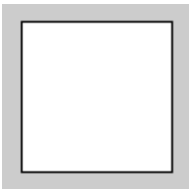


LE RECTANGLE

Un rectangle se dessine par quatre valeurs en faisant l'appel de `rect(x,y,largeur,hauteur)`. La première paire de valeurs `x` et `y`, par défaut (mode `CORNER`) correspond au coin supérieur gauche du rectangle, à l'instar du **point**. En revanche la seconde paire de valeurs ne va pas se référer à la position du coin inférieur droit, mais à la largeur (sur l'axe des x, horizontal) et à la hauteur (sur l'axe des y, vertical) de ce rectangle.

Exemple :

```
rect(10, 10, 80, 80);
```



Comme les deux dernières valeurs (largeur et hauteur) sont identiques, on obtient un carré. Amusez-vous à changer les valeurs et observez-en les résultats.

Pour que la première paire de valeurs corresponde au centre (le croisement des deux diagonales aux coordonnées 50, 50) du rectangle, il faut utiliser le mode `CENTER` comme suit :

```
rectMode(CENTER);
rect(50, 50, 80, 40);
```

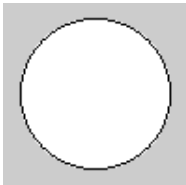
Cela donne le résultat identique à l'exemple précédent.

L'ELLIPSE

Comme pour le rectangle, l'ellipse se construit sous les modes `CENTER` (par défaut), et `CORNER`. Ainsi l'instruction suivante produit un cercle dont les coordonnées du centre sont les deux premières valeurs entre parenthèses. La troisième valeur correspond à la grandeur du diamètre sur l'axe horizontal (x) et la quatrième à la grandeur du diamètre sur l'axe vertical : notez que si les 3e et 4e valeurs sont identiques, on dessine un cercle et dans le cas contraire, une ellipse quelconque :

```
ellipse(50, 50, 80, 80);
```

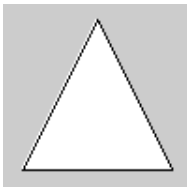
Amusez-vous à faire varier les 3e et 4e valeurs et observez-en les résultats.



LE TRIANGLE

Le triangle est un plan constitué de trois points. L'invocation de `triangle(x1,y1,x2,y2,x3,y3)` définit les trois points de ce triangle :

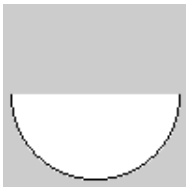
```
triangle(10, 90, 50, 10, 90, 90);
```



L'ARC

Un arc ou section de cercle, peut se dessiner avec l'appel de `arc(x, y, largeur, hauteur, début, fin)`, où la paire x, y définit le centre du cercle, la seconde paire ses dimensions et la troisième paire, le début et la fin de l'angle d'arc en radians :

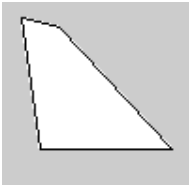
```
arc(50, 50, 90, 90, 0, PI);
```



LE QUADRILATÈRE

Le quadrilatère se construit en spécifiant quatre paires de coordonnées x et y sous la forme `quad(x1,y1,x2,y2,x3,y3,x4,y4)` dans le sens horaire :

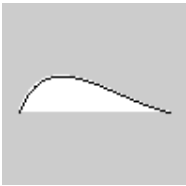
```
quad(10, 10, 30, 15, 90, 80, 20, 80);
```



COURBE

Une courbe se dessine à l'aide de `curve(x1, y1, x2, y2, x3, y3, x4, y4)`, où x1 et y1 définissent le premier point de contrôle, x4 et y4 le second point de contrôle , x2 et y2 le point de départ de la courbe et x3, y3 le point d'arrivée de la courbe :

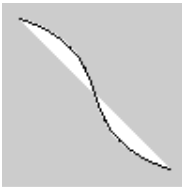
```
curve(0, 300, 10, 60, 90, 60, 200, 100);
```



COURBE BÉZIER

La courbe de type Bézier se construit à l'aide de
`bezier(x1,y1,x2,y2,x3,y3,x4,y4)`

```
bezier(10, 10, 70, 30, 30, 70, 90, 90);
```



COURBE LISSÉE

L'appel de `curveVertex()` dessine plusieurs paires de coordonnées x et y, entre deux points de contrôle, sous la forme `curveVertex(point de contrôle initial,xN,yN,xN,yN,xN,yN, point de contrôle final)` ce qui permet de construire des courbes lissées :

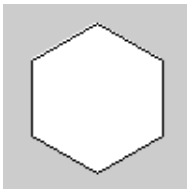
```
beginShape();
curveVertex(0, 100);
curveVertex(10, 90);
curveVertex(25, 70);
curveVertex(50, 10);
```

```
curveVertex(75, 70);
curveVertex(90, 90);
curveVertex(100, 100);
endShape();
```

FORMES LIBRES

Plusieurs formes libres peuvent être dessinés par une succession de points en utilisant la suite d'instructions `beginShape(),vertex(x,y),...,endShape()`. Chaque point se construit par ses coordonnées x et y. La fonction `CLOSE` dans `endShape(CLOSE)` indique que la figure sera fermée, c'est-à-dire que le dernier point sera relié au premier, comme dans l'exemple ci-dessous de dessin d'un hexagone :

```
beginShape();
vertex(50, 10);
vertex(85, 30);
vertex(85, 70);
vertex(50, 90);
vertex(15, 70);
vertex(15, 30);
endShape(CLOSE);
```



CONTOURS

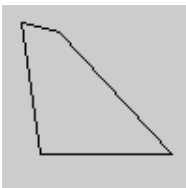
Vous avez remarqué que jusqu'à présent, toutes les figures données en exemple comportent un contour, ainsi qu'une surface de remplissage. Si vous voulez rendre invisible le contour, utilisez `noStroke()` en faisant bien attention de le placer avant la forme à dessiner :

```
noStroke();
quad(10, 10, 30, 15, 90, 80, 20, 80);
```

REMPLISSAGE

De la même manière, il est possible de dessiner des formes sans surface de remplissage avec l'instruction `noFill()` :

```
noFill();
quad(10, 10, 30, 15, 90, 80, 20, 80);
```



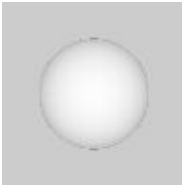
Par défaut, l'arrière-fond de la fenêtre de visualisation (l'espace de dessin) est gris neutre, les contours des figures sont noirs, et la surface de remplissage est blanche. Vous apprendrez au prochain chapitre comment modifier les couleurs à votre convenance.

PRIMITIVES 3D

Les formes prédéfinies disponibles en 3 dimensions (primitives 3D) peuvent être réalisées de manière simple en appelant `size(x, y, P3D)` au début de notre sketch puis en employant en fonction de vos besoins les instructions `sphere(taille)` et `box(longueur, largeur, profondeur)`. Il est également possible de produire des effets d'éclairage sur nos formes tridimensionnelles à l'aide de `lights()`.

La sphère

```
size(100, 100, P3D);
noStroke();
lights();  // éclairer l'objet 3D
translate(50, 50, 0);  // voir Chapitre "Transformations"
sphere(28);
```



La boîte

```
size(100, 100, P3D);
noStroke();
lights();
translate(50, 50, 0);
rotateY(0.5);
box(40);
```

7. LES COULEURS

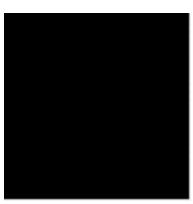
Dessiner une image à l'écran, c'est changer la couleur des pixels. Les pixels sont des petites zones, le plus souvent carrées, qui possèdent une couleur. Chaque couleur se définit par trois canaux qui sont le rouge, le vert et le bleu. Une valeur de 100% de chacun de ces trois canaux donne du blanc. Une valeur de 0% de chacun de ces trois canaux donne du noir. Il s'agit de lumière, et non de peinture. Ainsi, plus la valeur de chaque canal sera importante, plus la couleur sera lumineuse.

Par exemple, 100% de rouge, 80% de vert et 20% de bleu vont donner ensemble la couleur orange. La méthode `fill()` nous permet de spécifier la couleur des prochaines formes à dessiner. Chaque canal d'une couleur peut être donné sur une échelle de 0 à 255. Ainsi, 80% de 255 donne 204, et 20% de 255 donne 51.

LA COULEUR DE FOND

On peut changer la couleur de fond en appelant la méthode `background()`. Attention : rajouter `background()` à la suite d'une composition déjà existante, l'effacerait !

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



LA COULEUR DE REMPLISSAGE

A chaque fois que l'on dessine une forme, on le fait avec la couleur de remplissage qui est choisie à ce moment-là. On le fait en appelant la méthode `fill()`.

```
noStroke();
fill(255, 204, 51);
rect(25, 25, 50, 50);
```

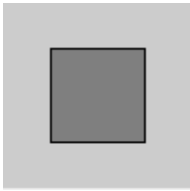
Processing nous offre différents formats pour exprimer la couleur. Si vous faites de la programmation Web, vous connaissez sans doute le format hexadécimal. Selon ce procédé, la même couleur orange peut être obtenue en écrivant :

```
fill(#ffcc33);
rect(25, 25, 50, 50);
```

Par ailleurs, il est possible de spécifier la valeur du canal alpha de la couleur utilisée, c'est-à-dire son degré de transparence. Pour ce faire, on doit donc préciser quatre paramètres à la méthode `fill()`. Le quatrième argument est la valeur alpha.

```
noStroke();
fill(255, 204, 51); // orange
rect(25, 25, 50, 50);
fill(255, 255, 255, 127); // blanc semi-transparent
rect(35, 35, 50, 50);
```


Si l'on souhaite choisir une couleur qui correspond à un ton de gris, il suffit de donner un seul paramètre à la méthode `fill()`. C'est la valeur de gris, de 0 à 255.

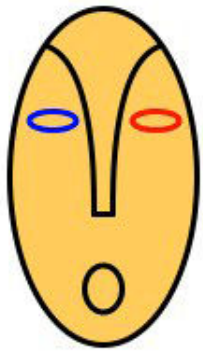


```
fill(127);
rect(25, 25, 50, 50);
```

On peut désactiver le remplissage des formes en appelant la méthode `noFill()`.

LA COULEUR DU CONTOUR

Pour changer la couleur du contour des formes que l'on dessine, on doit appeler la méthode `stroke()` avec comme paramètres les canaux de la couleur désirée. Appeler `noStroke()` désactive la couleur de contour. A titre d'illustration, voici un dessin de masque africain utilisant ces deux instructions :

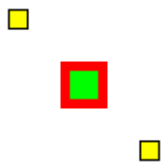


```
size(200, 200);
smooth();
background(255); // on dessine un fond blanc

stroke(#000000); // le contour sera noir
fill(#FFCC66);  // le remplissage sera jaune
strokeWeight(3);

translate(width / 2, height / 2);

ellipse(0, 0, 100, 180); // forme elliptique du masque

ellipse(0, 60, 20, 25); // ellipse de la bouche

stroke(255, 0, 0); // le contour sera rouge
ellipse(28, -30, 25, 10);  // ellipse de l'oeil droit

stroke(0, 0, 255); // le contour sera bleu
ellipse(-27, -30, 25, 10); // ellipse de l'oeil gauche

noFill(); // les prochaines formes n'auront pas de remplissage

stroke(#000000);  // le contour des prochaines formes sera noir

bezier(-30, -70, -5, -60, -5, 0, -5, 20); // courbe du sourcil droit

bezier(30, -70, 5, -60, 5, 0, 5, 20); // courbe du sourcil gauche
```

```
line(-5, 20, 5, 20); // ligne du nez pour joindre l'extrémité des courbe
```

LA PORTÉE DES MODIFICATIONS DE COULEUR

Par défaut, toute modification de style (couleur de remplissage ou de contour, épaisseur ou forme de trait) s'appliquera à tout ce que vous dessinerez ensuite. Pour limiter la portée de ces modifications, vous pouvez les encadrer par les commandes `pushStyle()` et `popStyle()`.



```
size(100, 100);
background(255);

stroke(#000000);
fill(#FFFF00);
strokeWeight(1);
rect(10, 10, 10, 10);

pushStyle(); // On ouvre « une parenthèse » de style

stroke(#FF0000);
fill(#00FF00);
strokeWeight(5);
rect(40, 40, 20, 20);

popStyle(); // On ferme notre parenthèse de style

rect(80, 80, 10, 10);
```

Exercice :

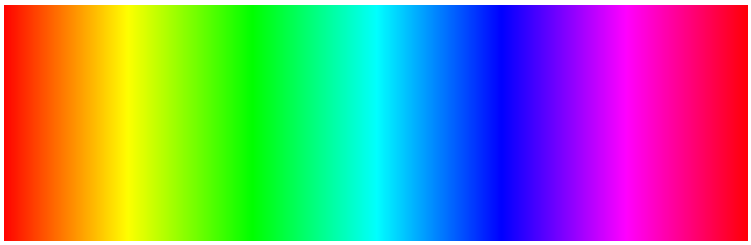
Supprimez les commandes `pushStyle()` et `popStyle()` et observez la différence de comportement.

L'ESPACE COLORIMÉTRIQUE

Définir des couleurs à partir des canaux rouge, vert et bleu constitue un moyen parmi d'autres de décrire l'espace colorimétrique de votre dessin. Processing supporte également le mode TSV. TSV signifie « teinte, saturation, valeur ». En anglais, on appelle ce mode HSB, pour « hue, saturation, brightness ». On choisit une échelle de 0 à 100 pour chacun de ces 3 canaux. La teinte correspond à un chiffre indiquant la position de la couleur sur l'échelle chromatique, soit un arc-en-ciel. Le rouge est à gauche, puis viens l'orange, le jaune, le vert, le bleu et le violet.

La méthode `colorMode()` est utile pour changer l'échelle numérique que l'on utilise pour spécifier les couleurs, et pour changer d'espace colorimétrique. Par exemple, appeler `colorMode(RGB, 1.0)`, va changer l'échelle que l'on utilise pour spécifier chaque canal des couleurs afin qu'il aille de zéro à un.

Ici, on change le mode de couleur pour le TSV afin de créer un dégradé de teinte qui ressemble aux couleurs d'un arc-en-ciel.



```
noStroke();
size(400, 128);

// La teinte sera spécifiée avec un chiffre de 0 à 400
colorMode(HSB, 400, 100, 100);

// On fait quatre cent répétitions
for (int i = 0; i < 400; i++) {
  fill(i, 128, 128);
  rect(i, 0, 1, 128);
}
```

8. LE TEXTE

On va maintenant dessiner des caractères textuels qui s'afficheront dans la fenêtre de visualisation. Attention : écrire dans l'espace de dessin ne ressemble pas à l'écriture dans le sens classique du terme, notamment comme dans un traitement de texte. Le texte dans Processing ressemblerait plutôt à du « graffiti urbain », c'est-à-dire à une sorte de peinture de caractères alphanumériques qui finiront tous par s'empiler les uns sur les autres. C'est davantage un procédé « typo-graphique » qu'un travail d'écriture, avec une insistance sur l'aspect graphique des mots ou des lettres que nous allons dessiner.

BONJOUR PROCESSING !

Tout comme les formes prédéfinies, telles les lignes ou les ellipses, il suffit d'une seule instruction dans notre sketch pour dessiner du texte. Ouvrez une nouvelle fenêtre d'édition, tapez la ligne suivante, et exécutez-la en cliquant sur le bouton run :

```
text("Salut!", 10, 20);
```

Vous devriez voir s'afficher la petite fenêtre de visualisation par défaut de 100 x 100 pixels, avec le mot « »Salut ! » écrit en blanc :

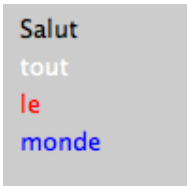
Comme son nom l'indique, l'instruction `text()` dessine du texte dans la fenêtre de visualisation. Elle requiert trois paramètres : {le message que nous voulons écrire}, {sa coordonnée x}, {sa coordonnée y}.

Ce texte peut également être coloré, comme n'importe quelle forme géométrique, en changeant la couleur de remplissage :

```
fill(0);
text("Salut", 10, 20);

fill(255);
```

```
text("tout", 10, 40);

fill(255,0,0);
text("le", 10, 60);

fill(0,0,255);
text("monde", 10, 80);
```

Ce programme devrait vous donner le résultat suivant, avec chacun des quatre mots écrits avec leur propre couleur :



Il suffit d'indiquer une couleur, puis dessiner du texte avec cette couleur. Bien sûr vous pouvez ne choisir qu'une seule couleur et dessiner plusieurs messages avec cette couleur, ou dessiner avec une combinaison de couleurs comme dans cet exemple.

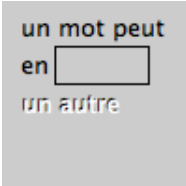
EMPILEMENT

Comme nous l'avons indiqué dans l'introduction de ce chapitre, les mots s'inscrivent à l'intérieur de la fenêtre de visualisation tel un dessin et non pas tel un système de traitement de texte. On ne peut ni sélectionner ni modifier directement ce texte à l'aide de la souris ou du clavier. La seule modification qui peut être appliquée s'effectue au niveau de votre programme en précisant à l'aide d'instructions appropriées que vous souhaitez par la suite effacer complètement le texte et d'écrire par dessus.

Voici une illustration rapide de ce principe de superposition :

```
fill(0);
text("un mot peut", 10, 20);
text("en cacher", 10, 40);

fill(204);
rect(28,25,50,20);

fill(0);
text("un autre", 10, 60);

fill(255, 255, 255);
text("un autre", 11, 61);
```

Nous avons écrit quatre messages, parfois avec la même couleur, parfois avec une autre. Notez que le mot « cacher » a été caché justement par la couleur de remplissage qui a été définie avec la même valeur que celle utilisée pour dessiner le fond.



Comme dans n'importe quel dessin, l'ordre des opérations est important pour déterminer le rendu final. Ce principe s'applique même en dehors du logiciel Processing : « s'habiller, sortir, aller au travail » ne donnera pas le même résultat que « sortir, aller au travail, s'habiller ».

DESSINER PLUS

9. Les images
10. Les styles de bordures
11. La typographie
12. Les transformations

9. LES IMAGES

Ce que nous appelons « image » dans Processing n'est en fait rien d'autre qu'une collection de pixels, rassemblés à l'intérieur d'un rectangle. Pour dessiner une image à l'écran, nous devons donner une couleur à chacun des pixels d'un rectangle, puis donner la position en {*x*,*y*} où nous voulons dessiner cette collection de pixels. Il est aussi possible de modifier la taille {*largeur*,*hauteur*} de notre image, même si ces dimensions ne correspondent pas à la taille originelle de l'image.

TROUVER UNE IMAGE

Pour dessiner une image dans Processing, il faut commencer par trouver une image et l'importer dans notre sketch. Vous pouvez prendre une photo à l'aide de votre appareil numérique ou directement depuis votre webcam ou bien encore effectuer une recherche d'images se trouvant déjà dans le disque dur de votre ordinateur. Pour cet exercice, l'origine de l'image importe peu. Par contre, nous recommandons de commencer avec une image relativement petite, par exemple d'une largeur de 400 x 300 pixels.

Ici nous allons commencer avec une image légèrement réduite de l'île des peupliers à Ermenonville, trouvée sur le site Commons de Wikimedia (base de données d'images

et de médias appartenant au domaine public ou sous licence libre) à l'adresse suivante : http://fr.wikipedia.org/wiki/Fichier:Erm6.JPG

FORMATS D'IMAGE

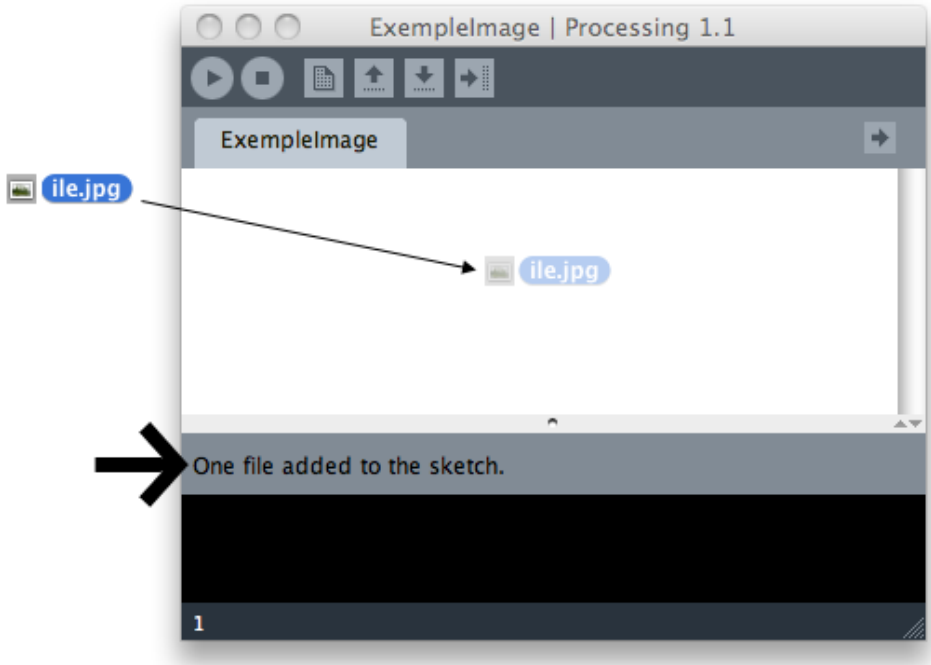
Trois formats de fichier d'image sont acceptés dans Processing : PNG, JPEG, ou GIF. Ceux qui ont de l'expérience dans la conception de sites web devraient reconnaître ces trois formats, car ce sont les plus répandus sur la Toile. Chaque format utilise ses propres méthodes de **compression** de l'image (réduction de la taille mémoire de l'image occupée sur l'ordinateur), lui donnant à la fois des avantages et des désavantages et que nous pouvons résumer de la manière suivante :

1. **JPEG** est souvent utilisé pour compresser des images de type photographique. Ce format ne permet pas d'avoir des zones transparentes.
2. **GIF** est historiquement utilisé pour l'illustration de boutons et autres éléments graphiques de l'espace de travail d'un programme. Sur les sites internet, ce format est utilisé pour les logos et de manière générale pour les dessins réalisés par ordinateur (notamment ceux qui comportent des aplats de couleurs). Ce format peut contenir des zones transparentes binaires (soit opaques soit transparentes, sans possibilité d'opacité intermédiaire). Il existe des images gif animées, mais Processing ignorera cet aspect.
3. **PNG** est de plus en plus utilisé pour les deux usages (photos + dessins) et peut contenir des zones transparentes non binaires, offrant des niveaux d'opacité (opaque, semi-opaque, totalement transparent).

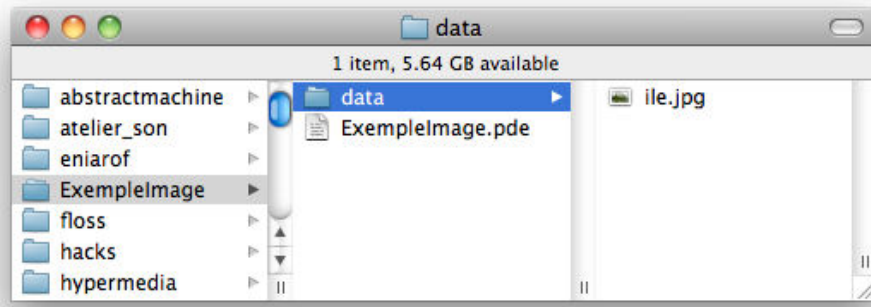
GLISSER-DÉPOSER

Nous allons maintenant importer le fichier de cette image dans l'environnement Processing. Pour bien réussir cette étape, nous vous recommandons de sauvegarder d'abord votre sketch, de préférence dans le dossier *processing* qui devrait se trouver par défaut dans le dossier *Documents* ou *Mes documents* de votre ordinateur.

Localisez maintenant votre fichier d'image, et glissez-le directement sur la fenêtre Processing :

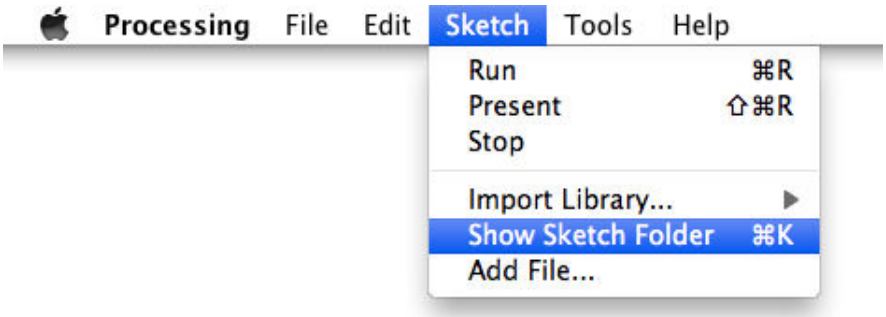


Lorsque nous glissons les fichiers de cette manière, Processing nous indique dans la bande grisée de la console qu'il a ajouté ce fichier « dans le sketch ». En réalité, Processing a simplement copié cette image dans un dossier nommé *data*, qui devrait maintenant se trouver à côté de votre *programme.pde* que vous venez de sauvegarder.



C'est dans ce dossier *data* que nous devons placer en réalité toutes les images dont nous voulons nous servir dans notre sketch. C'est dans cet emplacement que doivent également être rangés les autres fichiers médias comme les polices ou les sons.

Pour accéder rapidement à ce dossier, afin par exemple d'y placer de nouveaux fichiers d'images, il suffit de sélectionner, dans le menu *Sketch*, l'option *Show Sketch Folder*. Cette option est également accessible via le raccourci Ctrl-k sur Windows et GNU/Linux ou Cmd-k sur Mac :



IMPORTER UNE IMAGE

Maintenant que nous avons placé une image dans notre dossier *data*, nous pouvons désormais nous en servir dans notre programme.

```
size(500,400);

PImage ile;
ile = loadImage("ile.jpg");

image(ile,50,10);
```

D'abord nous avons donné à notre espace de dessin une taille plus grande que notre image, juste pour mieux comprendre comment celle-ci peut être positionnée dans le sketch final.

Il y a trois étapes pour afficher une image dans Processing :

1. Créer une *variable* qui contiendra les données (en pixels) de notre image.
2. Importer les pixels d'un fichier dans notre variable.
3. Dessiner les pixels de cette variable dans notre espace de dessin.

Tout d'abord, nous devons créer une *variable* Processing, avant d'importer notre *fichier* dedans. Mais avant tout, **à quoi sert une variable ?** Et bien dans ce cas précis, il s'agit d'un nom interne à notre programme qui contient l'ensemble des pixels de notre fichier *ile.jpg*. A chaque fois que nous écrirons par la suite le mot « `ile` » dans notre programme, Processing comprendra qu'il s'agit de la suite de valeurs en pixels qui composent notre image. C'est justement le rôle de cette action `loadImage("nomdefichier")` d'aller chercher ces pixels dans le fichier et les importer dans notre variable nommé « `ile` ».

Vous avez peut-être également noté un mot étrange au tout début du code ci-dessus, le mot `PImage`. Celui-ci indique à Processing le *genre* de la variable et lui permettra de consacrer assez de mémoire dans votre ordinateur pour contenir l'ensemble des données de ce genre. Dans le cas de n'importe quelle variable, la syntaxe à utiliser est {type de la variable} {nom de la variable} = {les valeurs de la variable}.

Par exemple, si par un tour de force il était possible d'importer un petit chien tout mignon dans Processing, il suffirait d'écrire `PetitChiot milou = loadDog("milou.dog");` On écrit d'abord le **type** de la chose, le **nom** de la chose, et enfin on lui donne sa valeur.

Ici, cette valeur est donnée par la fonction `loadImage()` qui va aller chercher les pixels de l'image dans le fichier et les importera dans notre *variable* nommée `ile`.

Pour plus d'informations sur les variables et les différents **types** de variables, reportez-vous au chapitre dédié à ce sujet.

Enfin, une fois notre variable remplie, nous la dessinons à une position {*x,y*} dans notre sketch. Si nous voulons dessiner notre image à sa taille originelle, nous utilisons la version courte de l'instruction `image` qui nécessite uniquement trois paramètres : `{PImage}, {x}, {y}`.

```
image(ile,50,10)
```

IMPORTER UNE IMAGE WEB

Nous pouvons également importer des images directement depuis Internet, en indiquant l'adresse web de l'image à la place du nom de fichier.

```
size(400,400);

PImage webcam;
webcam = loadImage("http://www.gutenberg.org/files/3913/3913-h/images/ro
image(webcam,10,20,width,height);
```

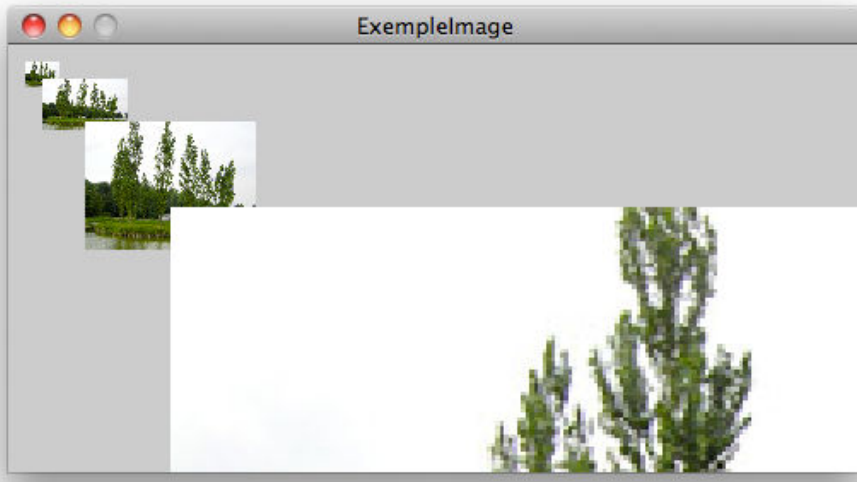
Si notre programme n'est pas animé, comme c'est le cas ici, il y aura juste une longue pause lorsque l'image se charge dans Processing.

Par contre, faites bien attention à placer ce type d'instructions au début du sketch sinon vous risquez de ralentir fortement le fonctionnement des autres parties du programme, celles-ci devant attendre le téléchargement complet de vos images depuis Internet avant de s'exécuter. Dans des cas très spécifiques, si vous avez besoin d'importer des images web en plein milieu de votre sketch, sachez qu'il existe des techniques appropriées dénommées « fils d'exécution ». Ces techniques séparent le chargement de fichiers médias depuis internet du reste des fonctions du programme. Il s'agit malheureusement d'un sujet trop avancé pour ce manuel. Pour davantage d'informations sur les fils d'exécution, reportez-vous au forum du site de Processing en faisant une recherche sur le mot « thread ».

CHANGER DE TAILLE

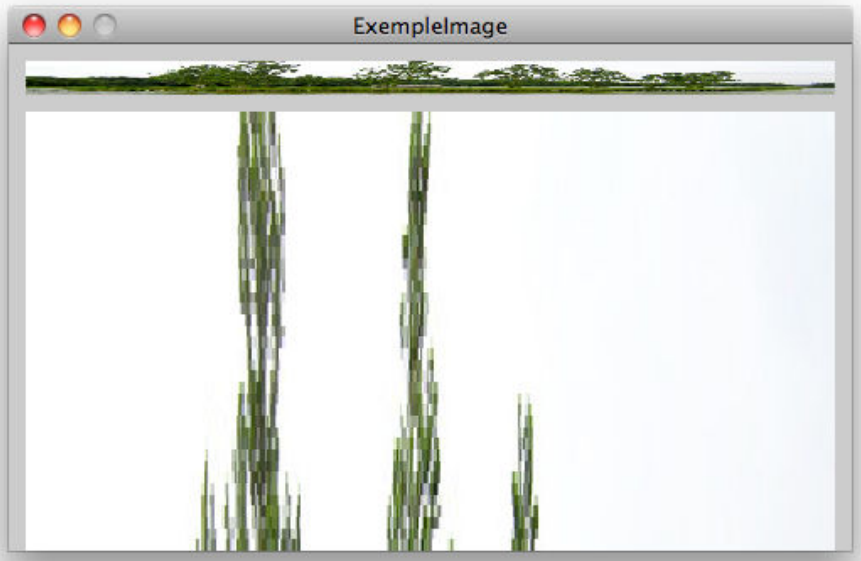
En ajoutant deux paramètres, nous pouvons changer la taille de l'image. Cette taille peut être plus petite ou plus grande que l'image d'origine, et à priori il n'y a pas vraiment de limite. Par contre, au-delà de la taille d'origine de l'image, Processing sera obligé d'inventer des pixels en doublant les originaux, ce qui donnera un effet pixelisé. Cet effet n'est pas forcément indésirable, à vous de voir.

Pour changer la taille de l'image, il suffit de rajouter deux paramètres à votre image, {*largeur*, *hauteur*}, ce qui nous amène à 5 paramètres : {*variableImage*, *x*, *y*, *largeur*,*hauteur*}.




```
size(500,250);

PImage ile;
ile = loadImage("ile.jpg");

image(ile,10,10,20,15);
image(ile,20,20,50,30);
image(ile,45,45,100,75);
image(ile,95,95,1000,750);
```

Notez que nous avons importé qu'une seule fois l'image dans notre variable `ile` et que celle-ci peut être dorénavant utilisée pour le restant de notre programme. Notez également que nous avons respecté les proportions {*x*,*y*} de notre image dans le changement de la taille, mais que celles-ci auraient pu prendre n'importe quelle valeur, ce qui reviendrait à étirer l'image sur un de ces deux axes.

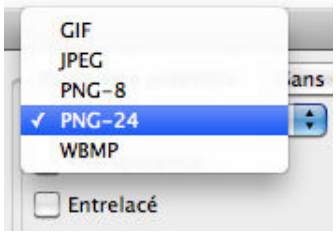


RENDRE TRANSPARENT L'ARRIÈRE-PLAN D'UNE IMAGE

Souvent nous avons besoin d'importer des images qui ne sont ni carrées, ni rectangulaires, comme dans le cas d'un petit jeu vidéo utilisant des silhouettes de personnages en deux dimensions : nous ne voulons pas voir en permanence un carré blanc autour du profil de notre héros. Malheureusement, à ce jour, les images à base de pixels doivent être importées dans un format carré.

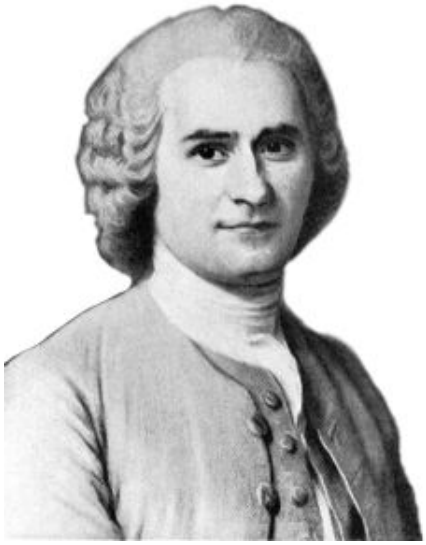
Pour lever cette contrainte, certains formats de fichiers font appel à une **couche alpha** pour gérer la transparence de certaines zones de l'image et ainsi laisser apparaître les éléments qui sont situés derrière elles (un fond coloré ou illustré par exemple). Cette couche se superpose aux couches *rouge, vert, bleu* utilisées pour composer l'image et indique l'intensité de la transparence pour chaque pixel, avec même la possibilité de pixels semi-transparents dans le cas du format *PNG*.

Si vous voulez sauvegarder votre image avec une couche alpha, vous avez trois possibilités qu'il faut choisir lors de l'exportation de votre image dans votre logiciel de traitement d'image préféré :



Chaque logiciel exportera les images à sa manière. Sachez juste qu'il existe une différence par exemple entre *GIF*, *PNG-8*, et *PNG-24*, notamment sur la façon dont chaque format va traiter cette couche alpha. Le plus sûr des trois, et qui vous donnera plus d'options, c'est le *PNG-24*.

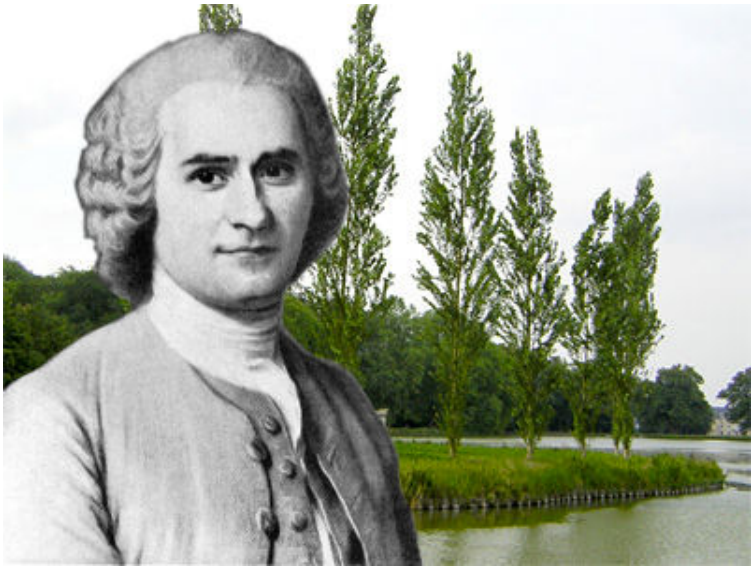
Voici une image du philosophe Jean-Jacques Rousseau avec un fond transparent.



Quel que soit le fond sur lequel elle sera placée, cette image s'y intègrera parfaitement parce qu'elle contient justement une couche alpha décrivant les parties transparentes et non transparentes du carré de pixels qui la composent. Notez que les pixels autour de Rousseau sont transparents, alors que son front et son cou sont opaques.

N'hésitez pas à copier cette image au format png sur votre ordinateur depuis la version en ligne de ce manuel : http://fr.flossmanuals.net/processing/, chapitre *Les images* (conçue à partir d'une illustration mise à disposition sur le site Commons de Wikimedia, cette image est libre de droit).

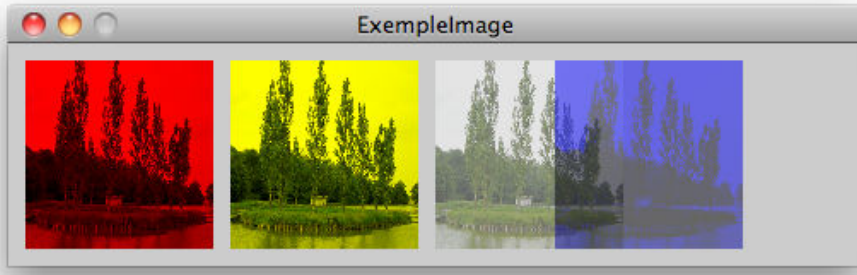
Dans notre programme Processing, nous allons pouvoir constater que les pixels situés autour du personnage sont effectivement transparents en superposant l'image de Rousseau (au format png) avec celle du paysage précédemment utilisée (au format jpg) :



```
size(400,300);

PImage ile;
ile = loadImage("ile.jpg");

PImage rousseau;
rousseau = loadImage("rousseau.png");

image(ile,0,0);
image(rousseau,20,20);
```

COLORIER LES IMAGES

On peut également colorier les images. Autrement dit, on peut changer la couleur des pixels, soit dans leur ensemble, soit individuellement. La plus simple de ces méthodes concerne le coloriage de l'ensemble des pixels. Cette méthode de coloriage ressemble à peu près au coloriage de rectangles, mais nécessite une nouvelle instruction, `tint()`, pour le distinguer du coloriage direct des pixels à l'intérieur du rectangle. Dans le cas de `tint()`, Processing va modifier la couleur de chaque pixel au moment où celui-ci est dessiné dans l'espace de dessin.




```
size(500,130);

PImage ile;
ile = loadImage("ile.jpg");

tint(255,0,0);
image(ile, 10,10, 110,110);

tint(255,255,0);
image(ile, 130,10, 110,110);

tint(255,255,255,127);
image(ile, 250,10, 110,110);

tint(0,0,255,127);
image(ile, 320,10, 110,110);
```

Tout comme les instructions `fill()` et `stroke()`, l'instruction `tint()` peut prendre une, deux, trois, ou quatre valeurs, selon ce que nous voulons faire. En indiquant trois paramètres, par exemple, nous pouvons augmenter/diminuer l'intensité de la couche **rouge**, **vert**, ou **bleu** de notre image. En indiquant quatre paramètres, nous pouvons augmenter/diminuer, en plus de ces trois couleurs, la valeur de *transparence/opacité* de l'image.

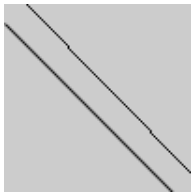
10. LES STYLES DE BORDURES

Le style des traits et des bordures des formes géométriques peut être ajusté afin d'éviter l'apparition d'effets graphiques indésirables sur les lignes obliques, aux intersections ou en bout de ligne. Différentes commandes permettant d'affiner le rendu sont présentées ci-dessous.

SMOOTH

La méthode `smooth()` permet d'activer le lissage des contours. Elle permet d'éviter l'effet d'escalier qui apparaît sur les lignes diagonales.

```
line(10, 0, 100, 90);  // Ligne sans lissage

//On active le lissage
smooth();
line(0, 10, 90, 100); // Ligne lissée
```



STROKEWEIGHT

La méthode `strokeWeight()` permet de varier l'épaisseur d'une ligne ou d'un contour.

```
line(10, 0, 100, 90); // Ligne de 1 pixel d'épaisseur

strokeWeight(5); //On définit l'épaisseur à 5 pixels
line(0, 10, 90, 100); // Ligne de 5 pixels d'épaisseur
```

STROKECAP

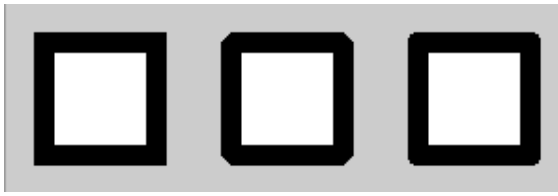
La méthode `strokeCap()` permet de définir l'apparence des extrémités d'une ligne. Cette méthode n'est pas utile pour les formes. Elle peut avoir les valeurs `SQUARE` (extrémité carré), `PROJECT` (extrémité avec 2 petits angles brisés) ou `ROUND` (extrémité arrondie). Par défaut c'est le mode `ROUND` qui est utilisé. Cette méthode ne fonctionne pas avec `P3D` ou `OpenGL`.

```
strokeWeight(10); // On définit l'épaisseur des traits à 10 pixels
strokeCap(ROUND); // extrémité arrondie
line(20, 40, 60, 80);

strokeCap(PROJECT); // extrémité avec 2 petits angles brisés
line(20, 20, 80, 80);

strokeCap(SQUARE); // extrémité carré
line(40, 20, 80, 60);
```

STROKEJOIN

La méthode `strokeJoin()` permet de modifier l'aspect des jointures. Elle peut avoir les valeurs `MITER`, `BEVEL` ou `ROUND`. Par défaut c'est le mode `MITER` qui est utilisé. Cette méthode ne fonctionne pas avec `P3D` ou `OpenGL`.

```
size(300, 100); // On modifie la taille du sketch

strokeWeight(10); // On définit l'épaisseur à 10 pixels
strokeJoin(MITER); // Jointure carré
rect(20, 20, 60, 60);

strokeJoin(BEVEL); // Jointure brisée
rect(120, 20, 60, 60);

strokeJoin(ROUND); // Jointure arrondie
rect(220, 20, 60, 60);
```



11. LA TYPOGRAPHIE

Ce chapitre va vous permettre de personnaliser l'usage des textes dans Processing en utilisant des polices de caractères alternatives.

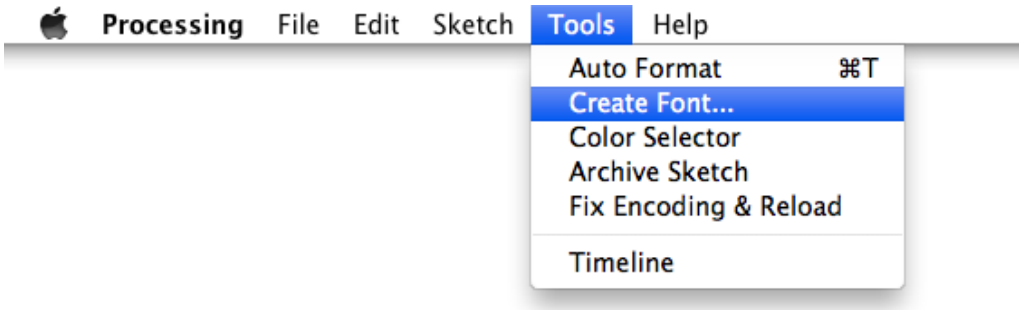
LA FORME DES MOTS

Si nous voulons dessiner avec une autre forme typographique que celle définie par défaut, il faut effectuer quelques étapes préalables :

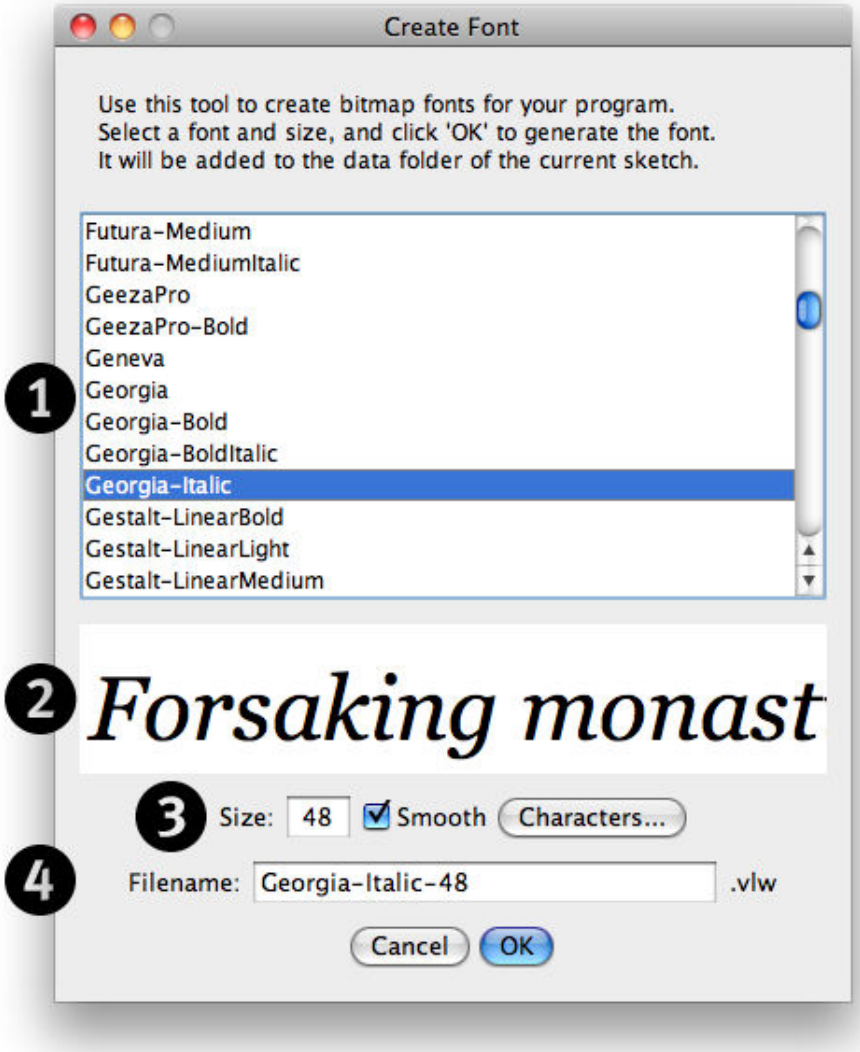
1. Convertir une police de caractères en un format de fichier compatible avec Processing ;
2. importer ce fichier dans le code du programme (ce fichier comporte toutes les informations graphiques décrivant l'apparence de la police utilisée) ;
3. sélectionner cette police et l'activer dans notre programme ;
4. dessiner du texte dans notre sketch à l'aide des instructions appropriées pour que le résultat s'affiche dans la fenêtre de visualisation de Processing.

IMPORTER UNE POLICE DE CARACTÈRES

Pour dessiner du texte dans notre fenêtre de visualisation, il faut choisir tout d'abord son apparence, en indiquant sa police de caractères. Pour bien réussir cette étape, nous vous recommandons de sauvegarder d'abord votre sketch dans votre dossier de travail (voir chapitre Bases de Processing). Une fois notre sketch sauvegardé, nous allons sélectionner, dans le menu **Tools**, l'action **Create Font...**



A priori nous devrions maintenant voir une fenêtre **Create Font** qui permet de convertir quasiment n'importe quelle police de caractère en une forme utilisable dans notre sketch. Cette police doit être installée au préalable dans notre ordinateur pour qu'elle apparaisse dans cette liste.



Cette fenêtre est décomposée en quatre parties :

1. La liste des polices actuellement installées sur notre ordinateur,
2. Une prévisualisation de la police actuellement sélectionnée, affichée à la taille indiquée dans le prochain champ (la zone numérotée 3 dans la copie d'écran ci-dessus),
3. A peu près la taille maximale à laquelle nous voulons dessiner cette police,

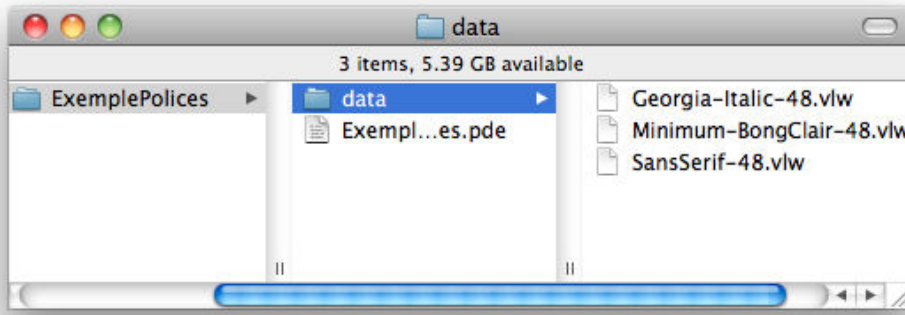
4. Le nom du fichier de cette police, une fois convertie dans le format natif de Processing (.vlw).

Vous avez peut-être noté qu'il existe également une case à cocher **Smooth** qui active/désactive l'anti-aliasing (fonction de lissage des polices pour éviter un effet de crénelage), ainsi qu'un bouton **Characters...** qui permet de préciser les caractères spéciaux qui doivent être inclus lors de la conversation de la police. Pour ne pas compliquer les choses, nous allons laisser ces deux options avec leurs valeurs par défaut.

Dans l'illustration ci-dessus nous avons sélectionné la police *Georgia*. C'est à partir du nom de cette police et de sa taille que Processing générera le fichier de la police à importer, ex : *« Georgia-Italic-48.vlw »*. Notons enfin que l'extension « .vlw » associée à l'intitulé du fichier sera rajoutée à toutes les polices que nous importerons de cette manière. Si par curiosité vous vous intéressez à l'origine de cette extension, son nom fait référence sous forme d'un acronyme au « Visual Language Workshop » (vlw) du MIT Media Lab. C'est ce laboratoire qui historiquement est à l'origine d'un certain nombre de principes et de travaux qui ont permis à Processing de voir le jour.

Si nous voulons savoir où Processing a sauvegardé notre police, il suffit de sélectionner, dans le menu **Sketch**, l'action **Show Sketch Folder**.

Cette action fera apparaître le dossier « data » dans lequel notre police a été sauvegardée. Son fichier s'appelle « Georgia-Italic-48.vlw ». C'est ce nom que nous devons retenir pour intégrer la police dans notre programme.



DESSINER UNE PHRASE

Nous allons enfin dessiner avec notre police. Pour cela, il faut faire trois choses :

1. Importer le fichier *Georgia-Italic-48.vlw* dans une *variable* afin que notre programme connaisse le nom de la police utilisée et sache la dessiner lorsqu'il affichera du texte. Ce fichier contient en effet les informations décrivant la structure géométrique de la police pour pouvoir la reproduire ;

2. Sélectionner cette variable dans notre programme comme police active ;
3. Dessiner un caractère ou une phrase quelque part dans notre sketch à l'aide des instructions appropriées pour le voir ensuite s'afficher dans la fenêtre de visualisation de Processing.

En option, il est possible de choisir la taille à laquelle nous voulons dessiner avec notre police, mais comme nous avons déjà paramétré cet aspect lors de la création du fichier, il ne sera pas nécessaire de l'indiquer ici.

Voici le code complet d'un programme simple qui réunit toutes ces étapes pour dessiner une phrase dans la fenêtre de visualisation. Par tradition, nous allons faire nos premiers pas dans l'écriture en écrivant « Salut tout le monde ! ».

```
size(500,150);

PFont police;
police = loadFont("Georgia-Italic-48.vlw");
textFont(police,48);

text("Salut tout le monde !", 20, 75);
```

Tout d'abord, nous avons fixé la taille de notre fenêtre de visualisation (l'espace de dessin), comme dans quasiment n'importe quel programme Processing.

Ensuite, nous avons importé notre *fichier* dans une variable Processing (dénommée police). A quoi sert une variable ? Et bien dans ce cas précis, il s'agit d'un nom interne à notre programme qui fait référence au fichier de la police *Georgia-Italic-48.vlw* que nous souhaitons utiliser. A chaque fois que nous écrirons par la suite le mot `police` dans notre programme, Processing comprendra qu'il s'agit de faire appel à la police Georgia Italic 48 contenue désormais dans ce mot.

Vous avez peut-être également noté un mot étrange en tout début de cette phrase, le mot `PFont`. Celui-ci indique à Processing le genre de la variable et lui permettra d'ouvrir assez de mémoire pour contenir l'ensemble des données de ce genre. Dans le cas de n'importe quelle variable, la syntaxe à utiliser est {type de la variable} {nom de la variable} = {les valeurs de la variable}.

Par exemple, si par un tour de force il était possible d'importer un petit chaton tout mignon dans Processing il suffirait d'écrire `petitChaton mioumiou = loadPetitChaton("mioumiou.chat");` On écrit d'abord le type de la chose, le nom de la chose, et enfin on lui donne sa valeur. Ici, cette valeur est donnée par la fonction `loadFont()` qui va aller chercher la structure géométrique de la police dans le fichier et l'importera dans notre variable nommée « `police` ».

La suite de notre programme est probablement un peu plus intuitive. Nous sélectionnons la police avec laquelle nous voulons dessiner. Même s'il n'existe qu'une seule police actuellement dans notre programme, il faut néanmoins passer par cette

étape. Notez que vous pouvez importer autant de polices que vous voulez et passer de l'un à l'autre, à l'image de ce qu'il est possible de faire avec les couleurs.

Dans cet exemple, nous avons indiqué la taille de la police juste pour vous montrer qu'il est possible de la changer en cours de route.

Enfin, nous dessinons une petite phrase et indiquons la position *{x,y}* où notre texte doit se dessiner. On obtient le résultat suivant :

POINT D'ORIGINE

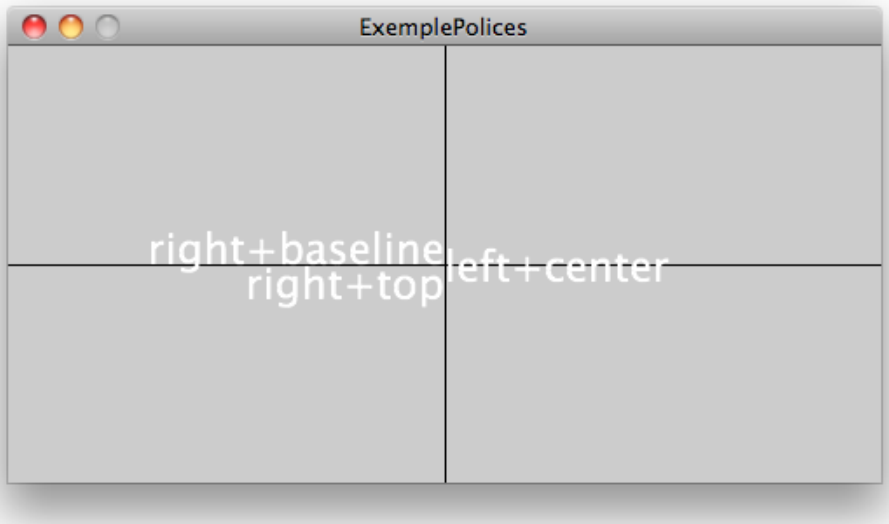
Pour rendre plus clair le rapport entre la position *{x,y}* de notre message et sa forme typographique, nous avons également dessiné dans l'illustration ci-dessus une petite croix pour rendre plus explicite la façon dont Processing positionne l'écriture :

```
size(500,150);

PFont police;
police = loadFont("Georgia-Italic-48.vlw");
textFont(police);

text("Salut tout le monde !", 20, 75);

// indiquer la position d'origine du texte
stroke(255,0,0);
line(15,70,25,80);
line(15,80,25,70);
```

Tout comme les rectangles, qui peuvent se dessiner depuis leur point supérieur gauche, depuis leur centre, ou depuis ses quatre extrémités, l'écriture du texte peut également être positionnée à partir de plusieurs points d'origine. Par défaut, le texte s'écrit depuis la ligne de base du texte, c'est-à-dire le point en bas à gauche du texte, mais au-dessus des caractères descendants comme les lettres « y » ou « j ».

Vous pouvez changer la façon dont Processing alignera ce texte, en se servant de la fonction `textAlign()` :

```
size(500,250);

PFont police;
police = loadFont("SansSerif-24.vlw");
textFont(police,24);

line(250,0,250,500);
line(0,125,500,125);

textAlign(RIGHT,TOP);
text("right+top", 250, 125);

textAlign(RIGHT,BASELINE);
text("right+baseline", 250, 125);

textAlign(LEFT,CENTER);
text("left+center", 250, 125);
```

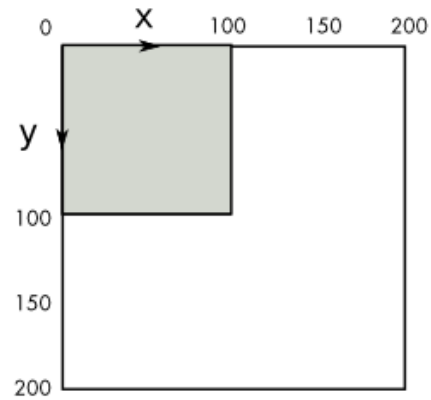


12. LES TRANSFORMATIONS

Jusqu'à présent, nous avons dessiné des formes dans la fenêtre de notre application, en nous repérant toujours par rapport au coin supérieur gauche de la fenêtre.

Grâce aux transformations, il va être possible de déplacer cette origine, mais aussi de redéfinir l'orientation des axes et même de changer la graduation de ces axes (on parle de changement d'échelle).

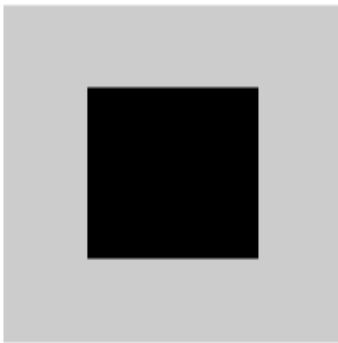
Par défaut, lorsque l'on dessine une forme (dans notre exemple un rectangle), Processing définit le repère suivant :

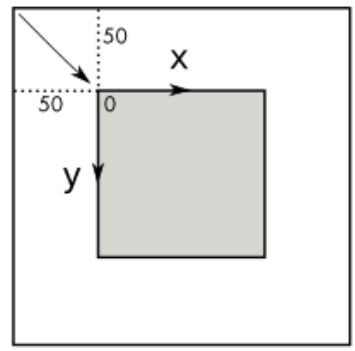



```
size(200, 200);
noStroke();
fill(0);
rect(0, 0, 100, 100);
```

DÉPLACER

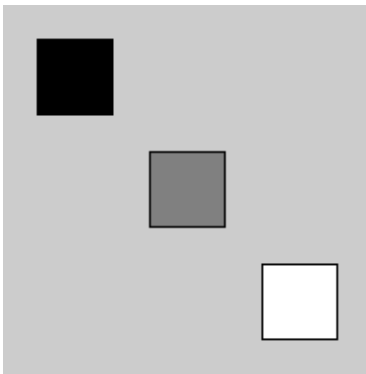
Le changement de la position de l'origine se fait par la commande `translate()`. Nous pouvons nous déplacer sur l'axe x (« horizontalement ») et sur l'axe y (« verticalement ») et nous allons indiquer à `translate()` de « combien » nous voulons nous déplacer sur chacun des axes. Dans l'exemple suivant, nous déplaçons l'origine de notre repère de 50 pixels en x et de 50 pixels en y. Notons que `translate()` va seulement affecter les formes géométriques qui sont dessinées après cette instruction.






```
size(200, 200);
noStroke();
fill(0);
translate(50, 50);
rect(0, 0, 100, 100);
```

Enchaîner les `translate()` permet d'accumuler les déplacements comme le montre l'exemple suivant.

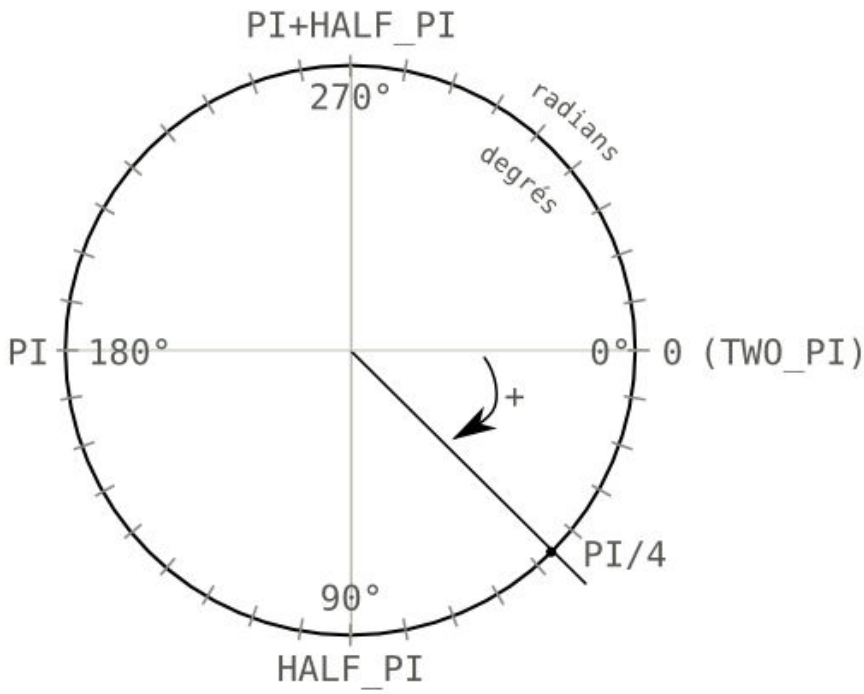


```
size(200,200);
```

```
// Noir
fill(0);
translate(20,20);
rect(0,0,40,40);

// Gris
fill(128);
translate(60,60);
rect(0,0,40,40);

// Blanc
fill(255);
translate(60,60);
rect(0,0,40,40);
```

TOURNER

Nous avons pu déplacer l'origine du repère de dessin. Nous allons maintenant appliquer une rotation sur les axes de notre repère. Grâce à la commande `rotate()`, les axes x et y peuvent changer d'orientation. `rotate()` prend en paramètre un nombre qui va représenter l'angle de rotation, c'est-à-dire de « combien » nos axes vont tourner par rapport à notre fenêtre. Des valeurs positives indiquent une rotation dans le sens des aiguilles d'une montre.



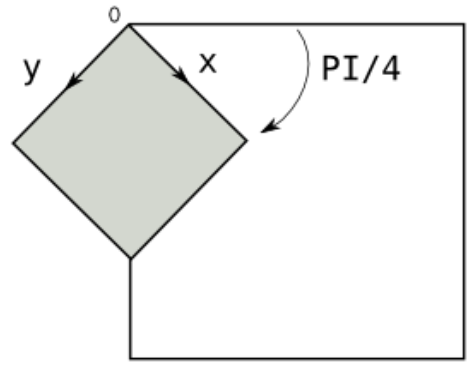
Unités de mesure

Deux systèmes de mesure existent pour mesurer un angle : les *radians* et les *degrés*. Par défaut, Processing travaille en radians mais pour nous il est d'habitude plus facile de raisonner en degrés. Par exemple tourner de 180°, c'est faire un demi-tour.

Processing permet de passer de transformer une unité en une autre grâce aux fonctions `radians()` et `degrees()`.

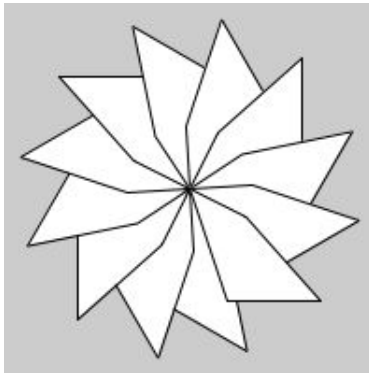
```
float d = degrees(PI/4); // transforme des radians en degrés
float r = radians(180.0); // transforme des degrés en radians
```

Illustrons la fonction `rotate()` par un exemple simple. Nous allons faire pivoter un carré autour de l'origine.




```
size(200, 200);
noStroke();
fill(0);
rotate(PI/4);
rect(0, 0, 100, 100);
```

Comme pour `translate()`, `rotate()` se place avant les formes géométriques à dessiner. Il est possible de combiner ces changements d'orientations, qui vont s'accumuler.

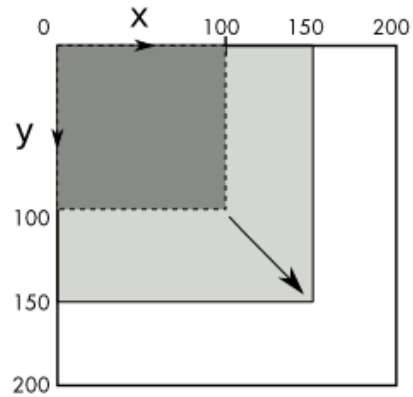


```
size(200,200);
smooth();
translate(width/2, height/2);
for (int i=0;i<360;i+=30){
  rotate(radians(30));
  quad(0, 0, 30, 15, 70, 60, 20, 60);

}
```

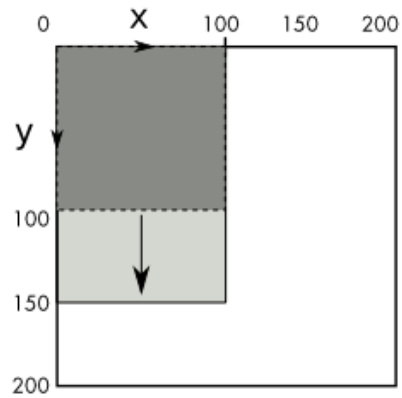
METTRE À L'ÉCHELLE

La mise à l'échelle permet de redimensionner les objets par la commande `scale()`. Cette fonction permet d'agrandir ou de diminuer la taille des formes géométriques. Elle accepte un ou deux paramètres. Par exemple, `scale(0.5)` va diminuer de moitié la taille des formes géométriques tandis que `scale(2.0)` va la doubler, `scale(1)` n'a aucun effet.

L'écriture avec deux paramètres permet de découpler le redimensionnement en x et en y. Par exemple, `scale(0.5, 2.0)` va écraser la forme sur les x de moitié tandis que sur les y sa taille sera doublée.




```
size(200,200);
scale(1.5);
rect(0,0,100,100);
```




```
size(200,200);
scale(1.0,1.5);
rect(0,0,100,100);
```

Comme pour `rotate()` et `translate()`, l'enchainement de `scale()` permet d'accumuler les mises à l'échelle. Illustrons cette propriété par le sketch suivant, qui reprend l'idée des poupées russes en emboitant des carrés par le jeu des `scale()` successifs.

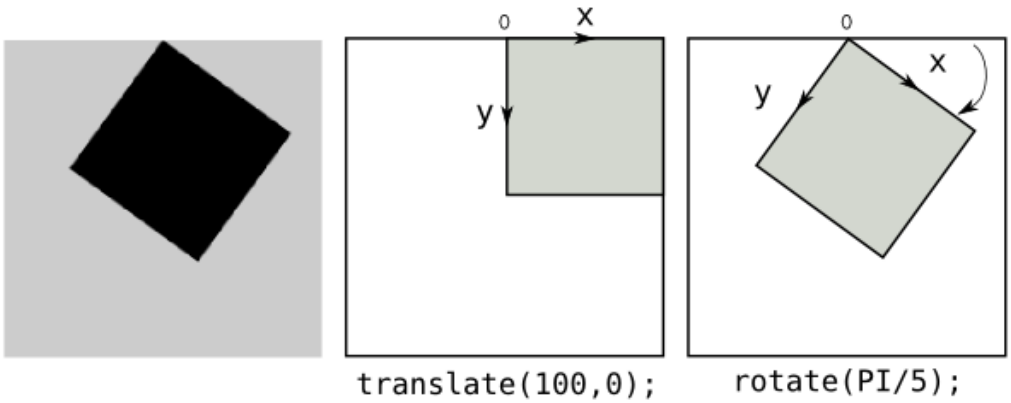
```
size(200,200);
noStroke();

// Noir
fill(0);
scale(1);
rect(0,0,200,200);

// Gris
fill(128);
scale(0.5);
rect(0,0,200,200);

// Blanc
fill(255);
scale(0.5);
rect(0,0,200,200);
```

L'ORDRE DES TRANSFORMATIONS

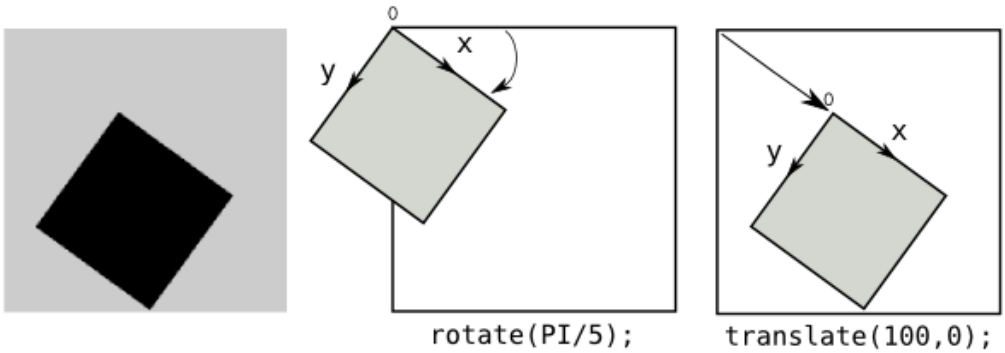
Il est possible de combiner plusieurs types de transformations. Comme nous l'avons vu dans les exemples précédents, les transformations s'accumulent au fur et à mesure des appels successifs à `translate()`, `rotate()` ou `scale()` et chacune des transformations tient compte des transformations précédentes.

Lorsqu'on utilise plusieurs types de transformations, leur ordre d'écriture va être important. Lorsque vous êtes en voiture, « tourner à gauche » puis « continuer tout droit » est différent de « continuer tout droit » puis « tourner à gauche ». Vous n'arrivez pas forcément au même endroit en suivant successivement ces deux instructions. C'est la même chose pour les transformations dans Processing.

Illustrons ceci par un exemple en inversant un `translate()` et un `rotate()`.




```
size(200,200);
smooth();
fill(0);

translate(100,0);
rotate(PI/5);
rect(0,0,100,100);
```




```
size(200,200);
smooth();
fill(0);

rotate(PI/5);
translate(100,0);
rect(0,0,100,100);
```

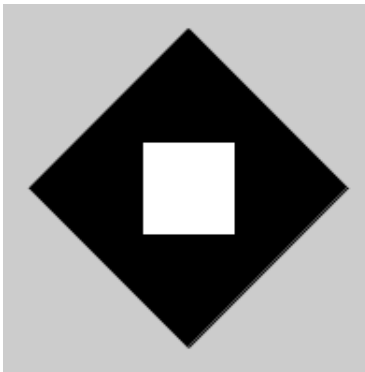
ISOLER LES TRANSFORMATIONS

Nous venons de voir que les transformations s'accumulaient au fur et à mesure de l'utilisation des commandes `translate()`, `rotate()` et `scale()`. Nous allons voir à présent comment sauvegarder les transformations à un moment donné et comment les restaurer ensuite, au cours d'une animation interactive faisant appel à la méthode `draw()`.

Nous allons utiliser pour cela deux fonctions qui s'utilisent toujours par paire : `pushMatrix()` et `popMatrix()`. Nous verrons en fin de chapitre pourquoi ces deux fonctions portent un nom si bizarre.

Pour les deux exemples qui suivent, nous allons identifier les transformations suivantes :

- **A** : origine en haut à gauche de la fenêtre.
- **B** : origine au centre de l'écran.
- **C** : origine au centre de l'écran, rotation de `PI/4`.

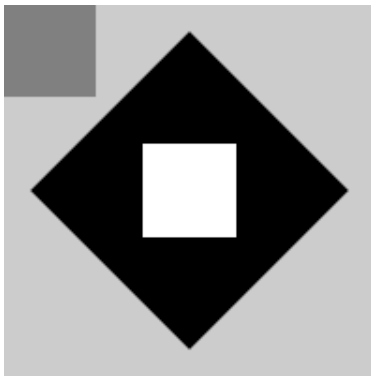


```
size(200,200);
smooth();
rectMode(CENTER);

// Repère au centre de l'écran
translate(width/2,height/2);

// Sauvegarde de A
pushMatrix();

// Transformation B
rotate(PI/4);

// Dessin du carré noir
fill(0);
```

```
rect(0,0,120,120);

// Restauration A
// A ce point-là, notre repère revient au centre de l'écran
popMatrix();

// Dessin du carré blanc qui ne tient pas compte de la rotation
fill(255);
rect(0,0,50,50);
```

Il est possible d'imbriquer les sauvegardes de transformations, c'est-à-dire qu'à l'intérieur de n'importe quelle paire de `pushMatrix()/popMatrix()` nous pouvons rappeler ces fonctions pour sauver l'état de la transformation courante.

Reprenons l'exemple précédent en plaçant une paire `pushMatrix()/popMatrix()` qui encadre la première transformation `translate(width/2, height/2)`.



```
size(200,200);
smooth();
rectMode(CENTER);
noStroke();

// Sauvegarde de A
pushMatrix();

// Transformation B
translate(width/2,height/2);

// Sauvegarde de B
pushMatrix();

// Transformation C
```

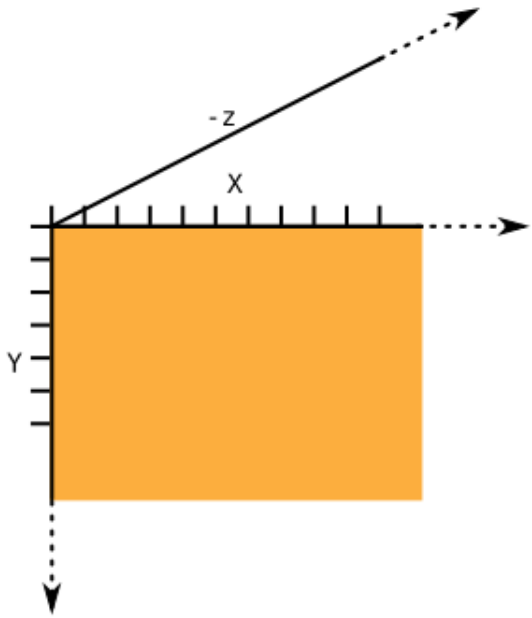
```
rotate(PI/4);

// Dessin du carré noir
fill(0);
rect(0,0,120,120);

// Restauration de B
popMatrix();

// Dessin du carré blanc qui ne tient pas compte
// de la rotation rotate(PI/4)
fill(255);
rect(0,0,50,50);

// Restauration de A
popMatrix();

// Dessin du carré gris
fill(128);
rect(0,0,100,100);
```

TRANSFORMER EN 3D

Toutes les transformations que nous venons d'aborder sont applicables en trois dimensions (3D). Processing permet de passer en 3D au moment de l'appel à `size()` :

```
size(300,300,P3D);
```

Dans ce mode, Processing définit un axe z qui pointe vers le fond de l'écran.

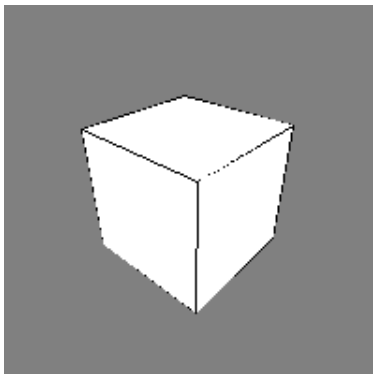


Les transformations de déplacement et de mise à l'échelle vont s'écrire en intégrant un troisième paramètre. Par exemple, pour se déplacer au centre de l'écran le long de x et y puis dessiner les formes comme si elles étaient éloignées de nous, nous pourrions écrire la ligne de code suivante :

```
translate(width/2, height/2, -100);
```

Pour les rotations, nous disposons de trois fonctions : `rotateX`, `rotateY` et `rotateZ` qui permettent respectivement de tourner autour des axes x, y et z.

Processing intègre des fonctions de dessin de formes simples en 3D, notamment les cubes et les sphères. Nous allons créer un cube dont la rotation autour des axes x et y va être paramétré par la position de la souris.



```
float rx = 0;
float ry = 0;
float z = 100;

void setup() {
  size(200,200,P3D);
}

void draw() {
  background(128);
  rx = map(mouseX, 0,width,-PI,PI);
  ry = map(mouseY, 0,height,-PI,PI);

  translate(width/2,height/2,z);
  rotateX(rx);
  rotateY(ry);
  box(30);
}
```

Dans ce sketch, nous avons introduit une nouvelle fonction `map()`, qui permet de transformer une valeur d'une plage de valeurs à une autre plage de valeurs. Voici un exemple simple pour expliquer ce concept :

```
float v = 100;
float m = map(v,0,200, 0,1); // m vaut 0.5
```

Dans cet exemple, `map()` va transformer la valeur 100 qui est dans l'intervalle [0;200] et calculer la valeur équivalente dans l'intervalle [0;1]. La valeur 0.5 est retournée dans `m`.

Dans notre sketch, cette fonction permet de transformer la valeur de `mouseX` dans l'intervalle situé entre `0` et `width` en une valeur équivalente dans l'intervalle entre `-PI` et `PI`.

Les concepts de `pushMatrix()` et `popMatrix()` sont aussi applicables en 3D pour sauvegarder et restaurer les transformations. C'est le meilleur moyen pour dessiner des univers en 3D, contenant plusieurs objets en mouvement les uns par rapport aux autres sans avoir recours à des concepts mathématiques complexes.

Toutes les transformations dans Processing sont stockées dans un tableau de 16 nombres qui est appelé matrice ou *matrix* en anglais. Ces nombres sont directement modifiés par des appels aux fonctions de transformations. Si vous êtes curieux, vous pouvez imprimer ce tableau par la fonction `printMatrix()`.

PROGRAMMER

13. Les variables
14. Les conditions
15. Les répétitions
16. Les listes
17. Les méthodes
18. Les objets
19. Les commentaires

13. LES VARIABLES

Une variable est une donnée que l'ordinateur va stocker dans l'espace de sa mémoire. C'est comme un compartiment dont la taille n'est adéquate que pour un seul type d'information. Elle est caractérisée par un nom qui nous permettra d'y accéder facilement.

42	491	33.145	5	« Salut tout le monde ! »	true
x	y	angle	i	message	vie

Il existe différents type de variables : des nombres entiers (`int`), des nombres à virgule (`float`), du texte (`String`), des valeurs **vrai/faux** (`boolean`). Un nombre à décimales, comme 3,14159, n'étant pas un nombre entier, serait donc du type `float`. Notez que l'on utilise un point et non une virgule pour les nombres à décimales. On écrit donc `3.13159`. Dans ce cas, les variables peuvent être annoncées de cette manière :

```
float x = 3.14159;
int y = 3;
```

Le nom d'une variable peut contenir des lettres, des chiffres et certains caractères comme la barre de soulignement. À chaque fois que le programme rencontre le nom de cette variable, il peut lire ou écrire dans ce compartiment. Les variables qui vont suivre vous donneront des exemples simples de leur utilisation. Pour résumer, une variable aura un type, un nom et une valeur qui peut être lue ou modifiée.

INT

Dans la syntaxe de *Processing*, on peut stocker un nombre entier, par exemple 3, dans une variable de type `int`.

```
int entier;
entier = 3;
print(entier);
```

```
3
```

FLOAT

Il s'agit d'un nombre avec décimales, par exemple 2,3456.

```
float decimal;
decimal = PI;
print(decimal);
```

```
3.1415927
```

DOUBLE

Il s'agit également de nombre avec décimales, mais qui fournissent davantage de précision que le type float.

```
double long_decimal;
long_decimal = PI;
print(long_decimal);
```

```
3.1415927410125732
```

BOOLEAN

Il s'agit d'un type de variable qui ne connaît que deux états : Vrai (`true`) ou Faux (`false`). Elle est utilisée dans les conditions pour déterminer si une expression est vraie ou fausse.

```
boolean vraifaux;
vraifaux = true;
println(vraifaux);
```

```
true
```

CHAR

Cette variable sert à stocker un caractère typographique (une lettre). Notez l'usage de ce qu'on appelle des guillemets simples.

```
char lettre;
lettre = 'A';
print(lettre);
```

```
A
```

STRING

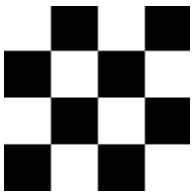
Cette variable sert à stocker du texte. Notez l'usage des guillemets doubles.

```
String texte;
texte = "Bonjour!";
print(texte);
```

```
Bonjour!
```

COLOR

Sers à stocker une couleur. Cette variable est utile lorsqu'on veut réutiliser souvent les mêmes couleurs.



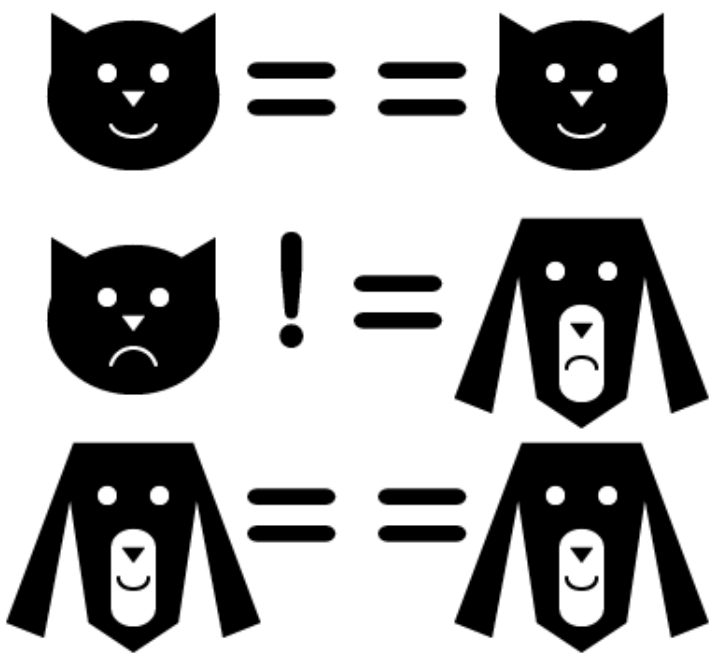
```
noStroke();
color blanc = color(255, 255, 255);
color noir = color(0, 0, 0);

fill(blanc); rect(0, 0, 25, 25);
fill(noir);  rect(25, 0, 25, 25);
fill(blanc); rect(50, 0, 25, 25);
fill(noir); rect(75, 0, 25, 25);

fill(noir); rect(0, 25, 25, 25);
fill(blanc); rect(25, 25, 25, 25);
fill(noir); rect(50, 25, 25, 25);
fill(blanc); rect(75, 25, 25, 25);

fill(blanc); rect(0, 50, 25, 25);
fill(noir); rect(25, 50, 50, 25);
fill(blanc); rect(50, 50, 75, 25);
fill(noir); rect(75, 50, 100, 25);

fill(noir); rect(0, 75, 25, 25);
fill(blanc); rect(25, 75, 25, 25);
fill(noir); rect(50, 75, 25, 25);
fill(blanc); rect(75, 75, 25, 25);
```

14. LES CONDITIONS

Les conditions donnent une part d'autonomie à votre ordinateur. Elles lui permettent de modifier le comportement du programme en fonction de diverses conditions de votre choix. Par exemple, si vous vouliez changer l'apparence de votre programme en fonction de l'heure, vous pourriez lui demander d'avoir un fond noir entre 10 heures du soir et 6 heures du matin et un fond blanc le reste du temps. C'est ce questionnement — « Quelle heure est-il ? » — qui constitue la *condition*. « S'il fait nuit, je dois dessiner un fond noir, sinon je dessine un fond blanc » pourrait constituer en quelque sorte le dialogue interne de Processing lorsqu'il rencontre une condition.

COMPARAISON

La base de la condition, c'est la comparaison. Avant de pouvoir agir selon une condition, il faut d'abord formuler la question que Processing doit se poser. Cette question sera quasiment toujours une question de comparaison.



Si le résultat de la question est « oui », Processing exécutera une suite d'instructions. Si la réponse est non, il exécutera une autre. Dans Processing, ce oui/non s'écrit « true » et « false ».

La syntaxe d'une condition est la suivante: `if (TEST) { }`. Le `TEST` correspond à l'opération (égalité, plus petit, plus grand) que vous aurez choisie pour comparer deux valeurs et déterminer si la réponse à la question est `true` ou `false`. Si la réponse est `true`, Processing exécutera les instructions entre les deux accolades. L'instruction `else` permet de gérer le cas de figure dans lequel la condition n'est pas validée. Elle exécute elle aussi tout ce qui se trouve à l'intérieur de ses accolades. Vous pouvez mettre autant d'instructions que vous voulez entre ces deux types d'accolades.

Egalité

Pour vérifier l'égalité de deux valeurs, on utilise la formule suivante: `if (valeur1 == valeur2) { }`. L'exemple suivant écrit "Il est midi" dans la console si la méthode `hour()` donne la valeur 12.

```
if (hour() == 12) {
  println("Il est midi !");
} else {
  println("Ce n'est pas midi !");
}
```

Résultat de l'application exécutée entre 12h00 et 12h59 est :

```
Il est midi !
```

Plus petit que et plus grand que

On peut vérifier qu'une valeur est plus petite ou plus grande qu'une autre en utilisant les opérateurs < et >. L'exemple suivant va écrire dans la console si nous sommes le matin ou non.

```
if (hour() < 12) {
  println("C'est le matin !");
} else {
  println("Ce n'est pas le matin !");
}
```

Résultat de l'application exécutée après 12h59 :

```
Ce n'est pas le matin !
```

COMBINER LES DÉCISIONS

Les `if` et `else` peuvent être combinés pour gérer plusieurs cas de figure.

```
if (hour() < 12) {
  println("C'est le matin !");
} else if (hour() == 12) {
  println("Il est midi !");
} else {
  println("Ce n'est pas le matin !");
}
```

Résultat de l'application exécutée avant 12h00 :

```
Ce n'est le matin !
```

COMBINER LES TESTS

Plusieurs tests peuvent être combinés au sein d'une même décision pour rendre le choix plus précis. Les opérateurs `&&` (et) ainsi que `||` (ou) permettent de combiner des tests. Par exemple pour déterminer si nous sommes la nuit ou le jour, nous avons besoin de trier les heures qui sont tard le soir et tôt le matin de celles du reste de la journée :

```
if (hour() < 6 && hour() > 20) {
  println("Il fait nuit !");
} else {
  println("Il ne fait pas nuit !");
}
```

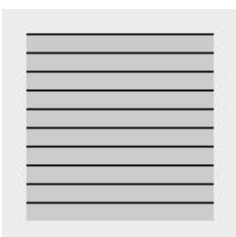
Résultat de l'application exécutée à 16h50 :

```
Il ne fait pas nuit !
```

15. LES RÉPÉTITIONS

Les répétitions permettent d'exécuter une série d'instructions plusieurs fois de suite. Elles évitent de dupliquer inutilement des portions de code. Attention ! les répétitions ne permettent pas de créer des animations dans le temps (d'autres instructions existent pour cela) ! Lorsque l'ordinateur lit le programme et rencontre une boucle, il va exécuter instantanément autant de fois de suite le code écrit dans le bloc de la boucle qu'on lui a indiqué.

L'exemple ci-dessous va nous permettre d'illustrer simplement cette notion. L'objectif est de réaliser un dessin affichant dix lignes noires horizontales. Le premier code contient dix fois l'instruction `line()`, le second code est réalisé à partir d'une boucle. Le résultat des deux codes est le même, la différence se situant au niveau de la longueur du code, l'un étant plus rapide à saisir (et ultérieurement à modifier) que l'autre.



```
line(0, 0, 100, 0);
line(0, 10, 100, 10);
line(0, 20, 100, 20);
line(0, 30, 100, 30);
line(0, 40, 100, 40);
line(0, 50, 100, 50);
line(0, 60, 100, 60);
line(0, 70, 100, 70);
line(0, 80, 100, 80);
line(0, 90, 100, 90);
```

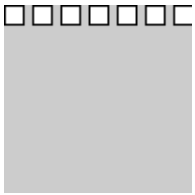
...ou plus simplement:

```
for (int i = 0; i < 100; i = i + 10) {
  line(0, i, 100, i);
}
```

LA BOUCLE FOR

Ce type de boucle permet de répéter une série d'instructions un nombre de fois défini. Elle incorpore une variable qui va s'incrémenter à chaque passage dans la boucle. On utilise souvent `i` comme nom pour la variable interne de la boucle. Elle comprend : un nombre de départ, un nombre maximal et une incrémentation. Sa syntaxe est la suivante : `for (int i = NombreDeDépart; i < NombreMaximal; i = i + INCREMENT) { }`.

L'exemple ci-dessous va afficher des rectangles blancs côte à côte qui auront chacun 10 pixels de large sur toute la largeur de l'espace de dessin. Ils seront espacés de 5 pixels. Nous allons afficher le premier rectangle aux coordonnées 0,0. Les suivants seront affichés aux coordonnées 15,0 puis 30,0 et ainsi de suite. Notre boucle va incrémenter sa variable de 15 pixels à chaque étape. Comme nous voulons remplir toute la largeur de l'espace de dessin, la valeur maximale sera égale à la largeur (`width`) de cette fenêtre de visualisation.



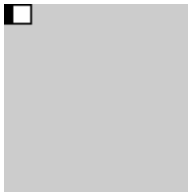
```
for (int i = 0; i < width; i = i + 15) {
  rect(i, 0, 10, 10);
}
```

LES COMPTEURS

Jusqu'à maintenant, nous avons utilisé les boucles `for` de manière à ce que la variable interne de la boucle soit directement exploitable. Dans l'exemple précédent, elle nous donne immédiatement la valeur exacte de la position sur l'axe x du rectangle à dessiner.

Les boucles `for` peuvent aussi être utilisées comme des compteurs. On leur donne un minimum, un maximum et on incrémente la variable seulement de 1 à chaque étape de la boucle : ceci permet de compter le nombre de fois que les instructions seront exécutées, d'où l'expression de compteur.

Cette méthode implique davantage de calculs mathématiques pour exploiter la variable au sein de la boucle.



```
for (int i = 0; i < 5; i = i + 1) {
  rect(i, 0, 10, 10);
}
```

Nous voyons dans l'exemple ci-dessus que les rectangles sont empilés les uns sur les autres. Puisque notre variable `i` aura des valeurs de 0 à 5 et que nous l'utilisons telle quelle pour placer les rectangles dans l'espace, ceux-ci seront placés aux coordonnées 0,0 ; 1,0; 2,0; 3,0; 4,0. Seul le dernier sera entièrement visible. Pour obtenir le même résultat que l'exemple précédent, il faut multiplier la variable. Dans le cas présent nous allons la multiplier par 15 (10 pour la largeur du rectangle et 5 de marge). Ils sont ainsi disposés tous les 15 pixels. Nous pourrions également utiliser le chiffre du compteur pour réaliser d'autres opérations.

```
for (int i = 0; i < 5; i = i + 1) {
  rect(i * 15, 0, 10, 10);
}
```

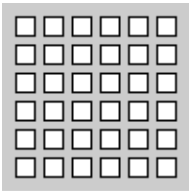
IMBRIQUER DES BOUCLES

Les boucles peuvent s'imbriquer les une dans les autres. Cette technique permet de rapidement passer à des visualisations à deux, voir trois dimensions. Lorsqu'on imbrique des boucles, il faut prendre garde au nom que l'on donne à la variable de chaque boucle. En effet si elles se nomment toutes `i`, le programme va mélanger les boucles. Chaque variable de chaque boucle doit avoir un nom propre. Par exemple : `i`, `j`, `k`, etc. ou si elles sont liées à des dimensions : x, y et z.

Boucle de départ

```
translate(7, 7);

for (int x = 0; x < 6; x = x + 1) {
  rect(x * 15, 0, 10, 10);
}
```

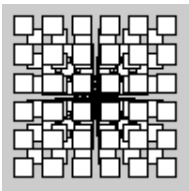
Deux boucles



```
translate(7, 7);

//Première boucle (hauteur)
for (int y = 0; y < 6; y = y + 1) {

  //Seconde boucle (largeur)
  for (int x = 0; x < 6; x = x + 1) {
    rect(x * 15, y * 15, 10, 10);
  }
}
```

Trois boucles

Dans cet exemple nous introduisons un espace 3D. Pour placer nos rectangles dans la profondeur, nous devons utiliser la méthode `translate()`.

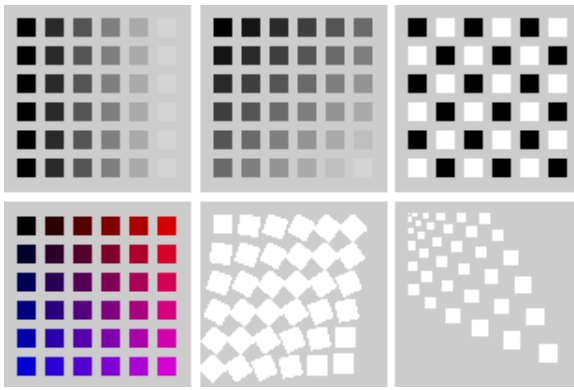


```
size(100, 100, P3D);

translate(7, 7);

//Première boucle (profondeur)
for (int z = 0; z < 6; z = z + 1) {
  translate(0, 0, z * -15); //On recule l'objet sur l'axe z

  //Seconde boucle (hauteur)
```

```
  for (int y = 0; y < 6; y = y + 1) {

    //Troisième boucle (largeur)
    for (int x = 0; x < 6; x = x + 1) {
      rect(x * 15, y * 15, 10, 10);
    }
  }
}
```

Variations

Voici une série de variations des exemples ci-dessous qui utilisent les méthodes `fill()`, `scale()` ou `rotate()`. A vous d'expérimenter les transformations au sein de boucles.



16. LES LISTES

On peut mettre de nombreux *genres* de choses dans une variable : un chiffre, un chiffre à virgule, la phrase d'un texte, voire même toute une image ou tout un morceau de son. Mais bien que les variables puissent théoriquement contenir presque tout type de valeur, elles ne peuvent contenir qu'une seule de ces valeurs à la fois. Dans certains cas, il serait pratique d'avoir plusieurs choses regroupées, au moins du même genre, dans une seule entité. C'est pour cette raison qu'un genre très particulier de variables à été inventé, *les listes*.

Les listes permettent de stocker un nombre fixé d'avance de données ou d'objets dans une même variable. Au lieu de créer 20 variables pour stocker 20 valeurs différentes d'un même genre, nous pouvons créer un seul contenant pour ces 20 valeurs et y accéder une par une via cette seule et unique variable.

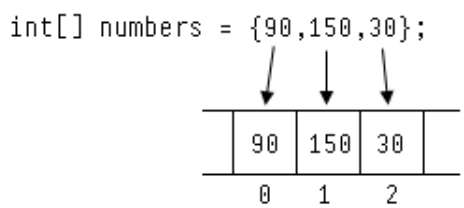
CRÉER UNE LISTE

Si nous utilisons des éléments dits fondamentaux, comme les chiffres, il est très facile de fabriquer une liste :

```
int[] numbers = {90,150,30};
```

Le signe du double crochet signifie qu'il ne s'agit plus d'une variable avec un seul entier à l'intérieur, mais d'une liste d'entiers avec plusieurs valeurs à l'intérieur. Ensuite nous remplissons cette liste dénommée numbers (nous aurions très bien pu lui donner un autre nom) avec les valeurs notées entre les accolades.

L'ordinateur créera assez d'emplacements dans la mémoire et placera chacune des valeurs dans les cases correspondantes :



C'est d'ailleurs pour cette raison qu'il faut indiquer le mot `int`, car Processing a besoin de connaître la taille de chaque case de la liste (dans ce cas précis, nous lui indiquons

à l'aide de cette instruction qu'il s'agit de nombres entiers). S'il s'agissait d'images, comme on le verra plus loin, chaque case de la liste aurait besoin de davantage de place.

Notez que sur certains claviers francophones ou sur certaines plates-formes, il est parfois difficile de localiser le crochet d'ouverture « [» et de fermeture «] ». Sur un clavier français d'ordinateur Apple, par exemple, il faut appuyer en même temps sur les 3 touches {*alt*} + {*maj*} + *(* pour le crochet d'ouverture, et {*alt*} + {*maj*} + *)* pour le crochet de fermeture. Par ailleurs, on doit taper {*alt*} + *(* pour l'accolade d'ouverture et {*alt*} + *(* pour l'accolade de fermeture.

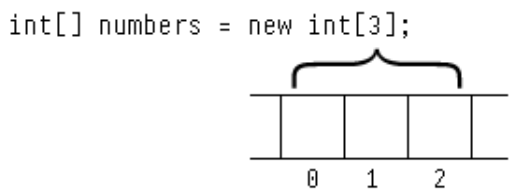
CRÉER UNE LISTE VIDE

Bien qu'elle soit lisible, cette méthode de création directe ne marchera pas avec des éléments comme les sons ou les images. A cause de cette limitation, vous risquez d'utiliser rarement cette méthode.

Déclarer une liste se fera donc le plus souvent de la manière suivante : `type[] nomDeLaListe = new type[NOMBRE D'ÉLÉMENTS]`. L'exemple ci-dessous crée une liste de trois nombres entiers. Attention ! Au moment où la liste dénommée numbers est créée, celle-ci est constituée d'une suite de cases vides qui ne contiennent aucune valeur.

```
int[] numbers = new int[3];
```

Le chiffre 3 entre accolades indique que nous créons une liste de 3 cases actuellement vides.



REMPLIR LA LISTE

Placer des valeurs dans une liste fonctionne de la même manière qu'assigner une valeur à une variable. Il faut en plus préciser à quelle position de la liste on ajoute la valeur.

```
numbers[0] = 90;
numbers[1] = 150;
numbers[2] = 30;
```

La première position de la liste commençant par 0, si nous créons une liste de 3 valeurs, les cases les contenant seront donc numérotées de 0 à 2. On passe d'une position de la liste à une autre en l'incrémentant de 1.

UTILISER LE CONTENU D'UNE LISTE

Utiliser l'élément d'une liste est similaire également à l'utilisation d'une variable. Il faut juste préciser la position de l'élément en question :

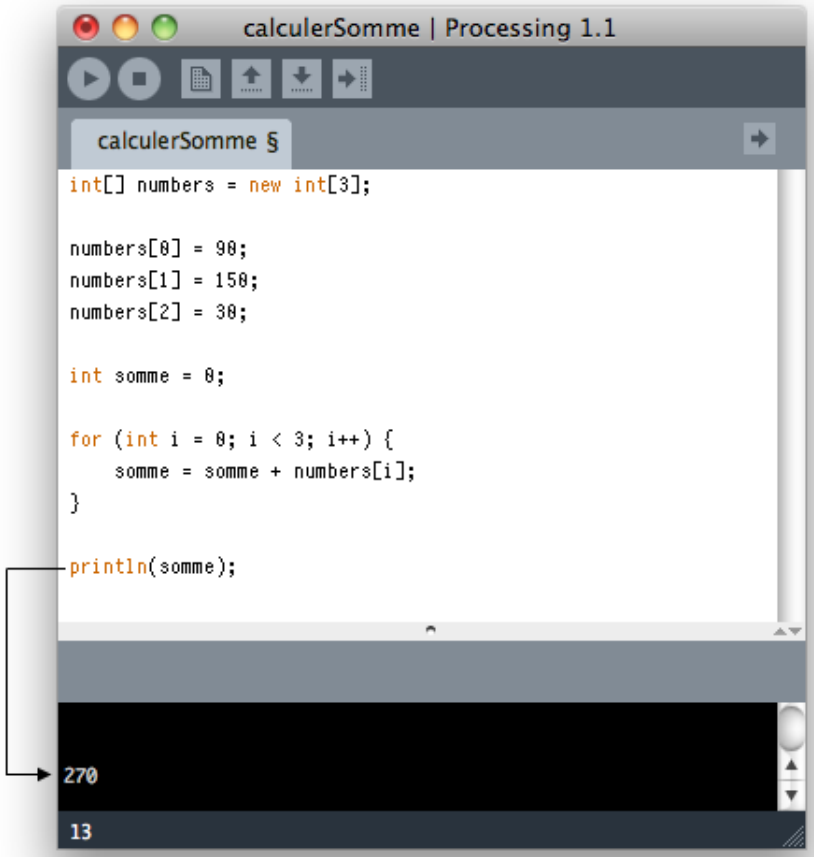
```
println( numbers[0] );
```

A nouveau, notez que l'ordinateur commence à compter à la position 0 et non pas 1. Si nous demandons la valeur `numbers[2]`, Processing nous donnera la valeur à la troisième position et non pas à la deuxième. Un bon moyen pour se rappeler cette particularité, c'est de considérer que Processing compte de la manière suivante : *zérotième, premier, deuxième, troisième,* etc. La *zérotième* valeur concerne la valeur au *début* de la liste.

Dans l'exemple ci-dessous nous utilisons une boucle pour calculer la somme de tous les éléments de la liste déclarée précédemment :

```
int[] numbers = new int[3];

numbers[0] = 90;
numbers[1] = 150;
numbers[2] = 30;

int somme = 0;

for (int i = 0; i < 3; i++) {
    somme = somme + numbers[i];
}

println(somme);
```

Le programme devrait afficher le résultat suivant dans votre console, tout en bas de la fenêtre Processing :



Quelques éléments supplémentaires d'explication pour mieux comprendre le fonctionnement de ce programme. Tout d'abord, nous créons une variable qui contiendra la somme de toutes les valeurs. Elle débute à zéro.

Ensuite, nous enclenchons une boucle qui se répétera 3 fois, en additionnant à chaque fois la prochaine valeur à la somme. Ici nous pouvons voir la relation étroite qui existe entre une liste et une boucle `for()`. La valeur `i` permet de passer une à une dans chacune des valeurs de la liste, en commençant avec la position zéro, ensuite la position 1, et enfin la position 2.

Si vous vous mettez à analyser le code de programmes consultables en ligne, par exemple dans les sketchs librement mis à disposition sur OpenProcessing, vous verrez beaucoup de code avec « `quelqueChose[i]` ».

UNE SUITE D'IMAGES

Un des usages les plus pratiques des *listes* concerne l'importation de médias dans Processing. Imaginons que nous voulons importer cinq images dans Processing destinées ensuite à être utilisées dans un sketch. Sans employer les listes, il faudrait écrire quelque chose comme ceci :

```
PImage photo1;
photo1 = loadImage("photo_1.png");
image(photo1,0,0);

PImage photo2;
photo2 = loadImage("photo_2.png");
image(photo2,50,0);

PImage photo3;
photo3 = loadImage("photo_3.png");
image(photo3,100,0);

PImage photo4;
photo4 = loadImage("photo_4.png");
image(photo4,150,0);

PImage photo5;
photo5 = loadImage("photo_5.png");
image(photo5,200,0);
```

Certes, cet exemple reste encore relativement gérable dans la mesure où vous faites appel à seulement 5 images. Cela étant, les occasions d'erreurs de saisie sont nombreuses. En écrivant ces quelques lignes pour ce manuel, nous-mêmes avons plusieurs fois effectué des erreurs de frappe, notamment en oubliant de changer un chiffre que nous venions de copier d'une ligne précédente !

Une meilleure façon d'écrire cet exemple serait d'utiliser des listes :

```
PImage[] images = new PImage[5];

images[0] = loadImage("image_0.png");
images[1] = loadImage("image_1.png");
images[2] = loadImage("image_2.png");
images[3] = loadImage("image_3.png");
images[4] = loadImage("image_4.png");

image( images[0], 0, 0);
image( images[1], 50, 0);
image( images[2], 100, 0);
```

```
image( images[3], 150, 0);
image( images[4], 200, 0);
```

En utilisant une liste, on peut mettre toutes nos images dans une seule variable qui doit être initialisée qu'une seule fois : `PImage[] images = new PImage[5]`. Ensuite, il suffit de remplir chaque valeur de fichier pour chacun des emplacements dans la liste.

Mais même cette écriture est trop longue. Comment allez-vous faire, par exemple, lorsque vous aurez 200 images, voire 2.000 ? Allez-vous vraiment écrire 200 fois toutes ces lignes ? Et comment ferez-vous lorsque vous voudrez changer le nom des fichiers importés ?

Voici la meilleure manière d'importer une suite d'images :

```
size(500,500);

PImage[] images = new PImage[20];

for(int i=0; i<images.size(); i++) {
  images[i] = loadImage("image_" + i + ".png");
  image( images[i], random(width), random(height) );
}
```

En utilisant une répétition `for()`, nous pouvons désormais importer autant d'images que nous voulons.

Tout d'abord, nous créons la liste d'images. Il s'agit au départ d'une liste vide :

```
PImage[] images = new PImage[20];
```

Ensuite nous récupérons la longueur de cette liste en demandant à la liste elle-même combien d'éléments elle contient via `images.size()` et utilisons cette longueur dans la répétition `for()`.

```
for(int i=0; i<images.size(); i++)
```

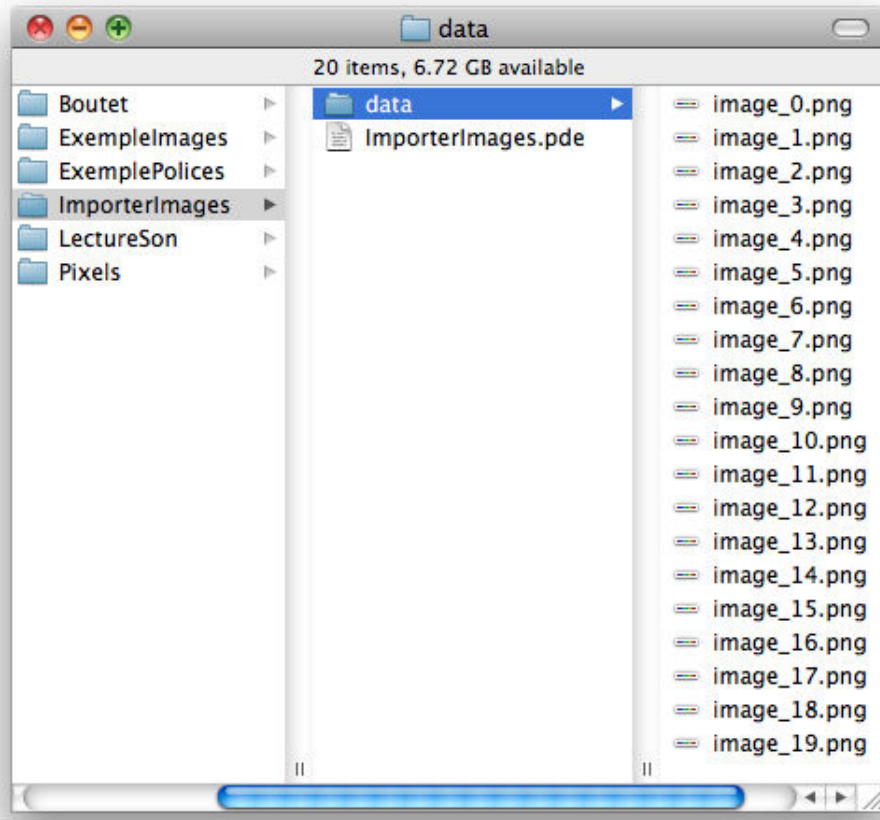
Nous pouvons même automatiser l'importation des images en utilisant la valeur `i` pour composer le nom du fichier des images dans le dossier *data*. En utilisant le signe « + », on peut concaténer deux mots ensemble, ce qui veut dire que nous allons assembler deux mots/signes en un seul message.

```
images[i] = loadImage("image_" + i + ".png");
```

En concaténant les mots « image_ », la valeur de la variable `i`, et « .png », nous obtenons un message qui ressemblerait à quelque chose comme « image_42.png ». De cette manière, nous pouvons utiliser une seule ligne de code pour faire rentrer autant d'images que nous voulons. Si la variable `i` contient la valeur de 9, le nom du fichier

importé sera *image_9.png*. Si la variable `i` contient la valeur 101, le nom du fichier importé sera *image_101.png*.

Pour que cette technique marche, il faut juste préparer au préalable des fichiers d'images dans votre dossier *data*. Voici, par exemple, un dossier data contenant 20 fichiers d'images :



Une fois chacune de ces images importées dans notre programme, nous dessinons l'image quelque part dans le sketch en écrivant :

```
image( images[i], random(width), random(height) );
```

A l'aide de l'instruction `random`, cette ligne de code affiche une image (*0, 1, 2, 3, ...*) qui sera placée à chaque fois de façon aléatoire sur l'axe *x* et sur l'axe *y* de l'espace de dessin. Les deux valeurs *width* et *height* permettent de connaître automatiquement la taille de la fenêtre de visualisation utilisée par le sketch. En faisant appel à elles, on

précise les limites verticales et horizontales maximales où doivent se positionner les images, l'instruction `random()` ayant pour fonction de générer un nombre au hasard ne dépassant pas la valeur mentionnée entre ses parenthèses.

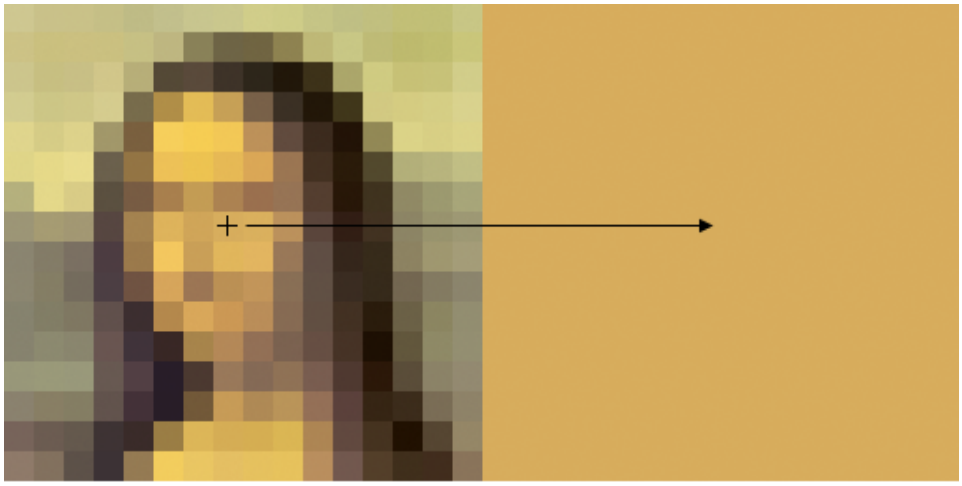
UNE SUITE DE PIXELS

Il se trouve que vous connaissiez déjà les listes et que vous vous en serviez déjà sans le savoir. Par exemple, si vous avez lu le chapitre sur les images vous savez déjà qu'une variable de type `PImage` ne contient pas une seule valeur mais plusieurs, à moins que votre image ne fasse qu'un pixel de large et un pixel de haut. A part cette exception rare, votre image contient un ensemble de valeurs qui représentent l'ensemble des pixels qui doivent former l'image. C'est cet ensemble qui s'appelle une liste.

Vous pouvez accéder directement aux pixels d'une image en demandant un accès direct à sa sous-variable nommée « pixels ».

Imaginons que nous avons au départ l'image d'une mystérieuse dame dans un fichier nommé *lhooq.png*

Si nous importons cette image dans Processing, nous pouvons l'utiliser pour récupérer la couleur d'un pixel en particulier en entrant à l'intérieur de sa liste de couleurs. Cette liste s'appelle `pixels[]`

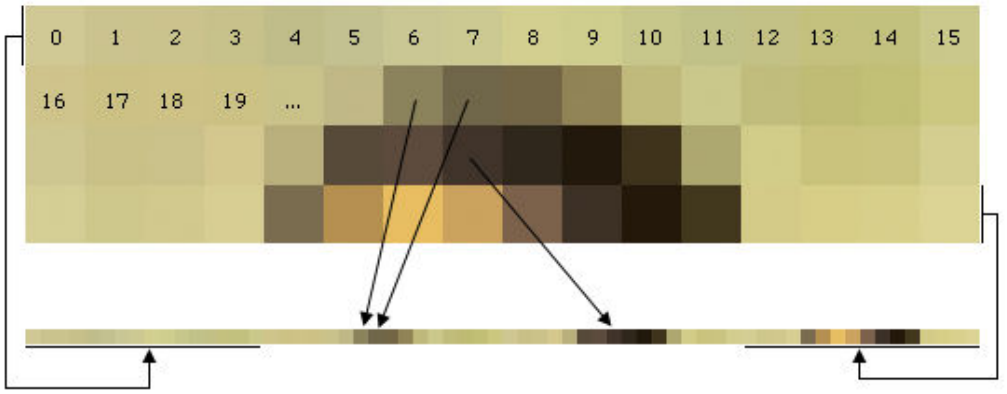


```
size(512,256);

PImage lhooq;
lhooq = loadImage("lhooq.png");
image(lhooq,0,0);

int x = 119;
int y = 119;

int index = x + (y * lhooq.width);
color c = lhooq.pixels[index];

noStroke();
fill(c);
rect(256,0,256,256);

stroke(0);
line(x-5,y,x+5,y);
line(x,y-5,x,y+5);
```

Les deux lignes les plus importantes sont soulignées en gras dans le code :

```
int index = x + (y * lhooq.width);
color c = lhooq.pixels[index];
```

Nous vous rappelons qu'une image n'est rien d'autre qu'une liste de pixels, une image étant notamment composée non pas d'une mais de plusieurs couleurs. Dans Processing, les variables de type `PImage` servent d'ailleurs à stocker les valeurs de ces pixels dans une longue liste linéaire.

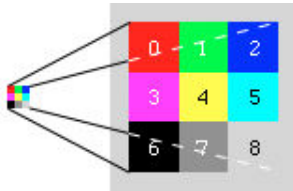


Ce que nous voyons comme une liste à deux {*x*,*y*} dimensions, `PImage` la voit en une seule dimension. Si nous voulons trouver une valeur à la position {*2,1*}, en réalité il faut indiquer que nous voulons la position 17 de l'image. En effet, entre un point sur une ligne et le point correspondant sur la prochaine ligne, il existe 16 pixels. Autrement dit, `PImage` doit compter 16 pixels chaque fois qu'il veut descendre une ligne.

C'est pour cette raison que la formule pour identifier un pixel dans une image s'écrit {*x + (largeur image) * y*}. Pour chaque ligne *y*, il existe une *largeur* de valeurs *x* qu'il faut dépasser.

COLORIER LES PIXELS DANS UNE IMAGE

Nous pouvons aller encore plus loin dans l'utilisation du principe de listes appliqué à la description interne d'une image dans Processing. Il est possible en effet de créer nos propres variables de type `PImage` et d'assigner à ses pixels des couleurs, à l'aide de variables de type `color`. Les pixels de l'image sont accessibles au moyen d'un chiffre numéroté à partir de zéro, comme dans n'importe quelle liste.



Dans cet exemple, on dessine une grille de trois pixels par trois. Nous allons dessiner cette image plus grande en l'étirant à 80 x 80 pixels afin de mieux la voir.

```
PImage img = createImage(3, 3, ARGB);
img.loadPixels();
img.pixels[0] = color(255, 0, 0);
img.pixels[1] = color(0, 255, 0);
img.pixels[2] = color(0, 0, 255);
img.pixels[3] = color(255, 0, 255);
```

```
img.pixels[4] = color(255, 255, 0);
img.pixels[5] = color(0, 255, 255);
img.pixels[6] = color(0, 0, 0);
img.pixels[7] = color(127, 127, 127, 255);
img.pixels[8] = color(255, 255, 255, 0);
img.updatePixels();
image(img, 10, 10, 80, 80);
```

Notez qu'il faut indiquer à Processing que nous allons modifier une image via `loadPixels()` et que nous avons terminé de modifier l'image via `updatePixels()`. En l'absence de ces instructions, on risque de ne pas voir les résultats de notre modification dans la liste.

Notez également que vous pourriez aussi réaliser ce genre de peinture à numéro en utilisant des répétitions.

Les applications créatives des listes sont multiples. Il ne vous reste plus qu'à laisser le champ libre à votre imagination.

17. LES MÉTHODES

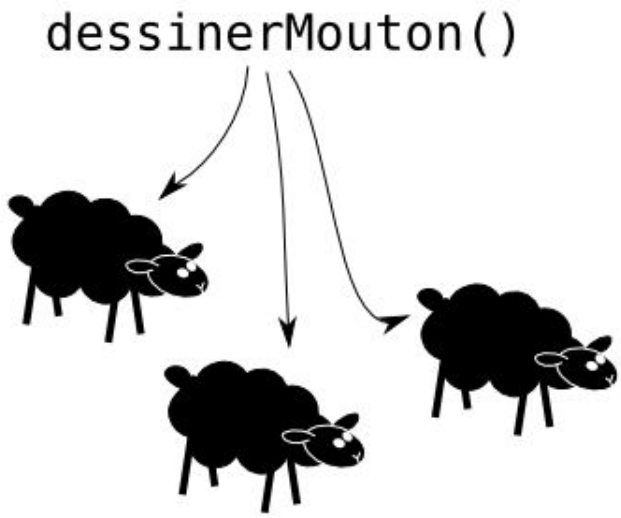
Une méthode est un bloc qui contient une série d'instructions que l'on souhaite réutiliser. L'intérêt des méthodes réside dans la possibilité de réutiliser du code : nous aimerions écrire une seule fois une action tout en pouvant la répéter autant de fois que nécessaire. En englobant notre code dans une méthode, il devient possible d'appeler celle-ci à différents moments de notre programme.

Vous connaissez déjà plusieurs méthodes, mais uniquement en tant qu'utilisateur. Par exemple, `rect()`, `ellipse()`, `line()` `stroke()` sont toutes des méthodes. En créant nos propres méthodes, nous nous rapprochons d'une certaine manière du rôle créatif des concepteurs de Processing : liberté nous est donnée d'inventer des formes ou des fonctionnalités qu'ils n'ont pas pu ou voulu écrire à l'origine.

Voici un exemple purement théorique de ce que la structuration d'un sketch à l'aide de méthodes pourrait éventuellement donner au final si nous devions dessiner un joli paysage.

```
background(255);

joliPaysage();

arbre(cypres, 0,300);
lune(400,100);
gazon(0,300,width,100);

mouton(50,133);
mouton(213,98);
mouton(155,88);
```

L'objectif, c'est de regrouper des éléments complexes du programme dans des mots clés que vous pouvez appeler autant de fois que vous le voulez, tout en les mélangeant avec les mots clés Processing. Ce procédé appelé encapsulation vous éloigne apparemment de votre code, mais c'est pour vous en donner un nouvel accès simplifié en le rendant davantage lisible. Cela permet également d'éviter les répétitions inutiles. Un peu plus d'ordre, pour un peu moins de copier-coller.



MOTS CLÉS

Lorsque nous créons nos propres méthodes, il faut donner à chacune d'entre elles un nom. Une fois la méthode définie, on peut s'en servir dans le programme. Il suffit de l'appeler par son nom.

Processing nous fournit déjà plusieurs méthodes que nous pouvons remplacer par nos propres versions. Ce sera le cas des méthodes `draw()`, `setup()`, `mousePressed()`... que vous découvrirez dans d'autres chapitres. Nous pouvons également créer des méthodes sur mesure en leur donnant le nom de notre choix. Dans ce cas, il faut simplement éviter d'utiliser un nom qui est déjà pris.

DÉCOMPOSER

Jusqu'ici, nous avons programmé dans Processing directement, en commençant à saisir du code depuis le haut du programme et en laissant celui-ci s'exécuter jusqu'en bas. Lorsque nous voulons construire nos propres méthodes, nous devons commencer à décomposer nos programmes en plusieurs parties séparées. Cela nous permettra par exemple d'indiquer les parties qui doivent s'exécuter tout de suite lorsque nous appuyons sur le bouton **run** de celles qui seront appelées par nos propres soins à l'intérieur du programme.

Par contre, en utilisant ce procédé de programmation, nous ne pourrons plus écrire des instructions directement dans Processing sans au préalable les avoir intégrées au sein d'une méthode ou d'une classe. Avec les méthodes, c'est tout ou rien.

VOID SETUP()

Processing nous offre une instruction ayant pour fonction de contenir le code de début de notre programme. Il s'agit de la méthode `setup()` :

```
void setup() {

}
```

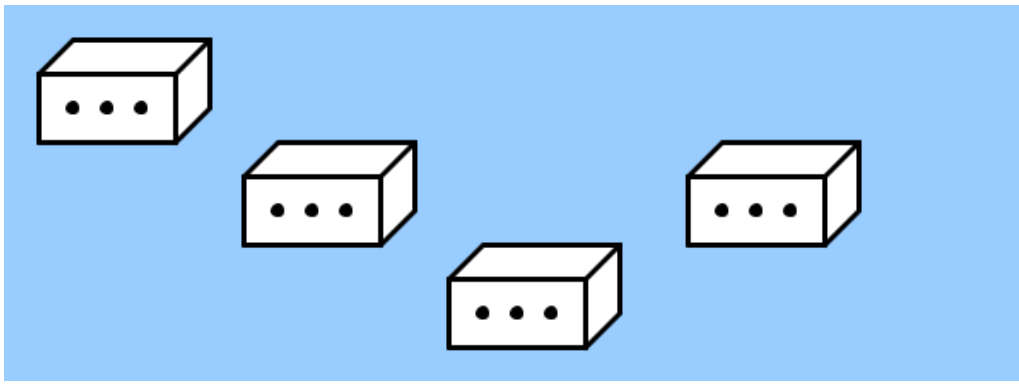
C'est à l'intérieur des accolades de la méthode `setup()` que nous allons placer tout le code qui doit être exécuté au début de notre programme. Pour l'instant, ne cherchez pas à savoir à quoi sert le mot `void`, sachez juste qu'il faut l'écrire, suivi du mot `setup`, puis de parenthèses, et enfin les accolades.

La plupart du temps, nous nous servirons de la méthode `setup()` pour définir la taille de notre sketch. Cette taille ne peut être définie qu'une seule fois — ce qui tombe bien, car le démarrage n'a lieu qu'une seule fois dans le vie d'un programme.

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

CRÉER DES MÉTHODES SUR MESURE

Dans l'exemple qui suit, nous allons créer une méthode `dessinerMouton()` qui contient des instructions pour dessiner un mouton.



En fait, ce mouton est caché dans une boîte, alors on ne voit qu'une boîte ! On y dessine également des trous, afin que le mouton puisse respirer. Nous appelons plusieurs fois cette méthode pour dessiner plusieurs moutons.

Voici le code de ce dessin :

```
void setup() {
```

```
  size(600, 220);
  background(153,204,255);
  smooth();

  // l'appel à notre méthode de dessin d'un mouton
  dessinerMouton();
  translate(120, 60);
  dessinerMouton();
  translate(120, 60);
  dessinerMouton();
  translate(140, -60);
  dessinerMouton();
}


// la méthode pour dessiner le mouton

void dessinerMouton() {

  strokeWeight(3);
  strokeJoin(ROUND);
  stroke(0);
  fill(255);

  rect(20, 40, 80, 40);
  beginShape();
  vertex(20, 40);
  vertex(40, 20);
  vertex(120, 20);
  vertex(120, 40);
  endShape(CLOSE);

  beginShape();
  vertex(100, 40);
  vertex(120, 20);
  vertex(120, 60);
  vertex(100, 80);
  endShape(CLOSE);

  fill(0);
  ellipse(40, 60, 5, 5);
  ellipse(60, 60, 5, 5);
  ellipse(80, 60, 5, 5);
}
```

Le début de ce programme se décrit à l'intérieur de la méthode `setup()`. En effet, puisque nous utilisons une méthode pour dessiner notre mouton, le reste du programme doit également être placé quelque part dans une méthode. En début du programme, nous allons donc saisir :

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

Ensuite, au sein des accolades de la méthode `setup()`, nous définissons la taille de notre sketch et sa couleur de fond.

```
  size(600, 220);
  background(153,204,255);
```

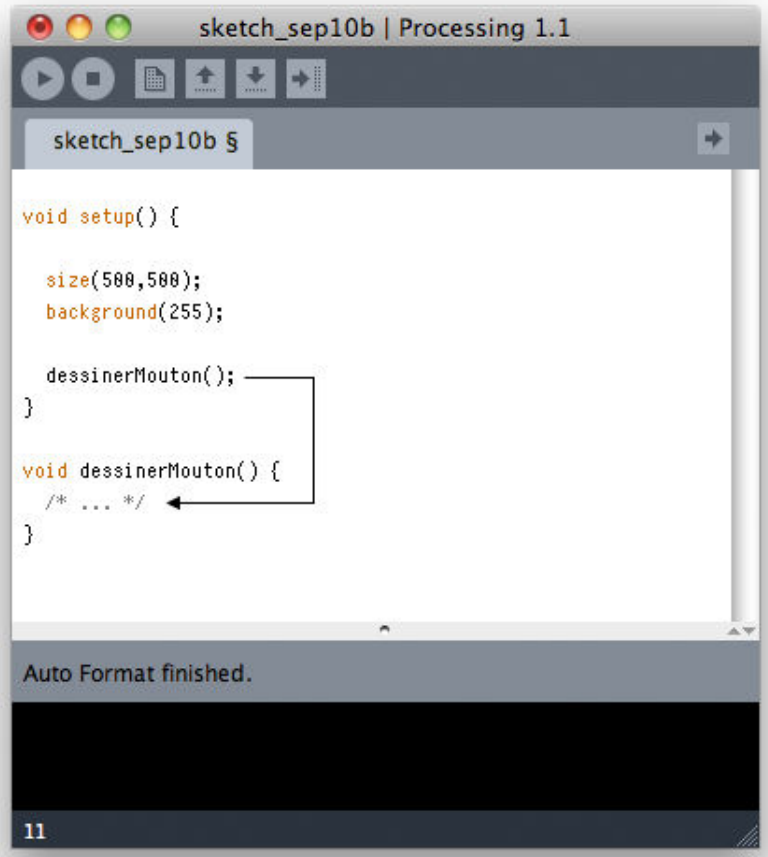
Vous vous êtes peut-être posé la question du rôle de `smooth()` dans notre programme. Optionnelle, cette ligne de code permet toutefois d'améliorer le rendu des lignes en lissant leur tracé : elles apparaissent ainsi plus jolies sur l'écran de l'ordinateur.

```
smooth();
```

Enfin, nous dessinons notre mouton, en faisant appel à une méthode que nous avons définie plus bas dans le programme.

```
  dessinerMouton();
```

Chaque fois que Processing tombe sur le mot `dessinerMouton()`, il vérifie si ce mot existe en tant que méthode quelque part dans le programme. Si cette méthode existe, il fait un détour par cette méthode et fait tout ce qui s'y trouve.



S'il ne trouve pas cette méthode — et qu'elle n'existe pas ailleurs dans la liste des fonctionnalités proposées directement par Processing —, votre programme s'arrêtera avec une erreur d'exécution.

Notez que vous pouvez écrire le mot clé `dessinerMouton()` autant de fois que vous voulez. Ici, dans ce programme, `dessinerMouton()` est écrit au final 4 fois :

```
dessinerMouton();
translate(120, 60);
dessinerMouton();
translate(120, 60);
dessinerMouton();
translate(140, -60);
dessinerMouton();
```

Nous avons placé entre chaque appel à la méthode `dessinerMouton()`, une instruction `translate(x,y)`. Cette instruction nous permet de ne pas dessiner quatre fois le même mouton au même endroit. Ce n'est pas le rôle de ce chapitre de vous expliquer les *transformations* comme `translate()` ; sachez néanmoins que `translate()` sert à déplacer le point d'origine où débutera le tracé d'un dessin.

```
void dessinerMouton() {

  /* ... */

}
```

Enfin nous arrivons à notre méthode `dessinerMouton()` proprement dite. C'est ici que nous dessinons les lignes et formes nécessaires pour obtenir le tracé de notre animal. Nous ne commenterons pas cette partie, puisqu'il s'agit uniquement d'instructions que vous trouverez davantage décrites dans le chapitre sur les formes.

Notez l'usage du mot-clé `void` devant le nom de notre méthode. Cela signifie qu'elle ne retourne rien. En faisant appel à elle, nous savons qu'elle n'a pas pour fonction de nous fournir des données.

LA VALEUR DE RETOUR D'UNE MÉTHODE

Une méthode peut avoir une valeur de retour. Jusqu'ici, nous n'avons pas expérimenté cette particularité. Ni la méthode `setup()`, ni la méthode `draw()` ne retournent une valeur de retour. Le mot `void` a été placé devant chacune de ces deux méthodes pour bien préciser à Processing que rien ne doit être retourné lorsque l'on fera appel à elles.

L'emploi d'une méthode avec une valeur de retour suppose que nous cherchons à obtenir quelque chose d'elle en l'invoquant. Quand nous voulons savoir quelle heure est-il, nous demandons aux méthodes `second()`, `minute()`, ou `hour()` de nous donner en retour leurs valeurs sous forme d'un chiffre entier (`int`). Si ces méthodes ne nous donnaient rien (`void`) en retour, elles ne serviraient pas à grande chose.

Pour les méthodes qui doivent retourner une valeur à celle qui l'appelle, on indique un mot-clé avant pour indiquer le **type** de valeur qui doit être retourné. Une méthode dont le type est `int` nous retourne une valeur de type `int` correspondant à un nombre entier, une méthode dont le type est `float` nous retourne une valeur de type `float` (nombre à virgule), et ainsi de suite.

Voici un exemple de méthode qui nous donne le nombre secondes depuis 00:00:00 ce matin.

```
int secondesAujourdhui() {
  return hour() * 3600 + minute() * 60 + second();
}
```

```
void draw() {
  println( secondesAujourdhui() );
}
```

Même si vous ne connaissez pas la méthode `draw()`, amusez-vous néanmoins à exécuter ce mini-programme et regardez les informations qui s'affichent dans la console située en bas de votre fenêtre d'édition de Processing. Vous verrez que la méthode `draw()` appelle en permanence la méthode `secondesAujourdhui()` et utilise le résultat de cette méthode pour nous afficher les secondes.

LES PARAMÈTRES D'UNE MÉTHODE

Une méthode peut accepter des paramètres. La plupart du temps, on les appelle des arguments. Ces paramètres doivent avoir chacun un type et un nom, tout comme les variables.

Pour appeler une méthode, on écrit son nom, et on le fait suivre d'une parenthèse ouvrante et d'une autre fermante. Entre ces parenthèses, on place les paramètres de la méthode. Ce qu'on y met sera envoyé dans la méthode.

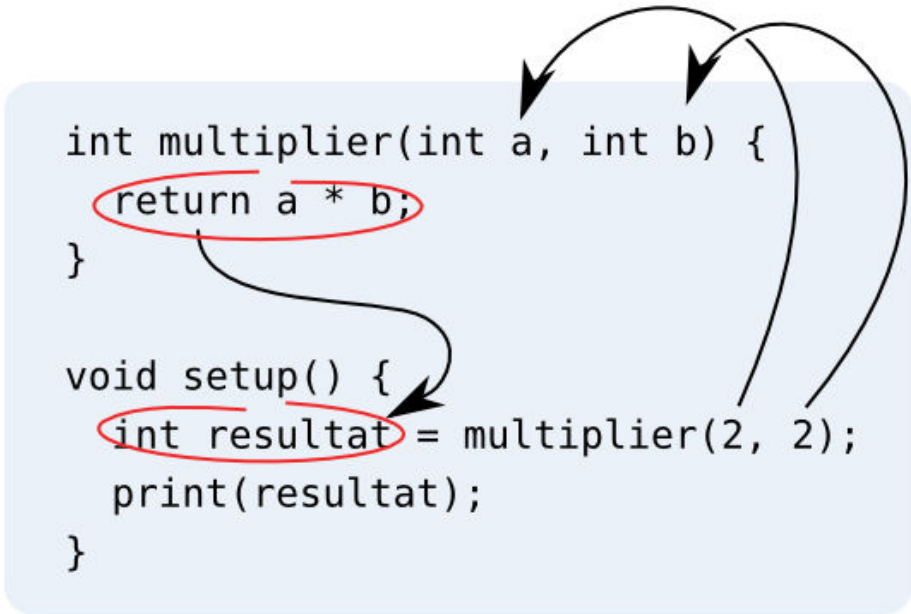
```
multiplier(2, 2);
```

Une fois arrivé dans le corps de la méthode, Processing peut accéder à leur valeur, comme il le fait avec les variables.

Ainsi dans l'exemple ci-après, lorsque cette méthode est appelée avec les argument 2 et 2, la valeur de `a = 2`, et celle de `b = 2` également. La valeur de retour de cette méthode sera donc 4 (2 fois 2 égale 4).

```
int mutliplier(int a, int b) {
  return a * b;
}
```

Notez que c'est la position des arguments qui détermine quelle valeur sera affectée à quel argument.



Pour résumer notre exemple, on a créé une méthode qui retourne le résultat de la multiplication de ses deux arguments. Un commentaire précède la définition de la méthode (une bonne pratique de programmation pour se rappeler ultérieurement de la fonction d'un morceau de code).

```
/*
 * Retourne le résultat de la multiplication de ses
 * deux arguments.
 */
int multiplier(int a, int b) {
  return a * b;
}

void setup() {
  int resultat = multiplier(2, 2);
  print(resultat);
}
```

La console de Processing affichera :

```
4
```

LA PORTÉE DES VARIABLES

Profitons de ce chapitre sur les méthodes et les variables pour vous mettre en garde contre une erreur classique qui peut survenir lorsque l'on utilise des variables et des méthodes dans un programme.

Les variables — que ce soit des objets ou des types fondamentaux de données — ne sont pas forcement accessibles à l'ensemble de votre programme ! Tout dépend de l'endroit où elles ont été déclarées. Une variable déclarée à l'intérieur d'une méthode ne sera accessible que dans celle-ci :

```
void setup() {
  int x = 10;
}

void draw() {
  /* Le programme générera une erreur car la variable x
   * n'existe qu'à l'intérieur de la méthode setup()
   */
  x = x + 1;
}
```

En lançant l'exécution du programme ci-dessus, la console de Processing affichera le message d'erreur suivant :

```
The field Component.x is not visible
```

Pour qu'une variable soit accessible à tout votre programme il faut la déclarer en en-tête comme ceci :

```
int x;

void setup() {
  x = 10;
}

void draw() {
  x = x + 1;
}
```

18. LES OBJETS

La **programmation orientée objet** (POO) permet de structurer son programme à partir des **éléments concrets** présents dans l'espace de dessin (balles, murs, personnages, etc.). Un objet est un modèle qui peut être dupliqué et dont **chaque copie est unique**. Ces 2 notions constituent les briques de base d'une application structurée. Un objet est composé de **caractéristiques** (propriétés) et d'**actions** (méthodes). Chaque **instance** (chaque copie unique) d'un objet possède sa vie propre avec des caractéristiques spécifiques tout en pouvant effectuer potentiellement les mêmes actions que ses sœurs (les autres instances du même objet).

Ne vous inquiétez pas si ces définitions vous paraissent obscures à première lecture. Au fil de ce chapitre, nous allons détailler ces notions à l'aide d'un exemple simple et concret : le tracé d'une puis de deux balles (de simples cercles) à l'écran. Pour réaliser ce dessin, nous faisons appel à un modèle de balle (l'objet) et à ses copies (les instances) qui possèdent chacune des caractéristiques différentes.

Le modèle de la balle est composé des **caractéristiques** (variables) suivantes :

- position sur l'axe x
- position sur l'axe y
- couleur

Le modèle de balle contiendra l'**action** (méthode) suivante :

- afficher

QUAND CRÉER UN OBJET

Comment faire pour dessiner plusieurs balles à l'écran? La solution la plus évidente semble être de dupliquer les variables qui caractérisent la balle et les instructions permettant de l'afficher et de la gérer. La taille du code sera proportionnelle au nombre d'éléments visibles à l'écran. Plus j'ai de balles plus mon code sera long.

Ce procédé pose deux problèmes :

- si l'on veut modifier ou ajouter des actions à l'ensemble de ces balles, on devra modifier autant de fois le code qu'il y a d'éléments affichés,
- au fur et à mesure que l'on rajoute des éléments à l'écran le programme s'allonge au point de devenir ingérable (et si notre programme comportait 1'000'000 de balles ?).

Pour pallier à ces limitations, nous allons transformer notre balle en **objet**. Dès qu'une entité (balle, avatar, forme, etc.) de votre programme devient **trop complexe** ou qu'elle doit exister en **plusieurs exemplaires**, il faut en faire un objet.

CRÉER UN OBJET

La création d'un objet se passe en deux étapes : la définition du modèle de l'**objet** et la création d'une **copie unique** de l'objet (instance).

Le modèle

Processing utilise le mot clé `class` pour définir un objet. Sa syntaxe s'apparente à la définition d'une méthode : `class nomObjet {}`. Toutes les **caractéristiques** et les **actions** sont écrites à l'intérieur. En général on va écrire la définition d'un **objet** tout à la fin de notre code. Éventuellement si l'objet est complexe, on créera un nouvel onglet dans la fenêtre d'édition de Processing (un nouveau fichier) afin de le séparer du reste du code.

```
class Balle {
}
```

L'instance

Une fois le modèle défini, il faut créer une copie de ce **modèle** qui sera unique. Processing utilise le mot clé `new` pour créer une **instance** d'un **objet** : `new nomObjet();`. Il faut stocker cet **objet** dans une **variable** afin de pouvoir le manipuler ultérieurement.

```
Balle maBalle = new Balle();
```

Dans l'exemple ci-dessus, nous déclarons une **variable** `maBalle` et nous lui assignons une copie de l'**objet** Balle. `maBalle` fait référence à cet **objet** et permet d'agir sur lui dans la suite du programme. Afin de rendre l'**objet** disponible dans tout le programme, nous plaçons la déclaration en en-tête de l'application.

LES CARACTÉRISTIQUES

Les **objets** ont des **caractéristiques** qui les définissent et les rendent uniques. Ce sont des **variables** qui sont déclarées au début de l'**objet**.

```
class Balle {
  //Déclaration des paramètres de base de la balle
  float x;
  float y;
  color couleur;
}
```

Pour modifier une **caractéristique** après la création de l'**objet**, il faut procéder de la manière suivante: `nomDeLInstance.nomDeLaCaractéristique = valeur;`
Par exemple :

```
maBalle.x = 100;
```

LE CONSTRUCTEUR

Le **constructeur** est une **méthode** appelée lorsque l'**objet** est créé. Il est l'équivalent de la **méthode** `setup()` de l'application. Il porte toujours le nom de l'**objet**. Le constructeur va prendre un certain nombre de variables en **paramètre** et les assigner à l'objet :

```
class Balle {
  //Déclaration des paramètres de base de la balle
  float x;
  float y;
  color couleur;

  Balle (float nouvX, float nouvY, color nouvCouleur) {
    x      = nouvX;
    y      = nouvY;
    couleur    = nouvCouleur;
  }
}
```

Lorsque nous allons créer l'instance (une copie) de l'**objet**, nous allons directement lui donner ses caractéristiques propres en paramètre. L'exemple ci-dessous crée une balle blanche placée aux coordonnées 100, 100 de la fenêtre de visualisation.

```
maBalle = new Balle(100, 100, color(255));
```

Attention ! Nous venons uniquement de créer une copie du modèle de l'**objet**. Il n'est pas encore affiché à l'écran.

LES ACTIONS

Les actions d'un **objet** représentent les différentes choses qu'il peut effectuer. Ce sont des **méthodes** mentionnées (déclarées) à l'intérieur de l'**objet**. Appeler une action sur une instance d'un **objet** se fait de la manière suivante:
`nomDeLInstance.nomDeLaMethode();`

Dans notre exemple, notre balle comportera une seule **action** : être affichée. Nous allons utiliser les **instructions** `fill()` et `ellipse()` pour la dessiner.

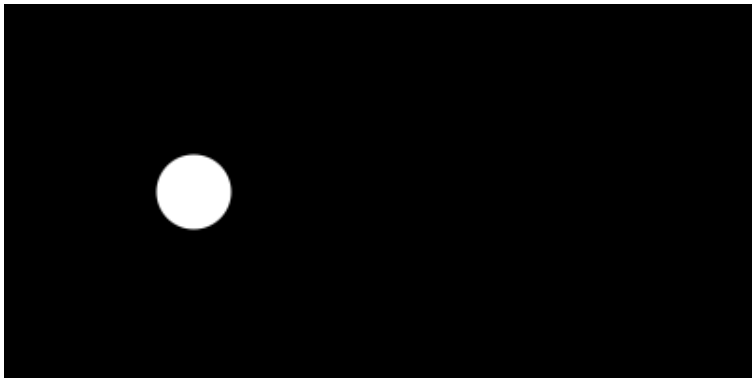
```
class Balle {
  //Déclaration des paramètres de base de la balle
```

```
  float x;
  float y;
  color couleur;

  Ball (float nouvX, float nouvY, color nouvCouleur) {
    x       = nouvX;
    y       = nouvY;
    couleur     = nouvCouleur;
  }

  void display() {
    fill(couleur);
    ellipse(x, y, 40, 40);
  }
}
```

Pour afficher notre balle, il faut appeler sa **méthode** `display()` depuis sa copie dans la **méthode** `draw()` du programme.

```
void draw() {
  maBalle.display();
}
```

PROGRAMME FINAL

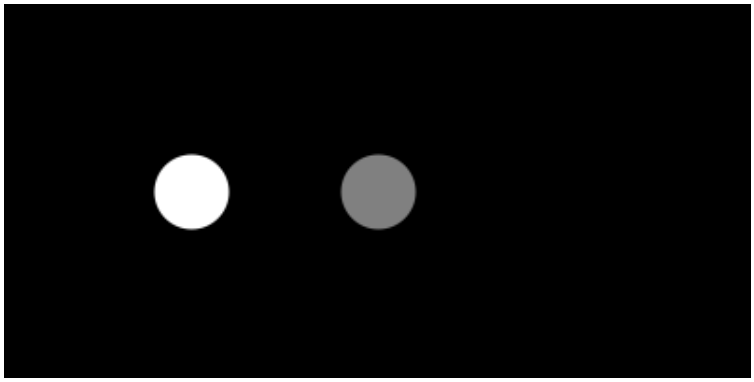


```
//Déclaration et création d'une instance de l'objet Balle
Balle maBalle = new Balle(100, 100, color(255));

void setup() {
  smooth(); //Lissage des dessins
  size(400, 200); //Taille de la fenêtre
}

void draw() {
```

```
  background(0); //On dessine un fond noir
  noStroke(); //On supprime le contour

  maBalle.display(); //Affichage de la balle
}

class Balle {
  //Déclaration des paramètres de base de la balle
  float x;
  float y;
  color couleur;

  //Constructeur de la balle
  Balle (float nouvX, float nouvY, color nouvCouleur) {
    x          = nouvX;
    y          = nouvY;
    couleur    = nouvCouleur;
  }

  //Dessin de la balle
  void display() {
    fill(couleur);
    ellipse(x, y, 40, 40);
  }
}
```

MULTIPLIER LES BALLES

Créer une seconde balle devient, grâce au système des objets, un jeu d'enfant. Il suffit de déclarer une deuxième balle, par exemple `maBalle2` et de l'afficher.



```
//Déclaration et création de plusieurs instances de l'objet Balle
Balle maBalle1 = new Balle(100, 100, color(255));
Balle maBalle2 = new Balle(200, 100, color(128));
```

```
void setup() {
  smooth(); //Lissage des dessins
  size(400, 200); //Taille de la fenêtre
}

void draw() {
  background(0); //On dessine un fond noir
  noStroke(); //On supprime le contour

  maBalle1.display(); //Affichage de la balle 1
  maBalle2.display(); //Affichage de la balle 2
}

class Balle {
  //Déclaration des paramètres de base de la balle
  float x;
  float y;
  color couleur;

  //Constructeur de la balle
  Balle (float nouvX, float nouvY, color nouvCouleur) {
    x          = nouvX;
    y          = nouvY;
    couleur    = nouvCouleur;
  }

  //Dessin de la balle
  void display() {
    fill(couleur);
    ellipse(x, y, 40, 40);
  }
}
```

N'hésitez pas à vous approprier ce programme en ajoutant autant de balles que vous voulez.

19. LES COMMENTAIRES

Commenter et documenter son programme sont essentiels pour maintenir un code clair et pour faciliter la collaboration avec d'autres personnes. Cette pratique de programmation permet également de se rappeler ultérieurement l'utilité de telle ou telle variable, méthode, etc. Au moment de la conception d'un programme, on possède une vue d'ensemble du code et parfois commenter notre sketch peut nous sembler inutile. Pourtant, lorsqu'on veut retravailler ou réutiliser ce code quelques jours, voire quelques mois plus tard, l'usage de commentaires nous permet de nous y replonger plus rapidement.

Processing offre deux manières distinctes de commenter son code : les commentaires en ligne, et les blocs de commentaires multi-lignes.

COMMENTAIRES EN LIGNE

Pour écrire un commentaire qui ne prend qu'une seule ligne, il suffit de placer les caractères // au début du commentaire. Tout le texte qui est écrit à sa droite sur la même ligne sera considéré comme commentaire et donc ne sera pas pris en compte par le programme lors de son exécution. Exemples de commentaires en ligne :

```
void setup() {
  // Définit la taille du programme
  size(400, 300);
  smooth(); // Active le lissage des contours
}
```

BLOCS DE COMMENTAIRES

Si une explication nécessite un commentaire plus long, on peut l'écrire sur plusieurs lignes en le plaçant entre les caractères /* et */. Par exemple :

```
/*
La méthode setup initialise le programme,
on peut y définir la taille de la fenêtre,
définir l'état initial du programme, etc.
*/
void setup() {

}
```

On peut commenter un bloc de ligne en allant dans le menu *edit* de Processing puis en cliquant sur *comment/uncomment*. On peut procéder de même pour enlever le commentaire sur un bloc.

UTILISATION JUDICIEUSE DES COMMENTAIRES

Commenter un programme ne signifie pas qu'il faille écrire le but de chaque ligne ! De manière générale, on inscrira un bloc de commentaires avant les méthodes pour expliquer leur utilité et éventuellement la façon de s'en servir. On évitera de commenter les méthodes de base de Processing telles que `setup() {}`, `draw() {}`, etc. On ne commentera pas non plus les instructions génériques telles que `size()`, `fill()`, `ellipse()`, etc. Leur utilité est évidente dès lors que vous êtes relativement familiarisé avec Processing.

Lorsqu'un ensemble de lignes effectue une action commune (par exemple modifier deux variables x et y), on évitera de mettre un commentaire pour chaque ligne et on utilisera de préférence un bloc de commentaires. Par exemple :

```
x = x + 10; //Ajout de 10 à la coordonnée x
y = y + 10; //Ajout de 10 à la coordonnée y
```

pourrait être écrit :

```
//Modification des coordonnées x et y
x = x + 10;
y = y + 10;
```

ANIMER

20. La méthode draw
21. La ligne de temps
22. L'animation d'un objet
23. L'animation de plusieurs objets
24. La lecture du son

20. LA MÉTHODE DRAW

Jusqu'ici, nous avons créé ce que nous pouvons appeler des programmes linéaires : on démarre le programme, celui-ci exécute notre dessin, et tout s'arrête à la fin des instructions de notre sketch, en bas de la fenêtre d'édition.

Mais Processing n'est pas uniquement un environnement de dessin écrit, c'est également un environnement interactif. Et pour qu'il y ait interactivité, il nous faut du temps, en d'autres termes, un moyen de prolonger notre dessin pour que celui-ci puisse se modifier en suivant chronologiquement certaines étapes, selon différents facteurs et conditions. C'est le rôle de la *boucle infinie*, appelée en permanence par la machine pour réactualiser notre dessin. Dans Processing, cette boucle infinie s'écrit `draw()`. Elle est souvent accompagnée par la méthode `setup()` qui permettra de préparer la fenêtre de visualisation (l'espace de dessin), par exemple en lui donnant une taille au départ.

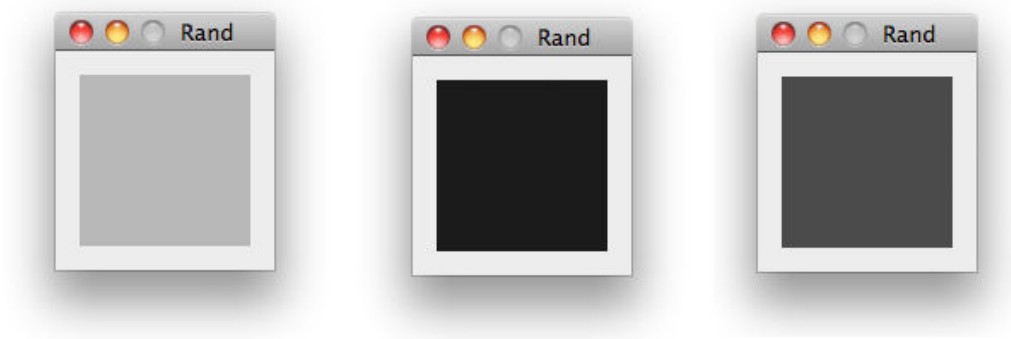
C'est l'absence de ces deux méthodes qui rend Processing inerte. Si votre code ne dispose pas de méthode `draw()`, l'exécution du programme s'arrêtera à la fin du code de votre programme.

DRAW()

Commencez un nouveau programme Processing vide, tapez les lignes suivantes dans la fenêtre d'écriture et appuyez sur le bouton run :

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

Ça y est, vous avez activé l'animation dans Processing. Vous devriez maintenant voir une fenêtre qui clignote 30 fois par seconde avec une couleur grise aléatoire quelque part entre le noir et le blanc. C'est la méthode `random(255)` qui donne en retour une valeur aléatoire entre 0 et 255. C'est cette valeur qui est ensuite récupérée par les parenthèses de la méthode `background()` et appliquée sur le fond de la fenêtre de visualisation de l'espace de dessin. Comme le tout se passe de manière répétée, on a l'impression d'assister à une animation.



Par défaut, les instructions qui se trouvent entre les deux accolades de la méthode `draw()` seront appelées 30 fois par seconde. 30 fois par seconde Processing ouvrira cette méthode et regardera ce qui est écrit dedans pour l'exécuter. Nous pouvons mettre autant d'instructions que nous voulons à l'intérieur de cette méthode. Ces instructions seront jouées par Processing de manière cyclique, tel un métronome.

Attention aux erreurs d'écriture, car dès que Processing en rencontrera une, votre programme s'arrêtera net — c'est la fin de votre animation. En revanche, si vous avez écrit une méthode `draw()` sans erreurs, elle sera appelée en boucle, 30 fois par seconde, jusqu'à ce que l'utilisateur arrête le programme, qu'une panne d'électricité arrive, ou que la fin du monde se produise. C'est donc grâce à cette méthode exécutant du code en répétition que nous allons pouvoir créer des animations.

FRÉQUENCE D'EXÉCUTION

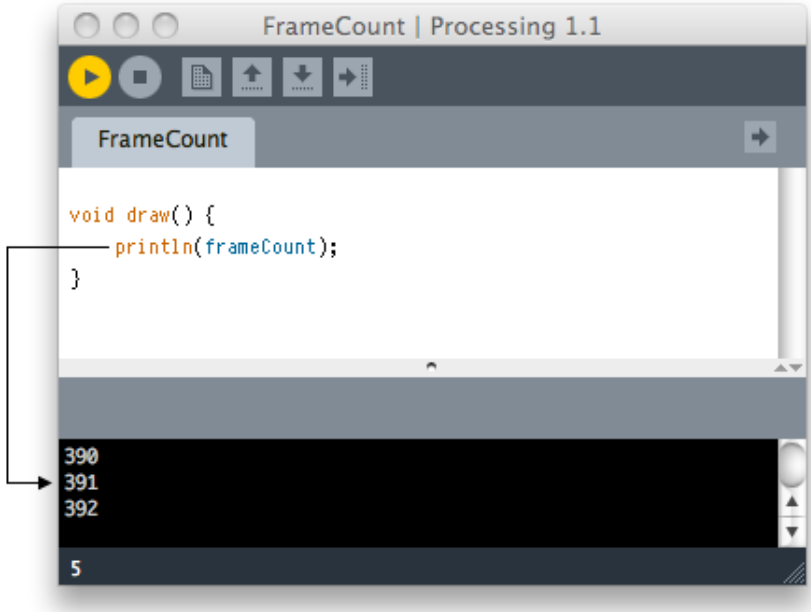
Il est possible de spécifier une valeur différente à notre métronome en utilisant la méthode frameRate().

Si vous modifiez l'exemple précédent comme suit, vous remarquerez que la vitesse de l'animation a été diminuée (divisée par 3).

```
void draw() {
   frameRate(10);
   background( random(255) );
}
```

NOMBRE DE FOIS OÙ DRAW() A ÉTÉ APPELÉE

Processing peut aussi compter le nombre de fois que cette méthode draw() a été appelée, depuis le lancement du programme via la variable frameCount.



SETUP()

Souvent, voire la plupart du temps, il est nécessaire de placer certaines instructions au tout début du programme. C'est la nature et la portée de ces instructions sur le programme qui nous incitent à les placer à cet endroit. Par exemple, dans Processing, la taille de la fenêtre de visualisation du dessin ne peut être définie qu'une seule fois dans un sketch ; ce paramétrage de l'espace de dessin ne peut donc être placé à l'intérieur de la méthode `draw()` car cette méthode s'exécute plusieurs fois durant le déroulement du programme.

A moins de se satisfaire de la dimension par défaut de 100x100 pixels, définir la taille de la fenêtre de visualisation peut s'avérer très utile. C'est pour toutes ces raisons qu'une deuxième méthode complémentaire a été créée : la méthode `setup()`.

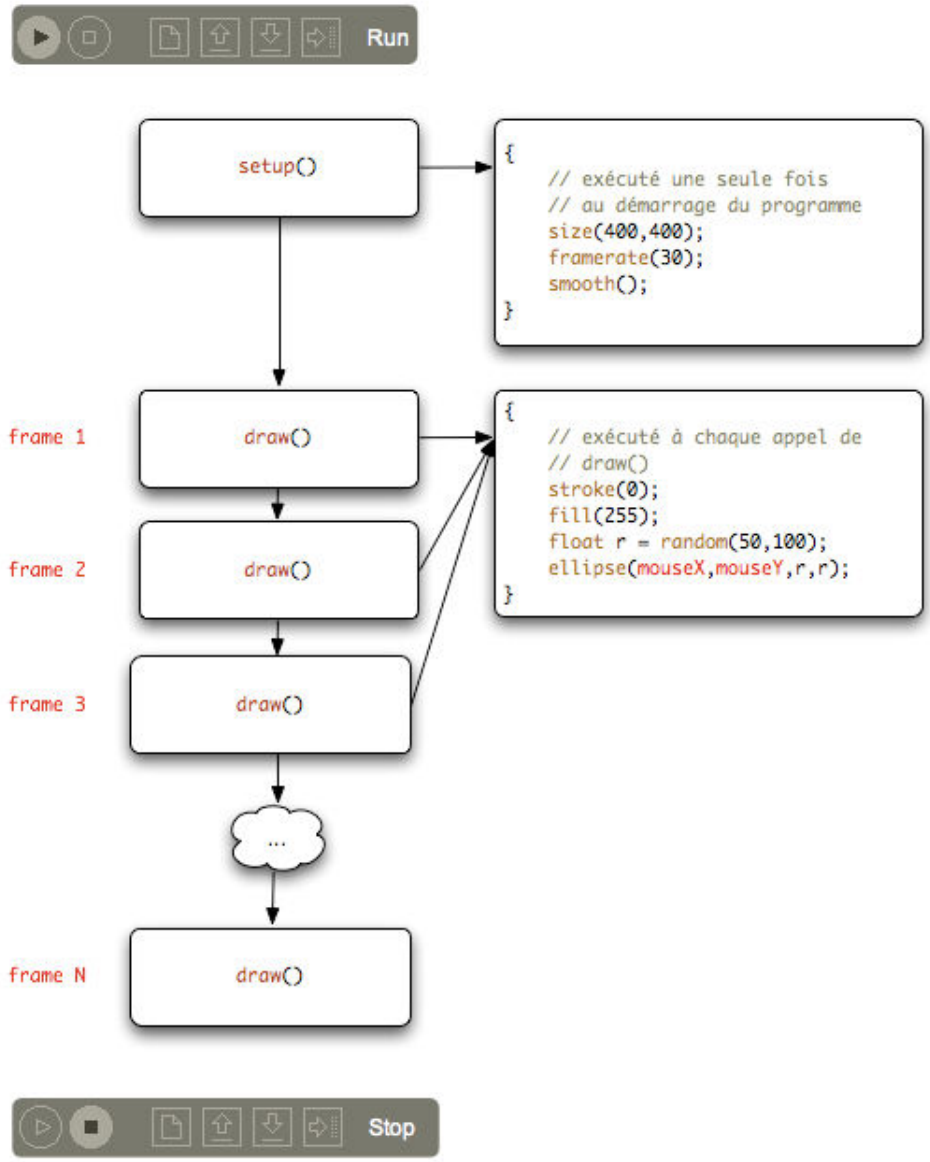
```
void setup() {

}
```

Les subtilités des différents signes qui composent cette méthode sont pour l'instant sans importance. Sachez simplement qu'il faut écrire `void setup()` tout au début, suivi d'une ouverture d'accolades, saisir ensuite les instructions que vous voulez exécuter, et enfin terminer avec la fermeture des accolades.

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

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

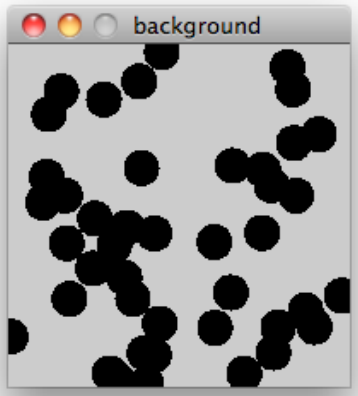
Lorsque ce sketch est lancé, Processing exécute tout d'abord les instructions qui se trouvent à l'intérieur de la méthode `setup()`. Ensuite, la méthode `draw()` commencera à être appelée de manière répétée, telle un métronome.

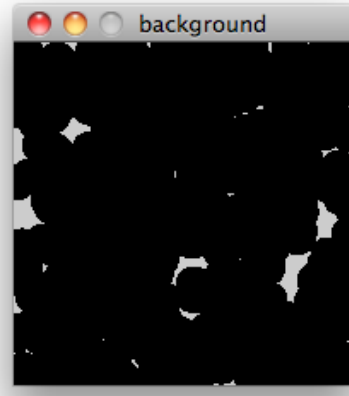
Ce principe de fonctionnement est illustré par le schéma suivant :



BACKGROUND()

Voici un programme qui remplit progressivement l'écran avec des ellipses.






```
void setup() {
  size(200, 200);
  fill(0);
}

void draw() {
   ellipse(random(200), random(200), 20, 20);
}
```

Comme vous pouvez le constater dans les captures d'écran, cette animation finira par remplir notre espace de dessin complètement par du noir. C'est peut-être l'œuvre conceptuelle ultime de toute notre carrière artistique ; néanmoins il serait bien utile d'apprendre à animer une seule ellipse, la voir évoluer dans l'espace ou changer de forme, sans que ses positions antérieures soient affichées en même temps.

Dans ce cas, il suffit d'ajouter un nettoyage de fond de l'espace de dessin à l'aide de la méthode `background()`. Celle-ci prend une à trois valeurs de couleur, comme pour la méthode `fill()` ou `stroke()`.

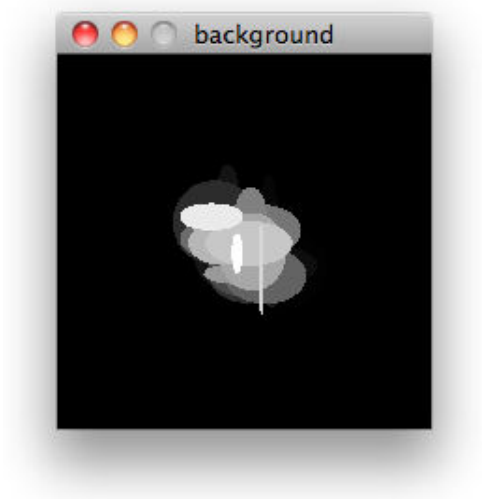
Voici une modification du programme qui permettra d'afficher maintenant une seule forme animée :

```
void setup() {
  size(200, 200);
  fill(0);
}

void draw() {
  background(255);
  ellipse( 100, 100, random(100), random(100));
}
```

En réalité, ce que nous demandons au programme, c'est 30 fois par seconde d'effacer l'intégralité de notre dessin et de tracer une nouvelle forme par dessus avec un nouveau fond blanc.

AJOUTER UN FONDU

Il existe une astuce, souvent utilisée dans la communauté des utilisateurs de Processing, qui consiste à effacer le fond de l'image à l'aide d'un rectangle semi-transparent plutôt qu'avec l'instruction `background()`. Ce procédé permet d'obtenir un effet d'effacement graduel, de fondu.




```
void setup() {
  size(200,200);
  background(0);
  noStroke();
}

void draw() {
  fill(0, 0, 0, 20);
  rect(0, 0, 200, 200);
  fill(255);
  ellipse(100 + random(-20,20), 100 + random(-20,20), random(50), ran
}
```

21. LA LIGNE DE TEMPS

Pour créer une animation, il faut qu'il y ait du mouvement. Le mouvement implique un changement du dessin dans le temps, par exemple une modification de la position ou de la couleur d'un de ces éléments graphiques. Les informations liées à ces changements peuvent être stockées dans des variables.

Pour créer des animations, il faut savoir à quel moment nous nous trouvons par rapport à une ligne de temps. Pour ce faire, nous pouvons soit utiliser l'heure qu'il est, soit compter (par exemple de un à dix).

QUELLE HEURE EST-IL?

Nous allons créer une horloge en appelant les méthodes `hour()`, `minute()` et `second()` de Processing. Nous allons utiliser le résultat de l'appel de ces méthodes pour faire varier la position sur l'axe horizontal de trois minces rectangles.

Le code pour réaliser cette horloge est très simple :

```
void setup() {
  size(60, 60);
  noStroke();
}

void draw() {
  background(0);

  // Les heures vont de 0 à 23, nous les convertissons à une
  // échelle de 0 à 60

  rect((hour() / 24.0) * 60, 0, 1, 20);
```

```
  rect(minute(), 20, 1, 20);
  rect(second(), 40, 1, 20);
}
```

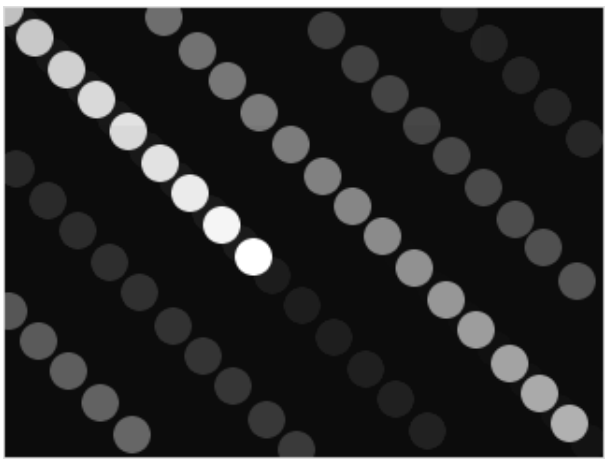
Les images sont dessinées les unes par-dessus les autres. L'appel à l'instruction `background(0)` remplit l'espace de dessin avec la couleur définie en paramètre (le nombre 0 correspondant à du noir) ce qui revient à chaque fois à effacer l'image précédente.

MESURER LE TEMPS QUI PASSE

On peut obtenir le temps qui s'est écoulé depuis le début de l'exécution d'un sketch en utilisant la méthode `millis()`. Celle-ci retourne un nombre en millisecondes. Cette valeur peut être exploitée pour créer des animations. Il y a mille millisecondes dans une seconde : une précision suffisante pour animer des formes ou des sons.

Par ailleurs, vous pouvez également avoir besoin de créer des animations cycliques dont la fréquence (le nombre d'animations par cycle) s'adapte en fonction d'autres éléments, par exemple la taille de la fenêtre de visualisation de l'espace de dessin. Pour déterminer à partir d'une valeur donnée ce nombre d'animations par cycle, la solution la plus simple consiste à utiliser l'opérateur modulo `%`. En mathématique, le modulo permet d'obtenir le reste d'une division par un nombre. Par exemple, l'expression `println(109 % 10);` affichera 9, car 109 divisé par 10, donne 10 comme quotient et 9 comme reste. De manière plus générale, si on prend deux nombres x et y, le reste de la division de x par y est strictement inférieur à y. L'opérateur modulo nous permet donc de compter sans jamais dépasser un certain nombre (la base du modulo).

Dans le prochain exemple, nous allons utiliser la méthode `millis()` et l'opérateur `%` pour dessiner un cercle qui parcourt très rapidement le sketch selon des trajectoires diagonales. Ce cercle laisse derrière lui une traînée qui s'efface graduellement.



Le code de cette animation est le suivant :

```
void setup() {
  size(320, 240);
  noStroke();
  frameRate(60);
  smooth();
}

void draw() {
  fill(0, 0, 0, 10);
  rect(0, 0, width, height);
  fill(255);
  ellipse(millis() % width, millis() % height, 20, 20);
}
```

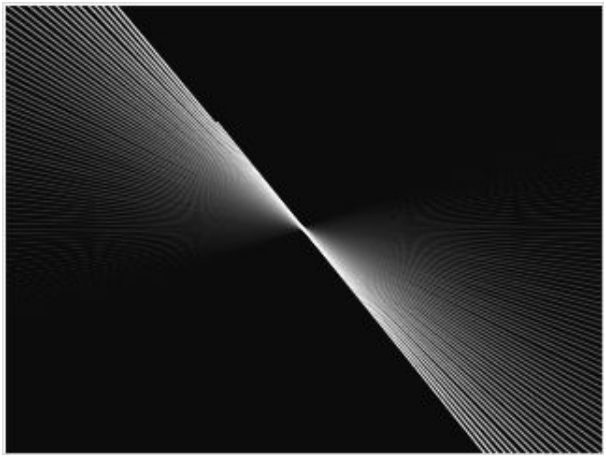
L'effacement du dessin précédent s'effectue à l'aide d'un rectangle semi-transparent. Ce procédé permet d'obtenir un effet de flou de mouvement contrairement à l'instruction `background()`. La présence de l'instruction `smooth()` au début du programme s'explique par notre amour des courbes lisses.

Un conseil : basez vos animations sur la mesure du temps plutôt que sur le décompte du nombre d'images dessinées. La vitesse de rendu peut très sensiblement se dégrader lorsque les capacités de l'ordinateur sont occupées à gérer un compteur : les animations deviennent alors moins fluides, plus saccadées. Ces limitations de performance n'existent pas lorsque l'on utilise les instructions associées à l'horloge interne de l'ordinateur pour mesurer le temps qui passe.

Pour satisfaire votre curiosité et malgré notre mise en garde sur l'utilisation de la méthode du décomptage du nombre d'images dessinées, nous allons néanmoins voir comment créer des animations à l'aide de compteurs, le principe de mis en oeuvre étant très simple

ANIMER À L'AIDE D'UN COMPTEUR

Nous allons maintenant voir comment créer une animation en comptant, même si, rappelons-le, nous vous déconseillons cette méthode pour des questions de performance. Dans notre exemple, l'animation représentant une simple ligne qui tourne sans cesse.



Le code est relativement simple, le compteur d'images constituant l'essentiel du programme. A chaque degré de rotation de la ligne correspond le numéro d'une image dessinée.

```
int compteur;

void setup() {
  size(320, 240);
  frameRate(60);
  fill(0, 0, 0, 10);
  stroke(255);
  smooth();

  compteur = 0;
}

void draw() {
  compteur = compteur + 1;

  rect(0, 0, width, height);

  translate(width / 2, height / 2);
  rotate(radians(compteur));
  line(-height, -height, height, height);
}
```

On déclare un compteur en en-tête de l'application.

```
int compteur;
```

Dans le `setup()` on initialise notre compteur.

```
void setup() {
  size(320, 240);
  frameRate(60);
  fill(0, 0, 0, 10);
  stroke(255);
  smooth();

  compteur = 0;
}
```

A chaque appel de méthode `draw()` on incrémente notre compte de 1.

```
compteur = compteur + 1;
```

On spécifie ensuite le repère de l'espace de dessin pour dessiner au centre et on génère un rotation qui dépend de `compteur`.

```
  translate(width / 2, height / 2);
  rotate(radians(compteur))
```

Une traine est visible dans l'animation. La superposition des lignes en s'effaçant graduellement donne un effet moiré.

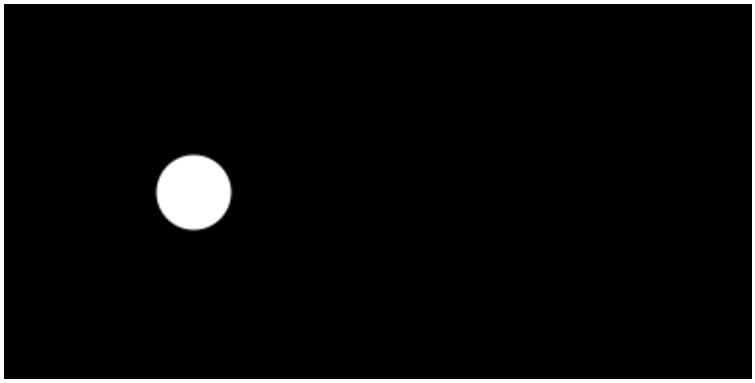
22. L'ANIMATION D'UN OBJET

En combinant animation et objet, il devient possible de concevoir des applications plus ambitieuses. Animer un objet revient à ajouter à son modèle des actions de type : se déplacer, rebondir, tester les collisions, etc. Dans ce chapitre, nous allons apprendre comment animer une balle et la faire rebondir sur les quatre bords de l'écran.

Le résultat final se présentera comme ci-dessous. Afin de mieux visualiser la trajectoire de la balle, nous avons intégré dans l'animation un effet de traine.

CODE DE BASE

Comme point de départ, nous allons reprendre le programme de la balle du chapitre « Les objets » qui nous permettait d'afficher une balle à l'écran et lui ajouter progressivement des morceaux de code. Les nouvelles parties du sketch sont signalées en gras avec la mention //AJOUT ou //DEBUT AJOUT et //FIN AJOUT. Pour vous familiariser avec le fonctionnement de ce nouveau programme, vous pouvez également copier tout le bloc du code initial dans votre fenêtre d'édition de Processing et progressivement lui ajouter la classe ou la méthode concernée.



```
//Déclaration et création d'une instance de l'objet Balle
Balle maBalle = new Balle(100, 100, color(255));

void setup() {
  smooth(); //Lissage des dessins
  size(400, 200); //Taille de la fenêtre
}

void draw() {
  background(0); //On dessine un fond noir
  noStroke(); //On supprime le contour

  maBalle.display(); //Affichage de la balle
}

class Balle {
  //Déclaration des paramètres de base de la balle
  float x;
  float y;
  color couleur;

  //Constructeur de la balle
  Balle (float nouvX, float nouvY, color nouvCouleur) {
    x          = nouvX;
    y          = nouvY;
    couleur    = nouvCouleur;
  }

  //Dessin de la balle
  void display() {
    fill(couleur);
    ellipse(x, y, 40, 40);
  }
}
```

LE DÉPLACEMENT

La balle doit pouvoir se déplacer sur les axes x et y. Nous allons créer deux variables dans le modèle de l'objet qui caractériseront sa vitesse sur les axes x et y. Ensuite nous allons ajouter une nouvelle méthode `bouge()` dans le modèle de l'objet qui sera appelée depuis la méthode `draw()` du programme. Cette méthode va, à chaque fois que l'objet est affiché, modifier la position de la balle par rapport à sa vitesse. Il faudra aussi initialiser les variables décrivant la vitesse dans le constructeur. Pour commencer, nous allons leur donner une valeur fixe.

```
class Balle {
  //Déclaration des caractéristiques de base de la balle
  float x;
  float y;
  float vitesseX; //AJOUT
  float vitesseY; //AJOUT
  color couleur;

  //Constructeur de la balle
  Balle (float nouvX, float nouvY, color nouvCouleur) {
    x      = nouvX;
    y      = nouvY;
    couleur    = nouvCouleur;

    vitesseX = 2; //AJOUT
    vitessesY = 2; //AJOUT
  }

  //Dessin de la balle
  void display() {
    fill(couleur);
    ellipse(x, y, 40, 40);
  }

  //DEBUT AJOUT
  void bouge() {
   x = x + vitesseX;
   y = y + vitesseY;

  //FIN AJOUT
}
```

Il faudra ensuite appeler la méthode `bouge()` depuis la méthode `draw()`. Par ailleurs, nous ajoutons un appel à l'instruction `background()` pour effacer l'écran à chaque nouvelle image.

```
void draw() {
  background(0); //On dessine un fond noir
  noStroke(); //On supprime le contour

  //Déplacement et affichage de la balle
  maBalle.bouge(); //AJOUT
  maBalle.display();
}
```

LES COLLISIONS

Pour le moment, dès que la balle touche le bord de l'écran, elle continue son chemin. Selon l'exemple de la balle qui rebondit sur les coins de l'écran du chapitre « La ligne de temps », nous allons ajouter une méthode `testCollision` qui inversera la vitesse de la balle lorsqu'elle touche les bords de l'écran.

```
class Balle {
  //Déclaration des caractéristiques de base de la balle
  float x;
  float y;
  float vitesseX;
  float vitesseY;
  color couleur;

  //Constructeur de la balle
  Balle (float nouvX, float nouvY, color nouvCouleur) {
    x      = nouvX;
    y      = nouvY;
    couleur    = nouvCouleur;

    vitesseX = 2;
    vitesseY = 2;
  }

  //Dessin de la balle
  void display() {
    fill(couleur);
    ellipse(x, y, 40, 40);
  }

  void move() {
   x = x + vitesseX;
   y = y + vitesseY;
  }

  //DEBUT AJOUT
```

```
  void testCollision() {
    //Si la balle touche un mur, elle rebondit
    if (x > width-20 || x < 20) {
      vitesseX = vitesseX *-1;
    }
    if (y > height-20 || y < 20) {
      vitesseY = vitesseY * -1;
    }
  }
  //FIN AJOUT
}
```

Il faut ensuite appeler la méthode `testCollision()` depuis la méthode `draw()`.

```
//ON REMPLACE L'INSTRUCTION BACKGROUND() PAR CES DEUX LIGNES
  fill(0, 0, 0, 1); // Couleur avec transparence.
  rect(0, 0, width, height);   noStroke();

  //Déplacement et affichage de la balle
  maBalle.bouge();
  maBalle.testCollision();//AJOUT
  maBalle.display();
}
```

CODE FINAL

Voici le code final, une fois toutes ces modifications effectuées.

```
//Déclaration et création d'une instance de l'objet Balle
Balle maBalle = new Balle(100, 100, color(255));

void setup() {
  smooth(); //Lissage des dessins
  size(400, 200); //Taille de la fenêtre
}

void draw() {
  fill(0, 0, 0, 1);
  rect(0, 0, width, height);

  noStroke();

  //Déplacement et affichage de la balle
  maBalle.bouge();
  maBalle.testCollision();
  maBalle.display();
```

```
}

class Balle {
  //Déclaration des paramètres de base de la balle
  float x;
  float y;
  float vitesseX; //AJOUT
  float vitesseY; //AJOUT
  color couleur;

  //Constructeur de la balle
  Balle (float nouvX, float nouvY, color nouvCouleur) {
    x          = nouvX;
    y          = nouvY;
    couleur    = nouvCouleur;

    vitesseX = 2; //AJOUT
    vitesseY = 2; //AJOUT

  }

  //Dessin de la balle
  void display() {
    fill(couleur);
    ellipse(x, y, 40, 40);
  }

  void bouge() {
   x = x + vitesseX;
   y = y + vitesseY;
  }

  void testCollision() {
    //Si la balle touche une mur, elle rebondit
    if (x > width-20 || x < 20) {
      vitesseX = vitesseX * -1;
    }
    if (y > height-20 || y < 20) {
      vitesseY = vitesseY * -1;
    }
  }
}
```

N'hésitez pas à modifier certains paramètres du programme pour vous approprier davantage son fonctionnement.

23. L'ANIMATION DE PLUSIEURS OBJETS

Dès lors qu'un objet est créé dans un programme, il est possible de le multiplier facilement et rapidement. Deux solutions sont possibles :

- Dans le chapitre d'introduction aux objets, nous avons vu qu'on pouvait obtenir deux balles à l'écran en déclarant une seconde copie de la balle et en l'affichant à son tour dans la méthode `draw()`. Ce procédé devient toutefois lourd lorsque le nombre d'objets à reproduire est supérieur à deux.
- Lorsqu'il y a plus de deux objets à animer, il est préférable d'utiliser des listes. Petit rappel du chapitre consacré à cette notion : les listes permettent de gérer facilement un ensemble d'éléments semblables que ce soit des chiffres, des images et même des objets.

Dans ce chapitre, nous allons poursuivre l'exemple de la balle rebondissante. Nous allons ajouter plusieurs balles en utilisant des listes puis ajouter une méthode permettant de gérer les collisions entre les balles.

CODE DE BASE

Comme point de départ, nous allons reprendre le code de la balle du chapitre « Animer un objet » qui nous permettait d'afficher une balle rebondissante à l'écran. Tout au long du chapitre, nous allons ajouter des portions de code à l'exemple de base. Les nouvelles parties sont signalées en gras avec la mention //AJOUT ou //DEBUT AJOUT et //FIN AJOUT. Pour vous familiariser avec le fonctionnement de ce nouveau programme, vous pouvez également copier tout le bloc du code initial dans votre fenêtre d'édition de Processing et progressivement lui ajouter la classe ou la méthode concernée.

```
//Déclaration et création d'une instance de l'objet Balle
Balle maBalle = new Balle(100, 100, color(255));

void setup() {
  smooth(); //Lissage des dessins
  size(400, 200); //Taille de la fenêtre
}

void draw() {
  fill(0, 0, 0, 1);
  rect(0, 0, width, height);

  noStroke();

  //Déplacement et affichage de la balle
  maBalle.bouge();
  maBalle.testCollision();
  maBalle.display();
}

class Balle {
  //Déclaration des paramètres de base de la balle
  float x;
  float y;
  float vitesseX; //AJOUT
  float vitesseY; //AJOUT
  color couleur;

  //Constructeur de la balle
  Balle (float nouvX, float nouvY, color nouvCouleur) {
    x          = nouvX;
    y          = nouvY;
    couleur    = nouvCouleur;

    vitesseX = 2; //AJOUT
    vitesseY = 2; //AJOUT
```

```
  }

  //Dessin de la balle
  void display() {
    fill(couleur);
    ellipse(x, y, 40, 40);
  }

  void bouge() {
   x = x + vitesseX;
   y = y + vitesseY;
  }

  void testCollision() {
    //Si la balle touche une mur, elle rebondit
    if (x > width-20 || x < 20) {
      vitesseX = vitesseX * -1;
    }
    if (y > height-20 || y < 20) {
      vitesseY = vitesseY * -1;
    }
  }
}
```

LISTER LES BALLES

Nous allons maintenant appliquer le concept de listes à notre balle rebondissante. Cela nous permettra d'avoir plusieurs balles à l'écran, sans dupliquer le code !

En premier lieu, nous allons déclarer une liste de balles et non plus une seule balle. Pour cela, nous allons utiliser une variable `nbreBalle` pour stoker le nombre de balles utilisées dans le programme.

Nous allons remplacer la déclaration en en-tête suivant

```
Balle maBalle = new Balle(100, 100, color(255));
```

par

```
//Déclaration d'une variable contenant le nombre de balles
int nbreBalle = 3;

//Déclaration d'une liste d'instances de l'objet Ball
Balle[] balles = new Balle[nbreBalle];
```

Comme dans l'exemple des nombres entiers, nous venons uniquement de déclarer des copies du modèle de balle. Il faut maintenant les créer dans le `setup()`. Nous allons dessiner trois balles au centre de l'écran. Toutes les trois seront blanches.

```
void setup() {
  smooth(); //Lissage des dessins
  size(400, 200); //Taille de la fenêtre

  //DEBUT AJOUT
  //Cette boucle va créée trois balles
  //blanches au centre de l'écran
  for (int i = 0; i < nbreBalle; i++) {
    balles[i] = new Balle(width/2, height/2,  color(255));
  }
  //FIN AJOUT
}
```

Dans la méthode `draw()` , nous allons aussi créer une boucle qui va parcourir tous les éléments de la liste pour les déplacer, tester leurs collisions et les afficher. Nous allons remplacer

```
  //Déplacement et affichage de la balle
  maBalle.bouge();
  maBalle.testCollision();
  maBalle.display();
```

par

```
  //Cette boucle va déplacer et afficher les trois balles
  for (int i = 0; i < nbreBalle; i++) {
    balles[i].bouge();
    balles[i].testCollision();
    balles[i].display();
  }
```

Une dernière opération va consister à modifier le constructeur du modèle de la balle afin que chaque balle ait une vitesse et une direction spécifiques. Pour ce faire nous allons utiliser la fonction `random()` qui permet de générer des nombres aléatoires. Nous allons remplacer le constructeur ci-dessous :

```
  //Constructeur de la balle
  Balle (float nouvX, float nouvY, color nouvCouleur) {
    x      = nouvX;
    y      = nouvY;
    couleur    = nouvCouleur;

    vitesseX = 2;
```

```
    vitesseY = 2;
}
```

par celui-ci

```
//Constructeur de la balle
  Balle (float nouvX, float nouvY, color nouvCouleur) {
    x      = nouvX;
    y      = nouvY;
    couleur    = nouvCouleur;

    vitesseX = 2 + random(-1,1);
    vitessesY = 2 + random(-1,1);
}
```

CODE FINAL

Voici le programme complet :

```
//Déclaration d'une variable contenant le nombre de balles
int nbreBalle = 3;

//Déclaration d'une liste d'instances de l'objet Balle
Balle[] balles = new Balle[nbreBalle];

void setup() {
  smooth(); //Lissage des dessins
  size(400, 200); //Taille de la fenêtre

  //Cette boucle va créer trois balles blanches
  //au centre de l'écran
  for (int i = 0; i < nbreBalle; i++) {
    balles[i] = new Balle(width/2, height/2,  color(255));
  }
}

void draw() {
  fill(0, 0, 0, 1); // Couleur avec transparence.
  rect(0, 0, width, height);

  noStroke();

  //Cette boucle va déplacer et afficher les trois balles
  for (int i = 0; i < nbreBalle; i++) {
    balles[i].bouge();
    balles[i].testCollision();
```

```
    balles[i].display();
  }
}

class Balle {
  //Déclaration des paramètres de base de la balle
  float x;
  float y;
  float vitesseX;
  float vitesseY;
  color couleur;

  //Constructeur de la balle
  Balle (float nouvX, float nouvY, color nouvCouleur) {
    x      = nouvX;
    y      = nouvY;
    couleur    = nouvCouleur;

    vitesseX = 2 + random(-1,1);
    vitesseY = 2 + random(-1,1);
  }

  //Dessin de la balle
  void display() {
    fill(couleur);
    ellipse(x, y, 40, 40);
  }

  //Déplacement de la balle
  void bouge() {
    x = x + vitesseX;
    y = y + vitesseY;
  }

  void testCollision() {
    //Si la balle touche un mur, elle rebondit
    if (x > width-20 || x < 20) {
       vitesseX = vitesseX * -1;
    }
    if (y > height-20 || y < 20) {
       vitesseY = vitesseY * -1;
    }
  }
}
```

N'hésitez pas à modifier certains paramètres du programme pour vous approprier davantage son fonctionnement.

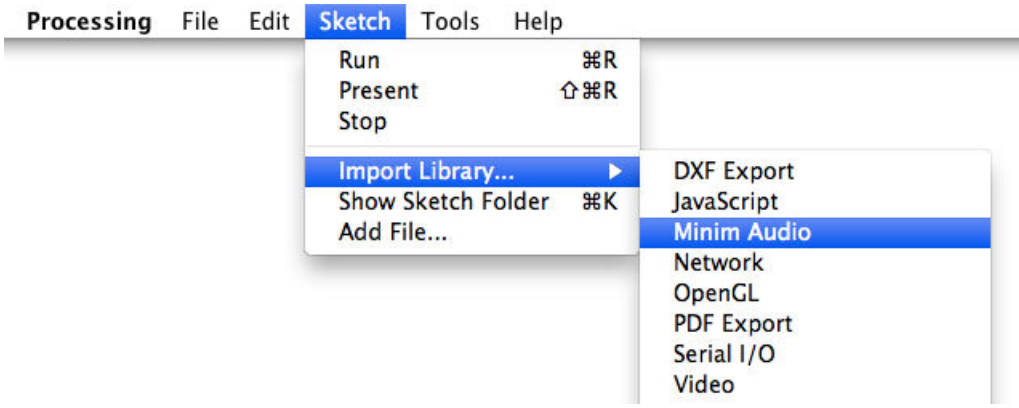
24. LA LECTURE DU SON

Processing est un environnement principalement dédié à la création visuelle. Il n'a pas été conçu au départ pour jouer du son, et encore moins pour générer directement des ondes audio, tel un logiciel comme Pure Data. Par la suite, de nombreuses bibliothèques externes ont été créées pour étendre les capacités sonores de Processing. Ainsi vous pouvez à n'importe quel moment intégrer ces bibliothèques externes dans vos programmes : des synthétiseurs, des sampleurs audio, des interfaces MIDI, des séquenceurs, etc. Sachez toutefois que ce n'est pas la mission de base de Processing davantage centrée sur la génération de formes visuelles.

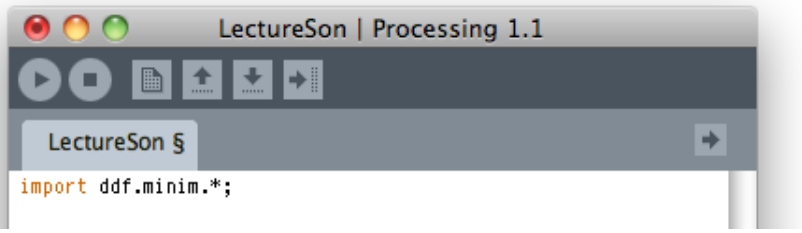
Dans ce chapitre, nous allons nous familiariser avec un *minim* de fonctionnalités audio disponibles par défaut dans Processing.

MINIM

Il a été décidé à un moment donné d'intégrer dans toute distribution Processing la bibliothèque audio Minim pour pouvoir par défaut jouer des fichiers audio ou capter le son entrant par le microphone. A priori cette bibliothèque est déjà installée sur votre machine. Pour vérifier sa présence et l'intégrer dans votre programme, il suffit d'aller dans le menu *Sketch › Import Library... › Minim audio*.



Selon votre plate-forme et votre version, à la suite de cette action, Processing ajoutera des lignes de code plus ou moins nombreuses en haut de votre programme. Par défaut, il faut au moins voir affichée l'instruction suivante :




```
import ddf.minim.*;
```

C'est cette instruction (que vous pouvez saisir à la main si vous le voulez) qui importera l'ensemble des fonctionnalités de la bibliothèque Minim pour les rendre accessibles à notre programme. C'est à partir de cette instruction `import ddf.minim.*;` que notre sketch va pouvoir jouer du son.

CONFIGURER MINIM

Le lancement de notre sketch via le bouton **Run** ne suffit pas pour jouer un son. Au préalable, il faut prévoir d'activer les fonctionnalités de gestion audio de Minim (son moteur audio en jargon informatique). Pour jouer un son, Minim doit en effet auparavant demander un accès à la carte son de votre ordinateur. Ne vous inquiétez pas, Minim s'occupe de toute cette cuisine interne pour vous, il suffit de lui dire de s'activer pour cela. Le fait d'avoir activé Minim suppose de prévoir de le désactiver à la fin de notre programme.

Si vous ne comprenez pas tout ce que nous venons de dire, ce n'est pas trop grave. Sachez simplement qu'il faut saisir le code suivant au début de chaque programme utilisant Minim :

```
import ddf.minim.*;

Minim minim;

void setup() {
  minim = new Minim(this);
}
```

Et qu'il faut également écrire la fin du programme les instructions suivantes :

```
void stop() {
  minim.stop();
  super.stop();
}
```

Si par curiosité vous voulez savoir à quoi sert le mot « `this` » dans ce programme, sachez qu'il sert à indiquer à Processing que nos instructions s'appliquent à « ce »

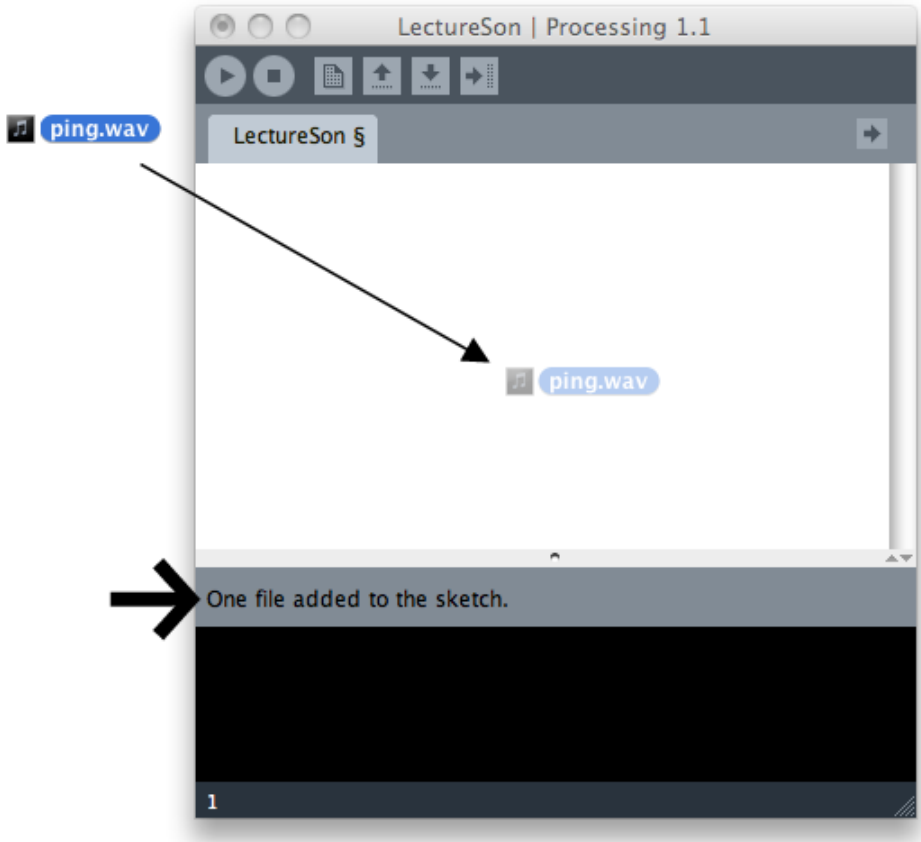
programme. Minim a besoin de connaître l'adresse de « ce » programme (le nôtre) pour le faire communiquer avec la carte son de notre ordinateur. Encore une fois, si vous ne comprenez pas ce que nous venons de dire, ce n'est pas grave. Sachez seulement que ces mots doivent être écrits au début et à la fin de chaque programme lorsque nous voulons produire du son.

IMPORTER UN SON

Tout comme les images photographiques ou la typographie, nous allons importer un fichier son dans Processing pour pouvoir le jouer dans notre programme. Pour bien réussir cette étape, nous vous recommandons de sauvegarder d'abord votre sketch.

Nos sons peuvent venir de plusieurs sources : des sons téléchargés, des sons que vous avez créés vous-même via un logiciel comme Audacity, voire des sons que vous pouvez trouver par défaut dans votre système d'exploitation.

Identifiez et sélectionnez à présent le fichier son que vous souhaitez utiliser, et glissez-le directement sur la fenêtre Processing :



Cette action placera le son de ce fichier dans le dossier « data » de votre sketch (un message s'affichant au niveau de la console de Processing vous le confirmera). Si vous voulez voir ce que contient ce dossier, taper sur ctrl-k (Windows/Linux) ou cmd-k (Mac), sinon choisissez dans le menu *Sketch › Show Sketch Folder*.

FORMATS AUDIO

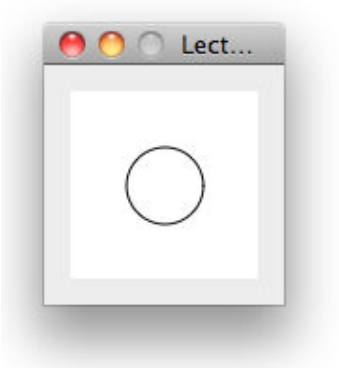
Vous pouvez utiliser trois formats audio avec Minim : *WAV*, *AIFF*, et *MP3*. Les deux premiers formats peuvent être exportés par défaut via des logiciels libres comme Audacity. Le troisième format nécessite à ce jour un plug-in pour Audacity, mais celui-ci est relativement facile à installer.

Comme pour les images, les différents formats ont leurs avantages et désavantages.

1. Les formats *WAV* et *AIFF* sont quasi identiques et sont deux formats *non compressés*. Ils peuvent être lus sans trop de difficultés par Minim, car ils ne requièrent pas de *décompression*. Ils sont souvent utilisés pour des sons courts, de type bruitage. Pour un morceau de musique de même durée, ces deux formats occupent beaucoup plus de place mémoire que les fichiers au format *MP3*.
2. Le format *MP3* est un format compressé. Il est souvent beaucoup moins volumineux que les fichiers *WAV* et *AIFF*. Par contre, la lecture du format *MP3* nécessite plus de ressources de calcul de votre ordinateur, car le fichier doit être décompressé lors de sa lecture.

UNE HORLOGE

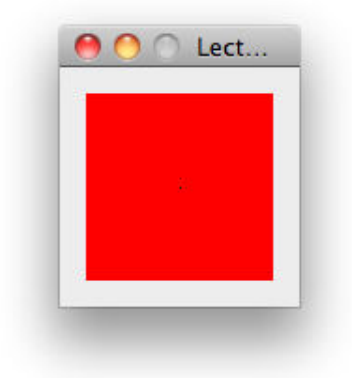
Voici un programme simple, qui anime une ellipse avec les secondes de l'horloge de votre ordinateur (l'horloge système). Nous allons utiliser ce programme comme base pour créer une horloge qui sonne toutes les minutes. Pour mieux anticiper cet événement qui se déroule toutes les soixante secondes — car il nécessite une certaine patience — nous allons représenter les secondes par une ellipse qui s'agrandit progressivement depuis le centre de l'espace de dessin.



```
void draw() {
  background(255);
  ellipse(50,50,second(),second());
}
```

Nous utiliserons la valeur des secondes de l'horloge système pour définir les paramètres {*largeur,hauteur*} de l'ellipse. Si les secondes sont égales à 42, nous aurons une ellipse de 42 x 42 pixels. Si les secondes sont égales à 2, nous aurons une ellipse de 2 x 2 pixels.

Nous pouvons également ajouter une condition pour signaler davantage le changement des minutes : à chaque fois que les secondes reviendront à zéro, nous marquerons l'événement avec un écran rouge.




```
void draw() {

  if (second() == 0) {
    background(255,0,0);
  } else {
    background(255);
  }

  ellipse(50,50,second(),second());
}
```

La condition `if(second() == 0) {}` compare les secondes avec la valeur 0. Si les deux sont égales, le fond sera rouge, sinon (`else`) le fond sera blanc.

UNE HORLOGE SONORE

Nous allons maintenant sonoriser ce programme en jouant un son à chaque fois que les secondes de notre horloge reviennent à zéro.

Pour commencer, nous allons ajouter autour de notre méthode `draw()` de l'horloge, toutes les méthodes nécessaires pour la sonoriser (lignes de code mentionnées en gras).

```
import ddf.minim.*;

Minim minim;
AudioSnippet ping;

void setup() {
  minim = new Minim(this);
  ping = minim.loadSnippet("ping.wav");
}

void draw() {

  if (second() == 0) {

    if (ping.isPlaying() == false){
      ping.play();
    }

    background(255,0,0);

  } else {
    background(255);
  }

  ellipse(50,50,second(),second());
}

void stop() {
  ping.close();
  minim.stop();
  super.stop();
}
```

Attardons-nous sur la structuration de ce sketch :

LectureSon | Processing 1.1

LectureSon §

```
import ddf.minim.*;

Minim minim;
AudioSnippet ping;

void setup() {
  minim = new Minim(this);
  ping = minim.loadSnippet("ping.wav");
}

void draw() {

  if (second() == 0) {

    if (ping.isPlaying() == false){
      ping.play();
    }
    background(255,0,0);

  } else {
    background(255);
  }

  ellipse(50,50,second(),second());
}

void stop() {
  ping.close();
  minim.stop();
  super.stop();
}
```

Ce programme nécessite de nombreuses étapes pour fonctionner :

1. Importer les fonctionnalités Minim dans notre sketch.
2. Créer deux variables dont l'une qui contiendra le moteur Minim et l'autre les données audio.
3. Démarrer le moteur Minim au début du programme.

4. Copier les données du fichier audio dans notre variable.
5. Actions conditionnelles : jouer le son et afficher un écran rouge lorsque les secondes sont égales à zéro (vérifier que ce son n'est pas déjà en train de jouer) .
6. Dessiner une ellipse en fonction de la valeur des secondes de l'horloge.
7. A la fin du programme, arrêter le son.
8. A la fin du programme, arrêter Minim.

Nous avons déjà expliqué le démarrage et l'extinction du moteur Minim. Nous n'y revenons pas. Par contre, nous attirons votre attention sur les parties 2, 4 et 5 du programme qui concernent l'importation et la lecture du son.

Tout d'abord, nous avons créé une variable de type *AudioSnippet*. Le type *AudioSnippet* est un type de variable défini à l'intérieur du code de Minim. Nous vous rappelons que dans le cas de n'importe quelle variable, la formule pour la déclarer est de la forme {type de la variable} {nom de la variable} = {les valeurs de la variable}. Par exemple, si par un tour de force il était possible d'importer un petit oiseau tout mignon dans Processing, il suffirait d'écrire `PetitOiseau pioupiou = loadBird("pioupiou.bird");`. En d'autres termes, on écrit d'abord le type de la chose, le nom de la chose, et enfin on lui donne sa valeur. Pour revenir à notre exemple d'horloge sonore, cette valeur est donnée par la fonction `loadSnippet()` qui va aller chercher les valeurs d'ondes dans le fichier audio et les importera dans notre variable nommée « ping ».

Pour plus d'informations sur les variables et les différents types de variables, reportez-vous au chapitre dédié à ce sujet.

Une fois la variable chargée avec le son, il suffit de le jouer. Dans le cas du *AudioSnippet* la commande pour jouer un son s'appelle `play()`. Enfin, il peut être important de vérifier à un moment donné si le son n'est pas déjà en train d'être joué. C'est le rôle de la commande `isPlaying()` de nous dire si oui (`true`) ou non (`false`) nous sommes dans cette situation. Dans notre exemple, si nous avions oublié cette condition, le son se serait lancé pendant soixante fois durant la totalité de la seconde où `second()` est égal à zéro.

AUDIOSNIPPET, AUDIOSAMPLE, AUDIOPLAYER

La bibliothèque Minim distingue plusieurs manières de jouer un son dans Processing. En principe, les instructions à utiliser dépendront de la nature et rôle du son dans votre programme. Si l'on se réfère à la documentation décrivant les fonctionnalités offertes par Minim, un `AudioSnippet` sera employé pour une lecture simple d'un court extrait sonore, un `AudioSample` pour une lecture rapide et répétée comme une boîte à rythmes, et enfin un `AudioPlayer` pour jouer des fichiers audio plus longs (souvent de type *MP3*) situés directement sur le disque dur de votre ordinateur pour ne pas alourdir encore davantage la mémoire interne du programme. Voici un exemple qui utilise les trois méthodes de lecture du son associées respectivement à trois fichiers audio imaginaires dénommés ping, pop et song.

```
import ddf.minim.*;

Minim minim;
AudioSample ping;
AudioSnippet pop;
AudioPlayer song;

void setup() {
  minim = new Minim(this);
  ping = minim.loadSample("ping.aiff");
  pop = minim.loadSnippet("pop.aiff");
  song = minim.loadFile("song.mp3");
}

void draw() {

}

void keyPressed() {
  if (key == 'a') ping.trigger();
  if (key == 'b') {
    pop.rewind();
    pop.play();
  }
  if (key == 'c') song.play();
}

void stop() {
  ping.close();
  pop.close();
  song.close();
  minim.stop();
  super.stop();
}
```

Notez avant tout que les instructions d'importation et de lectures changent selon le type choisi. `AudioPlayer` et `AudioSample` utiliseront tous les deux la méthode `play()` car leur lecture est toujours unique (un seul son à la fois). Par contre, `AudioSample` nécessite la lecture de nombreux sons simultanés (on peut taper plusieurs fois sur une batterie) et pour cette raison utilise une autre nomenclature, `trigger()`, pour décrire le lancement d'un son. En anglais, le mot « trigger » signifie une gâchette.

DOCUMENTATION

Pour plus d'informations sur les différentes fonctionnalités de Minim, ainsi qu'un grand nombre d'exemples d'utilisation (avec leur code source librement mis à disposition), reportez-vous au site http://code.compartmental.net/tools/minim/ ou à la documentation technique (plus difficile à lire) consultable à l'adresse http://code.compartmental.net/minim/javadoc/ddf/minim/package-tree.html.

SORTIR

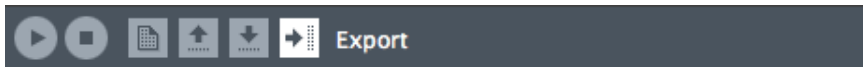
25. L'exportation
26. L'impression
27. La vidéo

25. L'EXPORTATION

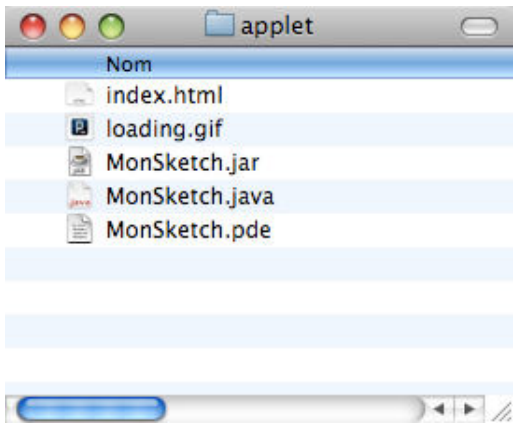
Pour l'instant, nous avons travaillé exclusivement dans l'environnement de Processing. Nous tapons des lignes de code et nous les exécutons depuis l'interface par le bouton **Run**. A présent, nous allons voir que Processing intègre deux manières d'exporter notre programme pour le distribuer. L'une, à destination du web, permet d'exposer son animation en ligne, visible sur une page dans un navigateur. L'autre procédé de diffusion permet de créer une application autonome fonctionnant directement sur un ordinateur utilisant Windows, Mac ou Linux. Dans les deux cas, plus besoin de l'éditeur pour faire tourner votre sketch.

EXPORTER POUR LE WEB

Une fois que votre programme fonctionne et que vous êtes satisfait du résultat, vous pouvez générer un *applet*, objet intégrable à une page web et lisible par votre navigateur internet habituel. Pour effectuer cette exportation, il suffit de cliquer sur le bouton **Export**.



L'opération effectuée, Processing ouvre automatiquement un dossier *applet* dans l'explorateur de fichiers de votre ordinateur. Celui-ci contient plusieurs fichiers.



- *index.html* est la page web générée par Processing. En double cliquant dessus, votre navigateur internet va se lancer et afficher l'animation que vous venez juste d'exporter.

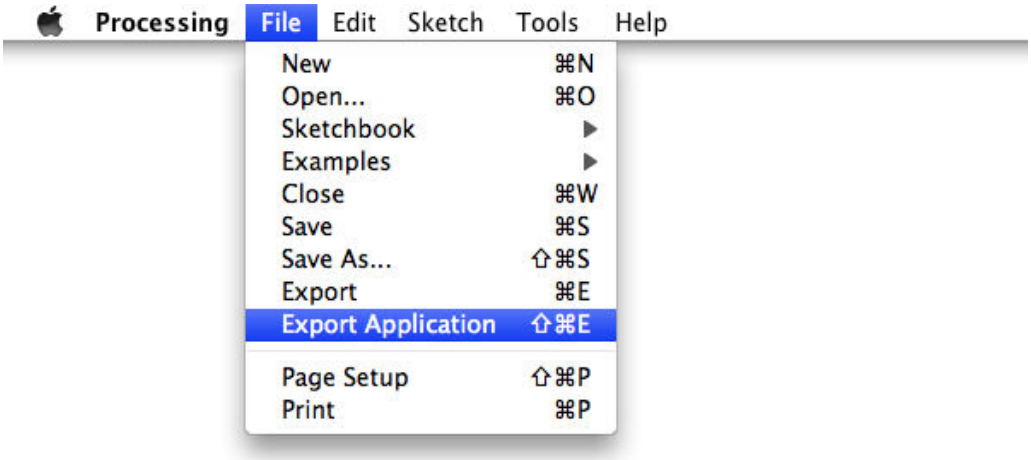
- *loading.gif* est une icône utilisée lors du chargement de l'applet dans la page web.
- *{votre nom de sketch}.jar* est l'*applet* à proprement parler. Il s'agit du code source de votre sketch qui a été transformé et sauvegardé avec toutes les données.
- *{votre nom de sketch}.java* est la version Java de votre code.
- *{votre nom de sketch}.pde* est le code source original du sketch que vous avez saisi dans l'éditeur de Processing.

Attention! A chaque exportation, Processing écrase automatiquement les fichiers portant le même nom. Par précaution, si vous effectuez des modifications sur le fichier *{votre nom de sktech}.html*, pensez au préalable à le renommer.

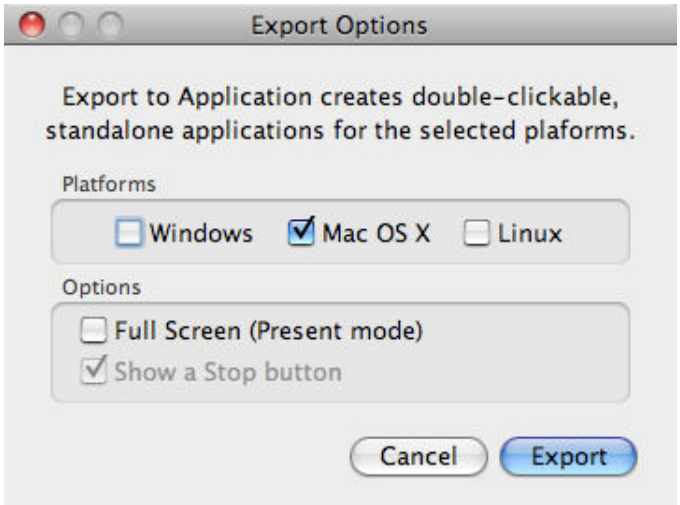
Si vous disposez d'un espace de stockage sur un serveur web, et que vous êtes familiers des transferts FTP, vous pouvez téléverser le dossier *applet* (en le renommant, si besoin est) dans votre espace en ligne. Vous pourrez ainsi rendre accessibles vos sketchs sur Internet.

EXPORTER POUR L'ORDINATEUR

Votre sketch Processing peut également être exporté sous la forme d'une application directement exécutable sur Windows, Mac et GNU/Linux. Il suffira alors de cliquer dessus pour le lancer comme n'importe quel autre logiciel présent sur votre ordinateur. Pour effectuer cette exportation, il suffit de sélectionner dans le menu **File › Export Application**.

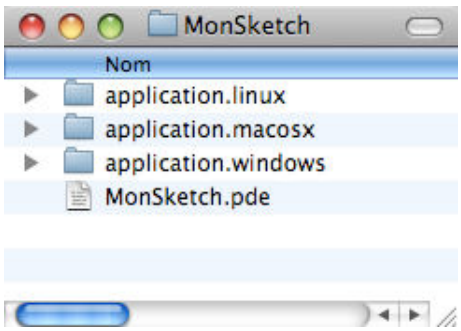


La fenêtre *Export Options* suivante devrait s'ouvrir.



- *Platforms* permet de sélectionner la plateforme. C'est là toute la beauté de Processing : nous pouvons exporter une application développée sur Mac pour GNU/Linux, par exemple.
- *Full Screen (Present mode)* lancera l'application en plein écran lors du double-clic.
- *Show a Stop button* , en mode *Full Screen*, affiche un bouton **Stop** sur l'écran pour arrêter l'application.

Une fois l'application assemblée, la fenêtre du dossier du sketch s'affiche avec les dossiers propres à chaque plateforme. Le fichier exécutable se trouve à l'intérieur, prêt à être lancé sur votre ordinateur.



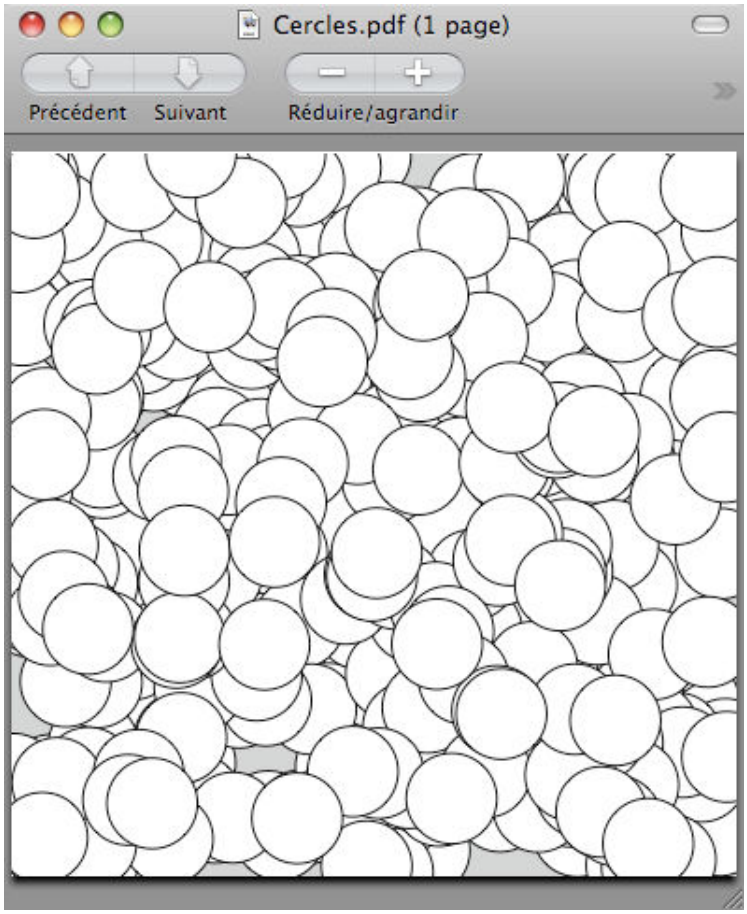
Comme vous pouvez le constater, les procédures d'exportation sont vraiment très simples à réaliser.

26. L'IMPRESSION

Pour l'instant, nous avons travaillé sur l'écran en générant des images et des animations. Nous allons maintenant nous intéresser aux possibilités qu'offre Processing en matière d'impression : nous allons créer un document `PDF` (*Portable Document Format*) contenant des formes géométriques vectorielles, document qui pourra ensuite être imprimé sur du papier ou un autre médium.

MODE DIRECT

Dans ce premier exemple, nous allons générer 500 cercles aléatoirement et les dessiner dans un document PDF, qui sera sauvé dans notre dossier de travail.

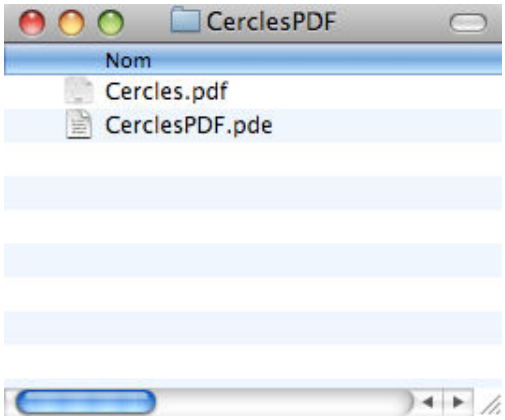


```
import processing.pdf.*;
size(400, 400, PDF, "Cercles.pdf");
for (int i = 0; i < 500; i++)
   ellipse(random(width),random(height), 50, 50);
exit();
```

Comment fonctionne ce programme ? Tout d'abord, nous commençons par importer toutes les classes de la librairie `processing.pdf`. Nous utilisons la méthode `size()` qui permet de définir les caractéristiques de notre espace de dessin : cette fois-ci, nous mentionnons quatre paramètres. Notez le mot-clé `PDF` qui indique à Processing que nous allons dessiner dans le document `"Cercles.pdf"`, passé en quatrième paramètre de la méthode.

Avec ce mode direct de tracé, puisque nous dessinons directement dans le document et non plus sur l'écran, nous n'avons plus besoin de la fenêtre de visualisation de Processing. Le programme se termine par la commande `exit()`, qui indique au logiciel de terminer le programme tout seul, exactement comme si nous avions appuyé sur le bouton stop de l'interface.

Le document généré au format pdf se trouve dans le dossier de sketch de votre application. Vous pouvez double-cliquer dessus pour l'ouvrir.



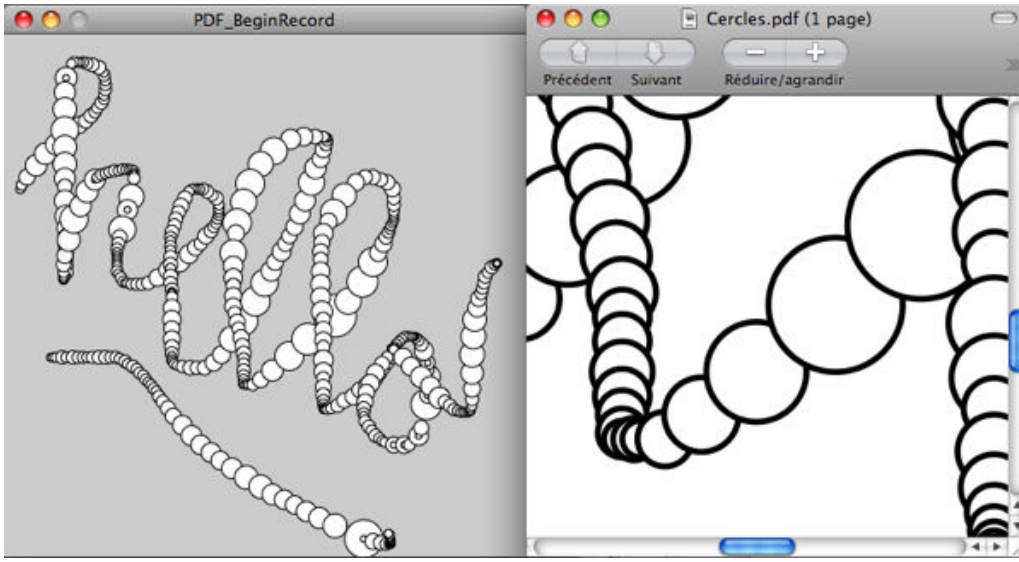
MODE ENREGISTREMENT

Nous allons voir maintenant comment dessiner à la fois sur l'écran et dans un fichier .pdf à l'aide des commandes `beginRecord()` et `endRecord()`. Ces deux méthodes vont nous permettre de démarrer et arrêter l'enregistrement des commandes graphiques dans un fichier PDF.

Comme précédemment, le fichier généré est sauvegardé dans le dossier du sketch.

Dans l'exemple suivant, nous allons réaliser un outil graphique, une sorte de pinceau numérique dont la taille du tracé dépend de la vitesse à laquelle nous déplaçons la

souris. Au démarrage de l'application, dans le `setup()`, nous indiquons le début de l'enregistrement par la commande `beginRecord()` qui prend en paramètres le type d'export (pour l'instant Processing ne supporte que l'extension `PDF`) et le nom du fichier en second paramètre. Lorsqu'on appuie sur la touche `ESPACE`, l'enregistrement est stoppé et l'application s'arrête. Comme précédemment, le fichier généré est sauvegardé dans le dossier du sketch.




```
import processing.pdf.*;
boolean drag = false;

void setup()
{
  size(400, 400);
  smooth();
  beginRecord(PDF, "Cercles.pdf");
}

void draw() {
  if (drag)
  {
    float r = dist(pmouseX, pmouseY, mouseX, mouseY) + 5;
    ellipse(mouseX, mouseY, r, r);
  }
}

void mouseDragged() {
  drag = true;
}
```

```
void mouseReleased() {
  drag = false;
}

void keyPressed() {
    endRecord();
    exit();
}
```

La ligne suivante :

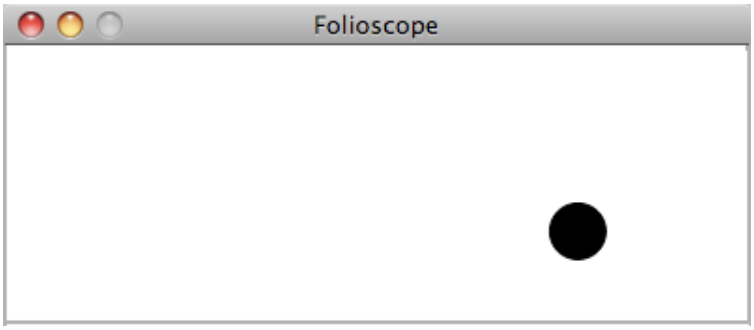
```
float r = dist(pmouseX, pmouseY, mouseX, mouseY) + 5;
```

... permet de calculer la vitesse de la souris grâce à la méthode `dist` et aux variables `pmouseX`, `pmouseY` et `mouseX` et `mouseY`. Nous ajoutons `5` pour ne pas avoir des cercles de rayon `0` sur l'écran (ce qui reviendrait à avoir des cercles invisibles).

La variable `drag` sert à savoir si l'utilisateur est en train de déplacer la souris en maintenant le clic appuyé, ce qui active à l'aide de `draw()` l'action de dessiner des cercles sous le pointeur de la souris.

UN GÉNÉRATEUR DE LIVRES

Au lieu de générer un document d'une seule page, nous allons à présent réaliser un livre de plusieurs pages. Nous allons créer dynamiquement un folioscope (http://fr.wikipedia.org/wiki/Folioscope) dont chaque page représentera une image de l'animation à un instant donné. C'est la méthode `nextPage()` qui va créer une page vierge à chaque fois que cette instruction sera appelée durant l'enregistrement.




```
import processing.pdf.*;

PGraphicsPDF pdf;
float x = 0;
float y = 75;

void setup() {
```

```
  size(400, 150);
  smooth();
  pdf = (PGraphicsPDF) beginRecord(PDF, "Bulle.pdf");
}

void draw() {
  background(255);
  fill(0);
  ellipse(x, y, 30, 30);

  x = x + random(5);
  y = y + random(-5, 5);

  if (x - 30 > width) {
    endRecord();
    exit();
  }
  pdf.nextPage();
}
```

Nous avons introduit la variable `pdf` pour stocker une référence au document que nous générons. Lorsque nous avons fini de dessiner une image dans `draw()`, nous passons à l'image suivante par la commande `pdf.nextPage()`. Le programme est conçu pour s'arrêter lorsque la bulle sort de l'écran. Après l'exécution du programme, le fichier sauvegardé du folioscope se trouve dans le dossier du sketch et porte le nom `Bulle.pdf`. En l'ouvrant, vous pourrez constater qu'il comporte 164 pages, chacune d'entre elles présentant notre balle en un instant donné de son mouvement dans l'espace de dessin.

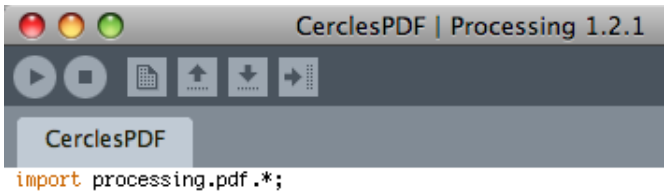
PIXEL VERSUS VECTORIEL

Lorsque nous dessinons sur l'écran, un rectangle par exemple, nous remplissons les pixels d'une zone de l'écran avec une couleur particulière. Lorsque nous sauvegardons cette image dans un fichier, la première méthode consiste à utiliser des formats d'image particuliers (`JPEG`, `GIF`, `TIFF` par exemple) qui enregistrent individuellement chaque pixel qui compose l'illustration. Avec ce type de format, si nous voulons agrandir l'image, l'ordinateur sera dans l'impossibilité de créer des pixels intermédiaires et d'ajouter du détail, car il ne possède aucune information dans le fichier lui permettant d'extrapoler les modifications liées à cet agrandissement de formes.

En enregistrant la même image au format vectoriel, Processing ne va plus sauver chaque pixel un à un, mais plutôt les caractéristiques de chaque forme dessinée : sa forme, sa position et sa couleur. Ainsi, lorsque nous agrandissons l'image, l'ordinateur est capable de dessiner avec précision les détails d'une image qu'il n'aurait pas été possible de produire avec une image pixelisée. Le pdf est un format de sauvegarde vectoriel.

IMPORTER LES FONCTIONNALITÉS PDF

Dans les exemples précédents, nous avons employé `import processing.pdf.*` qui permet d'importer la librairie relative à la création de document PDF. Une librairie est un ensemble de fonctionnalités qui étend les possibilités de Processing et chaque fois que nous aurons à utiliser l'export PDF dans notre sketch, nous utiliserons cette commande d'importation.



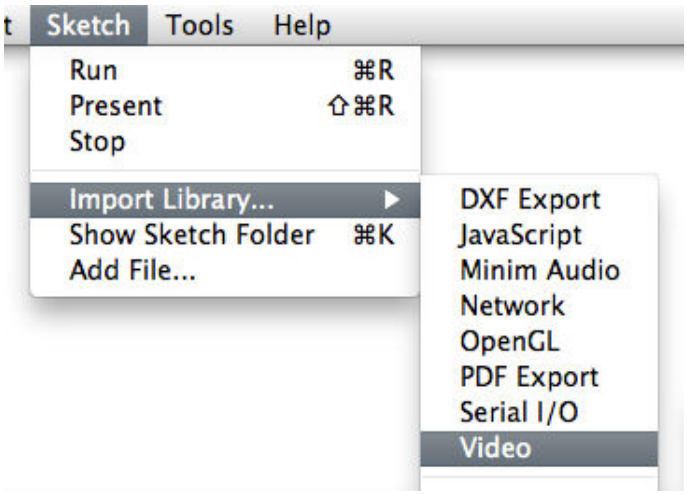
Si jamais vous essayez d'utiliser les méthodes relatives à l'export PDF sans avoir importé la librairie, vous allez rencontrer des erreurs qui s'afficheront dans la console et le programme ne pourra démarrer. Au lieu de taper cette ligne de code, vous pouvez l'ajouter automatiquement en cliquant dans le menu **Sketch › Import Library... › PDF Export**.

27. LA VIDÉO

Processing vous offre la possibilité d'exporter votre sketch animé ou interactif sous forme d'animation vidéo au format QuickTime. Pour créer des vidéos, nous allons utiliser un objet prédéfini `MovieMaker` de la librairie `org.processing.video`. Attention, pour les utilisateurs sous Linux, cette fonction n'est à ce jour pas disponible.

L'OBJET MOVIEMAKER

L'objet `MovieMaker` est une classe prédéfinie de Processing qui permet de créer un fichier vidéo, et de lui ajouter des images au fur et à mesure de l'exécution de votre programme. Pour pouvoir l'utiliser dans votre sketch, il faut importer la librairie vidéo de Processing :



Il faut ensuite déclarer une variable en en-tête de notre programme qui stockera notre objet vidéo. Dans le `setup()` du sketch, il faut créer l'objet en lui donnant un certain nombre de paramètres :

- Une référence au sketch
- La largeur de la vidéo
- La hauteur de la vidéo
- Le nom du fichier de sortie

```
import processing.video.*;

MovieMaker mm; //Déclaration de la variable qui recevra notre objet

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

  /*
   * Création de l'objet MovieMaker. La vidéo aura
   * la même taille que notre esquisse
   */
  mm = new MovieMaker(this, width, height, "Esquisse.mov");
}
```

AJOUTER DES IMAGES

A chaque appel de la méthode `draw()` de Processing, il faut indiquer à l'objet `MovieMaker` qu'il doit enregistrer l'image actuellement à l'écran dans le fichier vidéo.

```
void draw() {
  ellipse(mouseX,mouseY,20,20); //On dessine une ellipse

  mm.addFrame(); //On ajoute une image à la vidéo
}
```

FINALISER LA VIDÉO

A ce stade, l'enregistrement de la vidéo se termine lorsque l'on quitte le programme. On peut indiquer à l'objet `MovieMaker` d'arrêter l'enregistrement à tout moment. Dans notre exemple, nous allons ajouter une action qui stoppe l'enregistrement vidéo lorsqu'on appuie sur n'importe quelle touche du clavier de votre ordinateur.

```
void keyPressed() {
  mm.finish();
}
```

PERSONNALISER L'ENREGISTREMENT

Par défaut, l'objet `MovieMaker` enregistre une vidéo non compressée à 15 images par seconde. Pour des applications interactives nécessitant un temps de réaction très rapide de l'ordinateur, cette qualité d'enregistrement peut s'avérer trop lourde. A l'inverse, pour d'autres types de création, on peut avoir besoin de produire des vidéos encore plus fluides (comportant davantage d'images par seconde). Avec l'objet `MovieMaker`, il est possible de personnaliser les caractéristiques de l'enregistrement en précisant notamment un format particulier ainsi que le nombre d'images par seconde.

Les paramètres sont les suivants :

- Une référence au sketch
- La largeur de la vidéo
- La hauteur de la vidéo
- Le nom du fichier de sortie
- Nombre d'images par seconde
- Codec de compression
- Qualité

```
mm = new MovieMaker(this, width, height, "sketch.mov", 30,
MovieMaker.H263, MovieMaker.HIGH);
```

(Pour des raisons de lisibilité, le code de la ligne précédente a été réparti sur deux lignes. Si vous devez le copier pour le réutiliser dans un sketch, supprimer le retour chariot pour placer l'ensemble des instructions sur une seule ligne).

Avec ces paramètres, l'enregistrement de la vidéo est beaucoup plus fluide.

Voici quelques autres codecs vidéo intéressants :

- `MovieMaker.H264` : Lent à encoder et décoder. Occupe par contre très peu d'espace-disque sur votre ordinateur.
- `MovieMaker.MOTION_JPEG_B` : Très rapide à encoder et décoder. Occupe également peu d'espace-disque.
- `MovieMaker.ANIMATION` : Supporte le canal alpha permettant la gestion de la transparence, très utile pour certaines images. Occupe énormément d'espace-disque, la vidéo étant compressée sans perte.

INTERAGIR

28. Les événements clavier
29. Les événements souris
30. L'entrée microphone
31. L'entrée vidéo

28. LES ÉVÉNEMENTS CLAVIER

Capter les actions de l'utilisateur sur le clavier de votre ordinateur (les évènements clavier) et s'en servir pour faire réagir votre programme constitue une première forme d'interactivité. Nous allons découvrir dans ce chapitre comment récupérer les informations liées au clavier en fabriquant une machine à écrire très simple qui nous permettra d'imprimer du texte sur l'écran.

ACTIONS

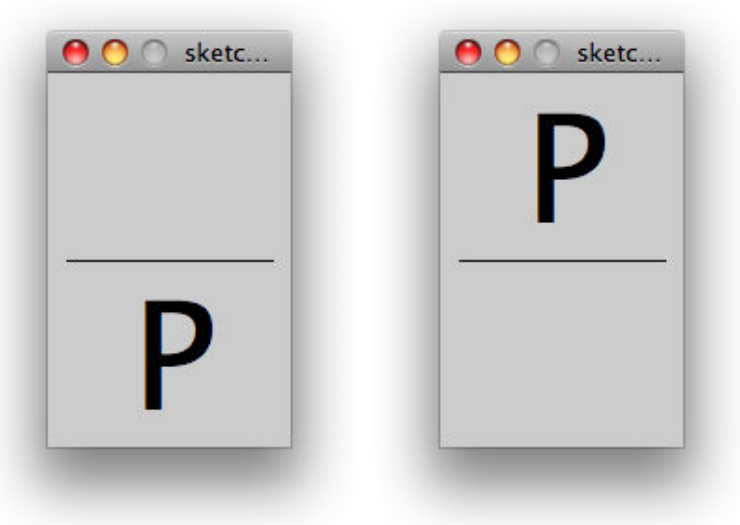
LA VALEUR DE LA DERNIÈRE TOUCHE APPUYÉE EST STOCKÉE DANS LA VARIABLE `key` proposée par défaut dans Processing. Cette variable ne peut stocker qu'un seul caractère à la fois, symbolisé par l'écriture `'a'` , `'b'` , `'c'`, ... Attention, `key` est sensible à la casse, c'est-à-dire qu'il fait une différence entre les minuscules et majuscules.

Dans l'exemple suivant, nous allons afficher dans la fenêtre de visualisation le caractère correspondant à la touche qui aura été appuyée sur le clavier de votre ordinateur. Pour ce faire, nous employons la méthode `text()`.

```
 void draw() {
  background(204);
  fill(0);
  textSize(70);
  text(key,30,70);
}
```

Par ailleurs, nous pouvons capter dans notre programme le moment où l'utilisateur a appuyé ou relâché une touche du clavier, par le biais des méthodes `keyPressed()` et `keyReleased()`. Ces deux méthodes vont être automatiquement appelées par Processing au moment où l'état d'une touche change.

Dans l'exemple suivant, `keyPressed()` et `keyReleased()` changent la valeur de la variable `y`, utilisée pour positionner le caractère à l'écran selon que la touche est appuyée ou relâchée.



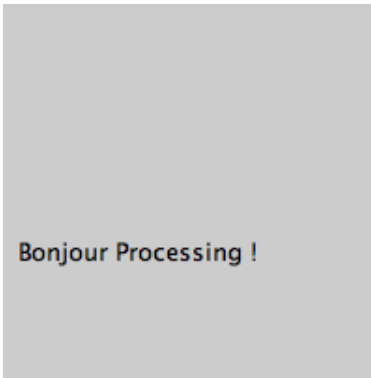

```
int y = 0;

void setup() {
  size(130,200);
  textSize(80);
  stroke(0);
  fill(0);
}

void draw() {
  background(204);
  line(10,100,120,100);
  text(key,35,y);
}

void keyPressed(){
  y = 180;
}

void keyReleased(){
  y = 80;
}
```

TOUCHES SPÉCIALES

Les touches spéciales comme les flèches (`UP`, `DOWN`, `LEFT`, `RIGHT`) ou `ALT`, `CONTROL`, `SHIFT` sont quant à elles stockées dans la variable `keyCode`. Le test `if (key == CODED)` permet de vérifier si la touche appuyée est une touche spéciale ou non. Dans un programme, il faudra distinguer les deux cas en fonction du type de touche appuyée que l'on veut tester.

Dans l'exemple suivant, nous allons créer une machine à écrire un peu particulière, puisqu'il sera possible de déplacer le texte grâce aux flèches.

A chaque fois qu'une touche sera appuyée, elle sera stockée dans une chaîne de caractère, pour pouvoir être affichée dans le `draw()` à l'aide de la méthode `text()`.




```
String s = "";
int x = 50;
int y = 50;

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

void draw() {
  background(255);
  fill(0);
  text(s, x, y);
}

void keyPressed() {
  if (key == CODED){
    if (keyCode == LEFT)   x  = x - 1;
    if (keyCode == RIGHT)  x = x + 1;
    if (keyCode == UP)     y = y - 1;
    if (keyCode == DOWN)   y = y + 1;
```

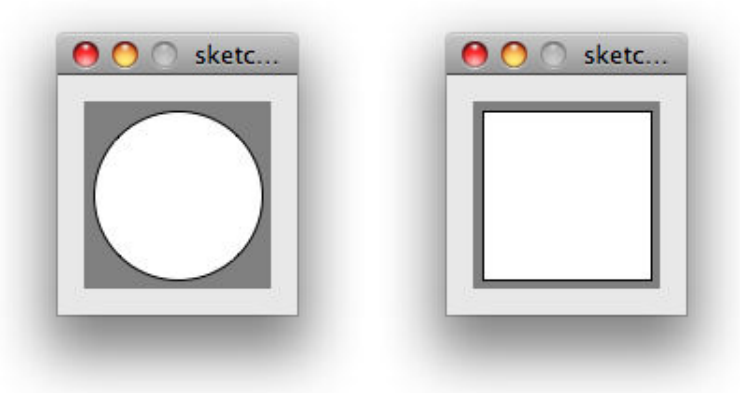
```
  }
  else {
    s = s + key;
  }
}
```

Notons l'omission des accolades après chaque `if`. Cette syntaxe est possible si le `if` ne s'applique qu'à une seule instruction. Nous utilisons par ailleurs une variable de type `String` dénommé dans cet exemple "s" dont la fonction est de stocker une suite de caractères, comme des phrases, par exemple.

VÉRIFICATION SPÉCIFIQUE DE L'APPUI D'UNE TOUCHE

Processing nous permet également de faire appel à la variable `keyPressed` et de la définir. Attention ! Bien que la syntaxe soit identique, elle n'est pas à confondre avec la méthode `keyPressed()` abordée au début de ce chapitre. Cette variable va nous informer en permanence si une touche est appuyée ou non et pourra être utilisée dans la méthode `draw()` notamment.

Dans l'exemple suivant, nous allons dessiner un cercle si la touche appuyée est `'c'`, un carré si la touche appuyée est `'r'`.




```
void draw()
{
  rectMode(CENTER);
  background(128);
  if (keyPressed == true)
  {
    if (key == 'c') {
      ellipse(50,50,90,90);
    }
    else if (key == 'r') {
      rect(50,50,90,90);
```

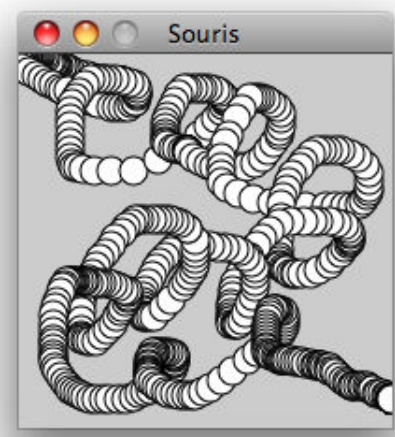
```
        }
    }
}
```

29. LES ÉVÉNEMENTS SOURIS

Dans ce chapitre, nous allons voir comment interagir avec la souris, en récupérant des informations comme sa position ou bien les actions de clics. Pour illustrer ces fonctionnalités, nous allons créer un petit logiciel de dessin, à base de cercles, en introduisant même du mouvement aléatoire !

POSITION DE LA SOURIS

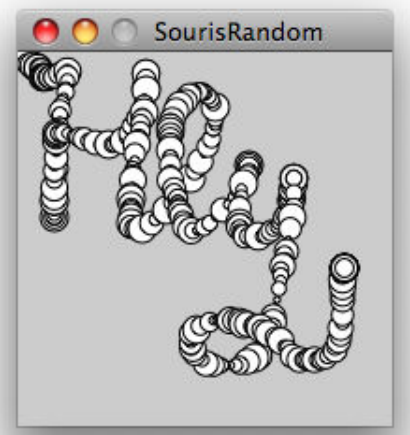
Les coordonnées de la souris dans la fenêtre sont accessibles par les deux variables `mouseX` et `mouseY`, disponibles par défaut dans Processing. Elles permettent de connaître la position de la souris par rapport à notre fenêtre de dessin, en prenant pour origine le coin gauche supérieur de la fenêtre. Dans l'exemple suivant, nous allons créer un programme qui va dessiner un cercle à partir de la position de la souris.




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

void draw() {
```

```
  ellipse(mouseX,mouseY,15,15);
}
```

A présent, nous allons modifier très légèrement l'exemple précédent en changeant dynamiquement le rayon du cercle. Celui-ci est choisi au hasard grâce à la méthode `random()`, qui génère un nombre aléatoirement au sein d'une plage de valeur. A chaque fois que Processing va exécuter `draw()`, la variable `r` sera remplie avec un nombre au hasard compris entre `3` et `30` : `12`, `25`, `23`, `11`, `22`, `4`, `10`, `11`, `25` ... Le changement de valeur du rayon va provoquer le tremblotement du cercle autour du curseur de la souris.




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

void draw() {
  float r = random(3,30);
  ellipse(mouseX,mouseY,r,r);
}
```

Dès que la souris sort de la fenêtre, la position de la souris n'est plus relayée à notre programme. Les cercles se dessinent et s'accumulent à la dernière position captée par notre sketch.

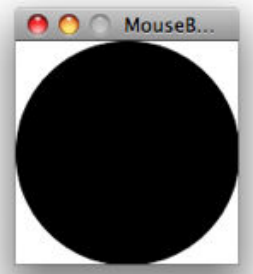
CLICS DE SOURIS

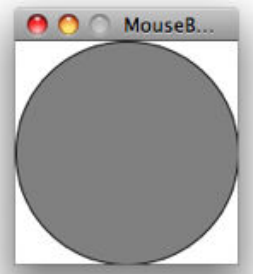
Nous pouvons intercepter les clics de souris grâce aux méthodes `mousePressed()` et `mouseReleased()`.

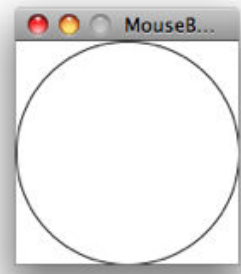
Ces deux méthodes permettent de savoir si l'utilisateur a appuyé ou relâché un des boutons de la souris. Reprenons l'exemple précédent en changeant la couleur de remplissage de notre cercle lorsque l'utilisateur appuie sur un des boutons de la souris. Nous allons choisir un niveau de gris pris au hasard au moment du clic que nous appliquerons au cercle à l'aide de la commande `fill()`.

```
float ton_de_gris = 255;

void setup() {
  size(300, 300);
  smooth();
}

void draw() {
  float r = random(10, 80);
  fill(grey);
  ellipse(mouseX, mouseY, r, r);
}

void mousePressed() {
  grey = random(255);
}
```

Processing nous permet d'identifier quel bouton de la souris a été appuyé. Pour cela, nous pouvons utiliser la variable `mouseButton` qui va contenir soit `LEFT`, `RIGHT` ou `CENTER` correspondant au bouton droit, gauche et central (si la souris dispose de tous ces boutons). Utilisons cette variable dans un exemple pour dessiner un cercle qui change de couleur en fonction du type de bouton pressé.








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

void draw() {
  background(255);
  ellipse(100,100,200,200);
}

void mousePressed() {
  if (mouseButton == LEFT)   fill(0);
  if (mouseButton == RIGHT)  fill(255);
  if (mouseButton == CENTER) fill(128);
}
```

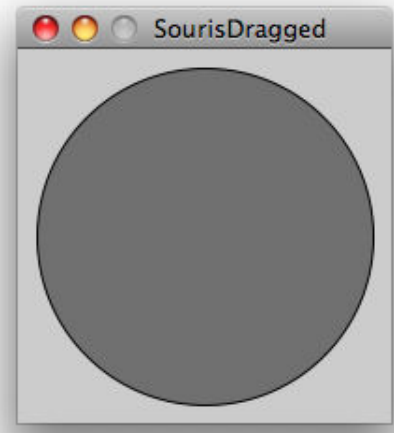
AUTRES ÉVÉNEMENTS

Processing permet de capter deux autres événements souris, notamment lorsqu'elle est en mouvement au-dessus de la fenêtre.

`mouseMoved()` va permettre de détecter le mouvement de la souris lorsque celle-ci se déplace au dessus de la fenêtre de dessin. Si le pointeur de souris sort de la zone de la fenêtre ou s'il ne bouge plus, alors la méthode n'est plus appelée.

`mouseDragged()` est appelée lorsque l'utilisateur a cliqué sur un bouton tout en bougeant la souris au dessus de la fenêtre. Cette instruction permet notamment de gérer le glisser-déposer, par exemple. `mouseDragged()` continue d'être actif même si la souris sort de la fenêtre.

Dans l'exemple suivant, nous allons utiliser ces méthodes. Lorsque `mouseMoved()` est appelé suite à l'action de l'utilisateur, on change la couleur de remplissage du cercle et quand c'est `mouseDragged()` qui l'est, on ajuste sa taille.



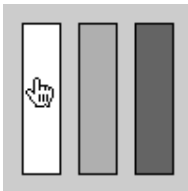

```
int r = 100;
int c = 100;

void setup() {
  size(255, 255);
  smooth();
}

void draw() {
  background(255);
  fill(c);
  ellipse(width/2, height/2, r, r);
}

void mouseMoved() {
    c = mouseY;
}

void mouseDragged() {
  r = mouseX;
}
```

CURSEUR

Parfois il est bien pratique de cacher le curseur de la souris, par exemple dans le cas d'une installation artistique où le pointeur pourrait venir malencontreusement parasiter l'affichage de votre animation. Dans Processing, il existe fort heureusement une instruction pour masquer le pointeur :

```
noCursor();
```

Il est également possible de modifier la forme du curseur pour signaler certains évènements à l'utilisateur, par exemple changer la forme du curseur au survol d'un élément. Il suffit pour cela d'utiliser `cursor()` avec comme paramètre une des valeurs suivantes : `ARROW`, `CROSS`, `HAND`, `MOVE`, `TEXT`, `WAIT`. L'exemple ci-après affiche une forme différente de curseur (une flèche, une croix, une main, un sablier) selon la zone de dessin survolée par la souris.



```
void setup() {
   fill(255);
   rect(10, 10, 20, 80);
   fill(175);
   rect(40, 10, 20, 80);
   fill(100);
   rect(70, 10, 20, 80);
}

void draw() {
  // Nous vérifions si on survole un des 3 rectangles
  // et modifions le curseur en conséquence

   if (mouseX > 10 && mouseX < 30 && mouseY > 10 && mouseY < 90) {
      cursor(HAND); // Affiche une main
   }
   else
   if (mouseX > 40 && mouseX < 60 && mouseY > 10 && mouseY < 90) {
      cursor(ARROW); // Affiche une flèche
   }
   else
   if (mouseX > 70 && mouseX < 90 && mouseY > 10 && mouseY < 90) {
      cursor(WAIT); // Affiche un sablier
  }
  else {
     cursor(CROSS); // Affiche une croix si on ne survole rien
  }
}
```

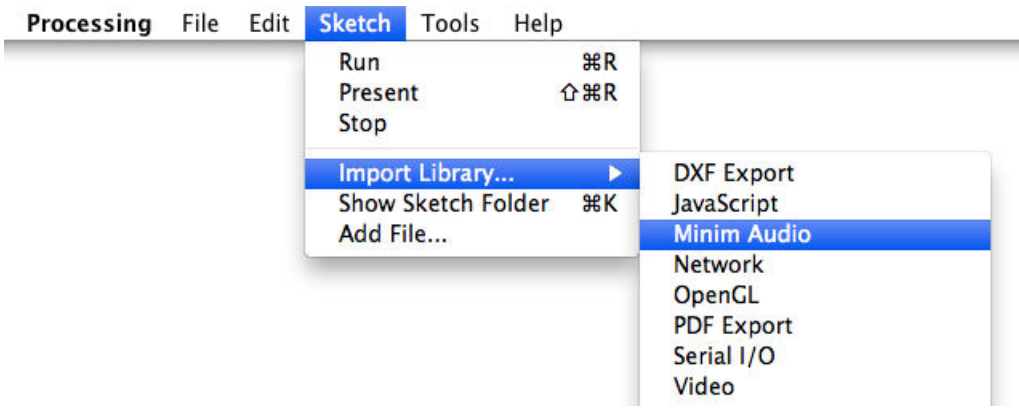
30. L'ENTRÉE MICROPHONE

Si votre sketch peut être programmé pour être sensible à l'appui d'une touche du clavier ou au clic de la souris, ce n'est pas la seule forme d'interactivité offerte par Processing. Vous pouvez également donner à votre programme le sens de l'ouïe en vous servant d'un microphone comme oreille et en faisant correspondre des actions du sketch à certains paramètres du son capté.

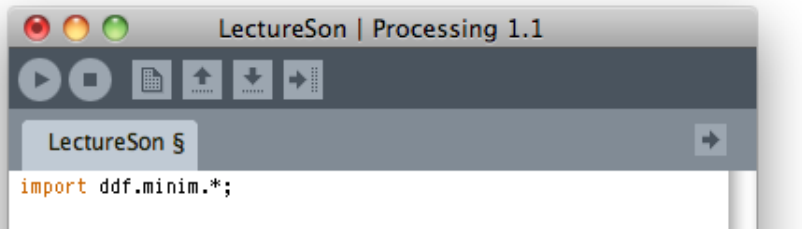
Nous allons tout d'abord rappeler certaines notions abordées dans le chapitre consacré à la lecture du son pour ensuite nous concentrer plus spécifiquement sur l'ajout d'une source sonore à l'aide d'un microphone.

MINIM

Il a été décidé à un moment donné d'intégrer dans toute distribution Processing la bibliothèque audio Minim pour pouvoir jouer au moins des fichiers audio ou capter le son entrant par le microphone. A priori cette bibliothèque (également appelé librairie en jargon informatique) est déjà installée sur votre ordinateur. Pour vérifier sa présence et l'intégrer dans votre programme, il suffit d'aller dans le menu *Sketch › Import Library... › Minim audio*.



Selon votre plate-forme et votre version, à la suite de cette action, Processing ajoutera des lignes de code plus ou moins nombreuses en haut de votre programme. Par défaut, il faut au moins voir affichée l'instruction suivante :




```
import ddf.minim.*;
```

C'est cette instruction (que vous pouvez saisir à la main si vous le voulez) qui importera l'ensemble des fonctionnalités de la bibliothèque Minim pour les rendre accessibles à notre programme. C'est à partir de cette instruction `import ddf.minim.*;` que notre sketch va pouvoir jouer du son.

CONFIGURER MINIM

Le lancement de notre sketch à l'aide du bouton **Run** ne suffit pas pour jouer un son. Au préalable, il faut prévoir d'activer les fonctionnalités de gestion audio de Minim (son moteur audio en jargon informatique). Pour jouer un son, Minim doit en effet auparavant demander un accès à la carte son de votre ordinateur. Ne vous inquiétez pas, Minim s'occupe de toute cette cuisine interne pour vous, il suffit de lui dire de s'activer pour cela. Le fait d'avoir activé Minim suppose de prévoir de le désactiver à la fin de notre programme.

Si vous ne comprenez pas tout ce que nous venons de dire, ce n'est pas trop grave. Sachez simplement qu'il faut saisir le code suivant au début de chaque programme utilisant `Minim` :

```
import ddf.minim.*;
Minim minim;

void setup() {
  minim = new Minim(this);
}
```

Et qu'il faut également écrire la fin du programme les instructions suivantes :

```
void stop() {
  minim.stop();
  super.stop();
}
```

AJOUTER UNE SOURCE SONORE

Pour exploiter l'entrée microphone, nous devons créer un objet *source sonore* qui permettra d'accéder au son capté par le microphone.

```
import ddf.minim.*;

Minim minim;
AudioInput in;

void setup() {
  minim = new Minim(this);
  in = minim.getLineIn(Minim.STEREO, 512);
}
```

Il faut également prévoir d'arrêter la capture du son ainsi que la librairie *Minim* à la fin de l'exécution du programme. A cet effet, nous employons la méthode `stop();`

```
void stop() {
  in.close();
  minim.stop();
  super.stop();
}
```

VISUALISER LE NIVEAU SONORE

L'objectif à présent est de faire varier la couleur du fond de la fenêtre de l'espace de dessin en fonction des sons captés par le micro. Plus le son sera fort, plus le fond sera blanc. Nous allons utiliser l'objet source audio (`in`). qui comporte trois sous objets : `left`, `right` et `mix`. Dans le cas d'un microphone stéréo, ces trois éléments représentent le canal gauche, droit et la moyenne des deux. A partir de ces canaux, il est possible de connaître le niveau sonore du son capté à l'aide de la méthode `level()` qui retourne au programme cette information sous la forme d'une valeur chiffrée.

```
 void draw() {
  background(in.mix.level()*2550);
}
```

Une capture d'écran de l'espace de dessin avec un niveau sonore moyen :

UN PETIT JEU

Nous allons à présent créer un jeu très simple qui va exploiter davantage les possibilités d'interaction avec un microphone. Le jeu sera composé d'une balle partant de la gauche de l'écran et qui subit une force d'attraction lente, mais permanente vers cette partie de l'espace de dessin. En faisant du bruit, l'utilisateur va la pousser vers la droite, son but étant de passer la ligne d'arrivée.

Pour commencer, nous allons changer la taille de notre fenêtre de visualisation de l'espace de dessin, activer le lissage du tracé et définir le contour des formes en lui appliquant la couleur blanche. Le code concerné est signalé en gras.

```
void setup() {
  size(600, 100);
  smooth();
  stroke(255);

  minim = new Minim(this);
  in    = minim.getLineIn(Minim.STEREO, 512);
}
```

Nous allons ensuite créer une variable qui stockera la position de la balle. Elle sera déclarée en en-tête du programme pour être disponible dans l'ensemble du sketch. Nous lui assignons la valeur o.

```
float ballX = 0;

void setup() {
...
```

Dans la méthode `draw()`, nous allons définir un fond noir et dessiner une balle qui réagit en fonction du niveau du son. A chaque appel de la méthode `draw()` nous ajoutons le niveau sonore du microphone à la coordonnée x de la balle. Elle va se mettre en mouvement au fur et à mesure que nous faisons du bruit.

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

  ballX = ballX + in.mix.level()*20;

  ellipse(25 + ballX, height - 25, 50, 50);
}
```

Pour éviter que la balle ne sorte de l'écran, nous allons ajouter deux conditions qui vont corriger sa position si la variable `ballX` est plus petite que 0 ou plus grande que la largeur du sketch.

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

  ballX = ballX + in.mix.level()*20;

  if (ballX < 0) {
    ballX = 0;
  }

  if (ballX > width-25) {
    ballX = width-25;
  }

  ellipse(25+ballX, height-25, 50, 50);
}
```

Nous allons ajouter une « ligne d'arrivée » ainsi qu'une condition qui spécifie que si la balle l'a franchi, elle change de couleur.

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

  ballX = ballX + in.mix.level()*20;

  if (ballX < 0) {
    ballX = 0;
  }

  if (ballX > width-25) {
    ballX = width-25;
  }

  if (ballX > 500) {
    fill(255, 0, 0);
  } else {
    fill(255);
```

```
  }
  line(500, 0, 500, 100);
  ellipse(25+ballX, height-25, 50, 50);
}
```

Afin de compliquer le jeu, nous allons ajouter un comportement à la balle qui la fait revenir en arrière en permanence. A chaque appel de la fonction `draw()` nous allons diminuer légèrement sa position sur l'axe x.

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

  ballX = ballX - 0.5;
  ballX = ballX + in.mix.level()*20;

  if (ballX < 0) {
    ballX = 0;
  }

  if (ballX > width - 25) {
    ballX = width - 25;
  }

  if (ballX > 500) {
    fill(255, 0, 0);
  } else {
    fill(255);
  }

  line(500, 0, 500, 100);
  ellipse(25 + ballX, height - 25, 50, 50);
}
```

A présent, il ne vous reste plus qu'à faire appel à votre imagination pour utiliser du son dans vos projets et en exploiter les nombreuses possibilités : analyse des fréquences d'un son, création d'effets sonores sur mesure à l'aide de fonctionnalités de synthèse audio, etc.

31. L'ENTRÉE VIDÉO

Processing permet de capturer les images provenant d'une caméra vidéo connectée à votre ordinateur par un câble ou à distance via le Wifi ou même Internet. Ces images peuvent ensuite être affichées dans la fenêtre de visualisation de votre sketch et le cas échéant, modifiées en fonction de vos besoins. Les applications créatives de l'entrée vidéo sont multiples.

LES TYPES DE CAMÉRAS SUPPORTÉES

Les caméras les plus simples à utiliser sont les caméras USB, souvent appelées caméras Web. Il est également possible d'employer des caméras que l'on branche sur un port IEEE1394 (Firewire) comme les caméras DC1394 et les caméras DV.

Pour exploiter l'entrée vidéo, vous pouvez utilisez la librairie `video` de Processing qui fait appel à la librairie QuickTime, de la compagnie Apple Computer. Toutes les caméras supportées par QuickTime devraient donc fonctionner sans aucune difficulté. Apple fournit cette librairie uniquement pour les systèmes d'exploitation Mac OS X et Microsoft Windows. Sur GNU/Linux, il est préférable d'utiliser la librairie GSVideo. Celle-ci fait usage de l'excellente GStreamer.

CAPTURER DES IMAGES AVEC LA LIBRAIRIE VIDÉO

Sous réserve que vous soyez sur Mac OS X ou Windows, voyons à présent comment utiliser la librairie `video`. Tout d'abord il faut importer la librairie vidéo en cliquant dans le menu: *Sketch › Import Library ›... › Video*. Une ligne de code s'affichera au début de votre programme :

```
import processing.video.*;
```

Ensuite nous allons déclarer une variable qui stockera notre objet `Camera`. Dans la méthode `setup()`, il faut préciser laquelle des caméras présentes sera utilisée. Notez que vous devez déjà avoir connecté et installer cette caméra vidéo sur votre ordinateur pour faire fonctionner cet extrait de code. Si le code génère une erreur au moment de créer la caméra, essayez d'autres numéros de `devices[]`.

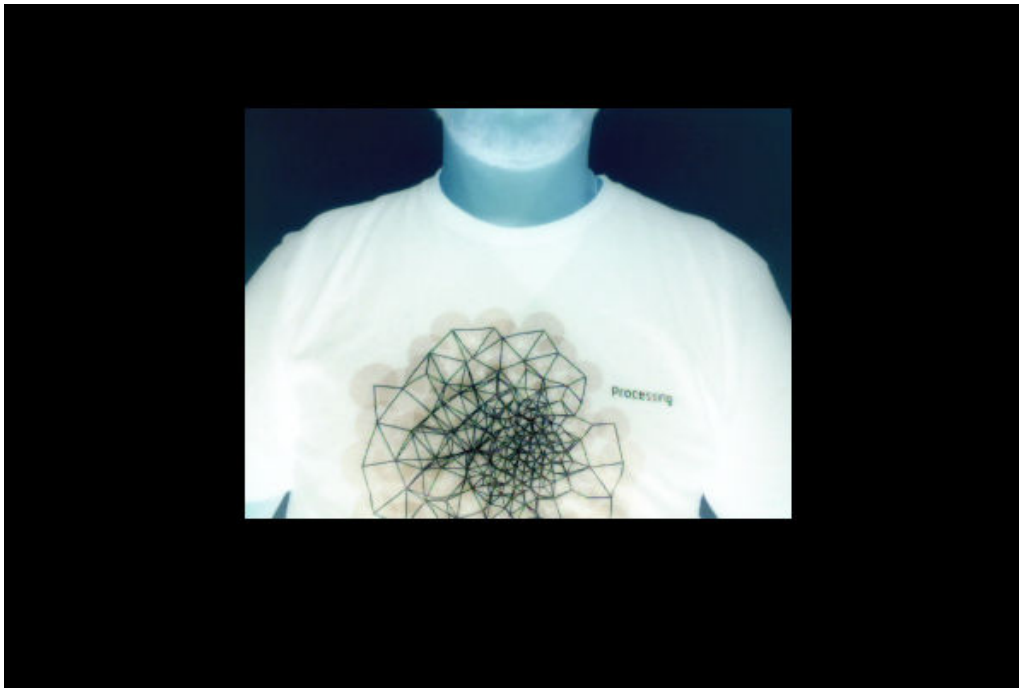
```
Capture camera;

void setup() {
```

```
  size(640, 480);
  background(0);

  String[] devices = Capture.list();
  println(devices);

  camera = new Capture(this, 320, 240, devices[0]);
}
```

L'étape suivante consiste à préciser, dans la méthode `draw()`, que nous affichons la dernière image obtenue par la caméra en vérifiant qu'une nouvelle est bien disponible. En effet, les images sont reçues de la caméra vidéo à une certaine fréquence, par exemple 30 fois par seconde, ce qui ne correspond pas forcément à celle de notre sketch. Dans l'exemple ci-après, nous avons demandé à notre programme d'appliquer sur l'image captée un filtre de solarisation (inversion des valeurs d'ombre et de lumière de l'image).

```
void draw() {
  if (camera.available()) {
    camera.read();

    camera.filter(INVERT);
    image(camera, 160, 100);
  }
}
```

La variable de type `Capture` peut être utilisée de la même manière qu'une variable de type `PImage` : il est possible de lire les pixels de l'image, la transformer, l'afficher plusieurs fois à l'écran, et bien d'autres choses encore.



Le programme complet de notre exemple :

```
import processing.video.*;

Capture camera;

void setup() {
  size(640, 480);
  background(0);

  String[] devices = Capture.list();
  println(devices);

  camera = new Capture(this, 320, 240, devices[0]);
}

void draw() {
  if (camera.available()) {
    camera.read();

    camera.filter(INVERT);
    image(camera, 160, 100);
  }
}
```

ANNEXES

32. Les astuces
33. Les erreurs courantes
34. Les librairies externes
35. La documentation en ligne
36. Processing dans l'enseignement
37. Arduino
38. A propos de ce manuel
39. Glossaire
40. Credits

32. LES ASTUCES

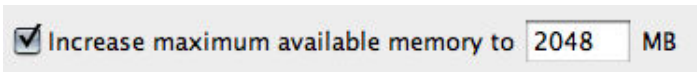
Pour vous éviter certaines erreurs de base et aller plus loin dans l'utilisation de Processing, ce chapitre présente plusieurs astuces et morceaux de code qui permettent de résoudre une série de problèmes fréquemment rencontrés. N'hésitez pas à les utiliser dans vos programmes.

AUGMENTER LA MÉMOIRE

Si vous créez des sketchs qui utilisent de la vidéo, de nombreuses images ou qui travaillent avec des données volumineuses, il peut arriver que votre programme génère une erreur pour cause de mémoire insuffisante. Voici l'erreur affichée par Processing dans la console.

```
OutOfMemoryError: You may need to increase the memory setting in Preferences.
```

Pour résoudre ce problème, vous devez ouvrir les préférences de Processing et changer la quantité de mémoire allouée à votre sketch.



INVERSER LA VALEUR D'UN BOOLÉEN (VRAI/FAUX)

Voici un mécanisme permettant d'inverser la valeur d'un booléen sans avoir à tester à l'aide de la condition if/ then si sa valeur est vraie ou fausse . Cette astuce repose sur le fait qu'en précédant la valeur booléenne d'un point d'exclamation, on obtient son contraire.

```
boolean afficheInfo;

void setup() {
  afficheInfo = false;
}

void draw() {
  background(0);
  // Si vrai, on affiche le texte
  if (afficheInfo) text("Info", 10, 20);
```

```
}

void keyPressed() {
   // on affecte à afficheInfo l'inverse de sa valeur actuelle
   afficheInfo = ! afficheInfo;
}
```

INTERVALOMÈTRE AVEC FRAMERATE(), FRAMECOUNT ET MODULO

Si l'on souhaite effectuer une action à intervalle régulier, le procédé le plus simple consiste à utiliser les méthodes de comptage d'images de Processing : `frameRate()`, `frameCount` ainsi que la commande `%` (modulo). Notez que cette astuce ne permet de gérer qu'un seul intervalle de temps par programme. Détaillons chacune des ces fonctions :

- la variable `frameCount` permet de connaître le nombre de fois où `draw()` a été appelé depuis le lancement du programme.
- `frameRate()` fixe la fréquence des appels à `draw()` par seconde (par défaut 30).
- l'opérateur `%` (modulo) retourne le reste de la division de deux nombres, ce reste étant strictement inférieur au nombre diviseur (c'est une règle mathématique!). Ainsi `7%3` retourne 1 ou encore `8%2` retourne `0`. L'opérateur modulo permet de compter sans jamais dépasser un certain nombre (la base du modulo).

Ainsi, si l'on veut effectuer une action toutes les secondes, on la déclenchera dans le cas où le reste de la division de frameCount par la valeur passée à `frameRate()` est bien égal à zéro. Dans l'exemple qui suit, nous changeons la couleur de fond à chaque seconde en vérifiant que notre compteur est un multiple de 24.

```
color couleur;

void setup() {
  couleur = 0;
  frameRate(24);
}

void draw() {
  background(couleur);
  if (frameCount % 24 == 0) {
     couleur = (int) random(255);
  }
}
```

Exercice : modifiez le code ci-dessus pour que le changement de couleur ait lieu toutes les 3 secondes.

CRÉER UNE CLASSE INTERVALOMÈTRE

Dans l'astuce précédente, nous avons vu comment créer un intervalomètre simple, très utile pour exécuter une action à intervalle régulier, mais qui comporte certaines limitations. A présent, nous allons apprendre à programmer un intervalomètre gérant plusieurs intervalles de temps au sein d'un même sketch, et ce, avec précision (sans variation de vitesse lorsque votre ordinateur est occupé à exécuter simultanément de nombreuses tâches). A titre d'illustration, nous allons créer un objet `Intervalometre` chargé d'exécuter un bloc de code à un intervalle régulier. Le système de comptage fonctionne sur le principe suivant : mémoriser le moment auquel l'objet a été exécuté pour la dernière fois et à chaque appel de la méthode `draw()` de vérifier si le temps actuel est plus grand que ce moment de temps antérieur auquel on a ajouté un intervalle donné. Nous allons nous aider de la méthode `millis()` qui retourne le nombre de millisecondes depuis le lancement du programme.

Notre classe sera composée de deux caractéristiques et d'une action :

- L'intervalle de temps (un intervalle que vous aurez déterminé à votre convenance)
- Le temps de la dernière vérification de l'intervalomètre
- Une action qui vérifie si le prochain intervalle a été franchi.

```
class Intervalometre {
  int intervalle;
  int dernier_tic;

  Intervalometre(int intervalle_initial) {
    intervalle = intervalle_initial;
    dernier_tic = millis();
  }

  boolean verifierIntervalle() {
    if (millis() > dernier_tic + intervalle) {
      dernier_tic = millis();
      return true;
    } else {
      return false;
    }
  }
}
```

Ici, la méthode `verifierIntervalle()` vérifie si l'intervalle de temps est écoulé depuis la dernière fois. Elle retourne vrai ou faux. En d'autres termes, cela revient à décider d'exécuter une certaine action à chaque fois que notre horloge fait **tic**.

La dernière étape de programmation de notre sketch va consister à déclarer une variable qui stockera notre intervalomètre. Chaque appel de la méthode `draw()`

exécutera `verifierIntervalle()` de notre objet `Intervalometre`. Si cette méthode nous retourne vrai, nous allons faire apparaître un rectangle positionné au hasard sur l'écran. L'intervalle de temps choisi dans cet exemple est de 100 millisecondes.

```
Intervalometre intervalometre;

void setup() {
  intervalometre = new Intervalometre(100);
}

void draw() {
  if (intervalometre.verifierIntervalle()) {
    rect(random(0, width), random(0, height), 10, 10);
  }
}
```

SUSPENDRE LA RÉPÉTITION DE DRAW()

Tant que l'on n'a pas quitté le programme, et sauf indication expresse de notre part, la fonction `draw()` est appelée en boucle durant toute l'exécution du programme. Il existe cependant un moyen d'interrompre ce cycle à l'aide de la fonction noLoop(). La fonction `loop()` a l'effet inverse.

L'exemple suivant montre comment interrompre et reprendre ce mouvement à l'aide de la touche 's' et de la barre d'espacement.

```
int position = 0;
int increment = 1;

void setup() {
}

void draw() {
  background(0);
  ellipse(50, position, 5, 5);
  // on déplace le point de 1 pixel
  position += increment;
  // si on touche les bords
  if (position > height || position < 0) {
   // alors on inverse la valeur d'incrément
   increment *= -1;
  }
}

void keyPressed() {
```

```
 if (key == ' ') {
  // on relance le cycle
  loop();
 }
 else if (key == 's' || key == 'S') {
  // on arrête le cycle
  noLoop();
 }
}
```

QUITTER LE PROGRAMME

Pour quitter un programme en utilisant du code plutôt qu'une intervention utilisateur, vous pouvez appeler la méthode `exit()`.

```
// Vous pouvez ajouter ce code à l'exemple précédent
else if (key == 'e' || key =='E') {
  exit(); // on quitte le programme
}
```

33. LES ERREURS COURANTES

Processing affiche un certain nombre de messages d'erreur dans la bande grise de la console (qui dans ce cas de figure change de couleur en devenant rouge foncé). Voici une liste des erreurs les plus courantes.

UNEXPECTED TOKEN

Cette erreur se produit fréquemment lorsque vous oubliez un « ; » en fin de ligne ou que vous n'avez pas correctement fermé un bloc d'accolades. Le programme voit une instruction qui fait deux lignes et dont la syntaxe est par conséquent erronée.

```
int nombreEntier  = 1
float nombreVirgule = 0.1;
```

unexpected token: int

Le code correct est :

```
int nombreEntier  = 1;
float nombreVirgule = 0.1;
```

CANNOT FIND ANYTHING NAMED

Cette erreur se produit lorsque vous appelez une variable qui n'existe pas. Vérifiez que vous avez bien déclaré votre variable et que vous avez respecté la portée des variables.

```
int nombre2 = nombre1 + 5;
```

Cannot find anything named "nombre1"

Le code correct est :

```
int nombre1 = 10;
int nombre2 = nombre1 + 5;
```

FOUND ONE TOO MANY...

Cette erreur se produit lorsque vous n'avez pas correctement fermé un bloc par une accolade.

```
void draw() {

}

void setup() {
```

Found one too many { characters without a } to match it.

Le code correct est :

```
void draw() {

}

void setup() {

}
```

ARITHMETICEXCEPTION: / BY ZERO

```
int diviseur = 0;
println(1 / diviseur);
```

ArithmeticException: / by zero

Dans Processing, on ne peut jamais diviser un nombre par zéro. Au besoin, vérifier que le diviseur n'est pas zéro avant de faire la division. C'est la fonction du morceau de code suivant :

```
int diviseur = 0;
if (diviseur != 0) {
  println(1 / diviseur);
} else {
  println("On ne peut pas diviser par zéro!");
}
```

CANNOT CONVERT ... TO ...

Cette erreur survient quand vous essayez d'affecter une valeur qui ne correspond pas au type d'une variable.

```
int a = 10;
float b = 10.5;

a = b;
```

cannot convert from float to int

Dans le cas présent, pour que nous puissions affecter un nombre à virgule à une variable qui est prévue pour stocker un nombre entier, il faut forcer Processing à la transformer. Le code correct est :

```
int a = 10;
float b = 10.5;

a = int(b);
```

ARRAYINDEXOUTOFBOUNDSEXCEPTION

Cette erreur survient lorsque vous tentez d'accéder à un élément en dehors des limites d'un tableau. Dans l'exemple suivant, le tableau à une taille de 3 (case 0, 1 et 2) et nous essayons d'accéder à la case 4.

```
int[] nombres = new int[3];
nombres[0] = 1;
nombres[1] = 20;
nombres[2] = 5;

println(nombres[4]);
```

ArrayIndexOutOfBoundsException: 4

Le code correct est le suivant (la 3e case correspond au rang 2 et non pas au rang 3 ou 4 puisqu'un tableau commence toujours à partir d'une case de rang 0) :

```
int[] numbers = new int[3];
numbers[0] = 1;
numbers[1] = 20;
numbers[2] = 5;

println(numbers[2]);
```

NULLPOINTEREXCEPTION

Cette erreur se produit lorsque vous tentez d'accéder à un élément qui n'existe pas en mémoire. Par exemple en essayant d'accéder à un objet qui n'a pas encore été initialisé.

```
Balle maBalle;

void setup() {
  maBalle.x = 10;
}

class Balle {
  int x;
  int y;
}
```

NullPointerException

Le code correct est :

```
Balle maBalle = new Balle();

void setup() {
  maBalle.x = 10;
}

class Balle {
  int x;
  int y;
}
```

34. LES LIBRAIRIES EXTERNES

La vie est pleine de problèmes intéressants à résoudre. Quand on écrit un bout de code original qui propose de nouvelles fonctionnalités ou une plus grande rapidité d'exécution, on peut vouloir le partager et en faire bénéficier d'autres personnes. Si on le publie sous une licence libre, par exemple la Licence publique générale de GNU (GPL), d'autres programmeurs pourront par la suite améliorer ce code, plutôt qu'à chaque fois être obligé de partir de zéro en « réinventant la roue ».

Une librairie, c'est une collection de classes que l'on peut réutiliser dans chacun de nos projets. Mise à disposition sur Internet, une multitude de librairies intéressantes existe pour Processing. Pour pouvoir en utiliser une, il suffit de la télécharger de son site internet, de décompresser l'archive et de le placer dans le dossier *libraries*. Le dossier libraires se trouve dans le dossier de travail de Processing (menu **Sketch** puis **Show Sketch Folder**).

Par ailleurs, plusieurs librairies sont déjà installées par défaut dans ce dossier lors de l'installation de Processing.

Dans tous les cas, pour les utiliser dans un de vos sketchs, choisir son nom dans le menu **Sketch › Library**. Ces librairies sont libres : les utilisateurs ont la liberté d'exécuter, de copier, de distribuer, d'étudier, de modifier et d'améliorer leur code.

OÙ TROUVER DES LIBRAIRIES

La majorité des librairies ne sont pas installées par défaut avec Processing, mais doivent plutôt être téléchargées via le site web de leur auteur. Cette pratique permet d'obtenir en tout temps la version la plus à jour d'une librairie et simplifie grandement sa distribution.

On peut, sur le site de Processing, accéder à un annuaire des principales librairies à l'adresse suivante : http://processing.org/reference/libraries/.

INSTALLER UNE LIBRAIRIE

Nous allons maintenant installer une librairie que nous téléchargeons de l'internet.

Nous utiliserons la librairie *ShapeTween*, car celle-ci est très simple à utiliser. Elle fournit une variété de moyens d'animer des éléments. Ce genre d'outil est fort utile pour créer des animations fluides.

Pour désigner des animations et des transitions, le mot *tween* est souvent utilisé. Ce mot vient de l'anglais *in between* et désigne les images que l'on dessine entre chacune des images-clés dans le cinéma d'animation traditionnel. Il s'agit donc d'interpolation de mouvements. L'interpolation consiste à déduire des positions intermédiaires entre des points connus.

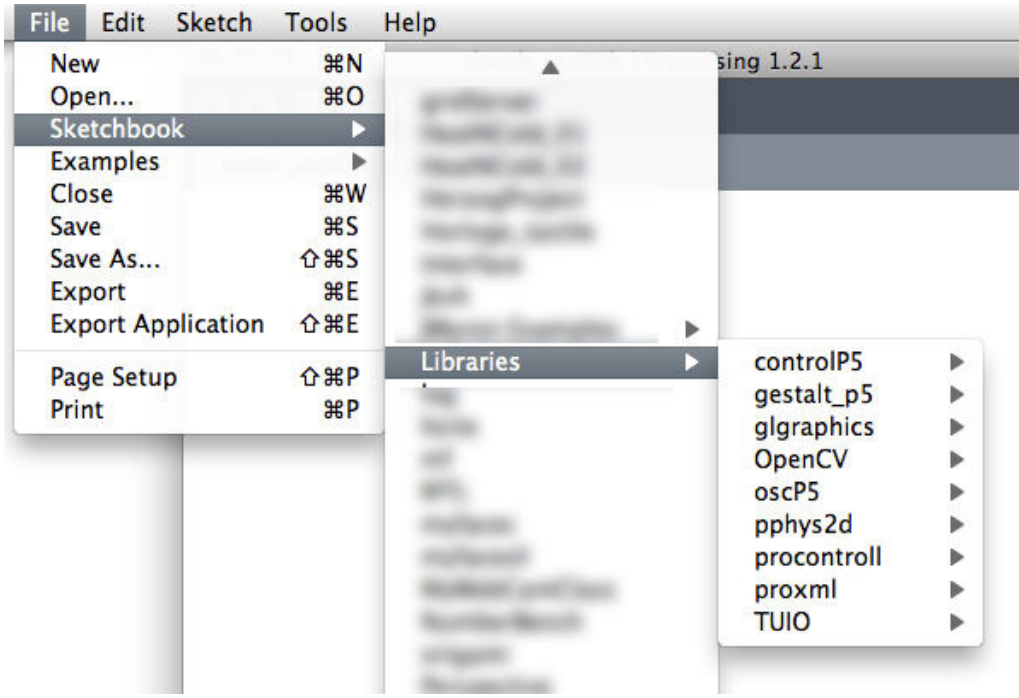
On peut trouver un lien vers la page du projet ShapeTween à partir de la page *Libraries* sur le site de Processing à l'adresse http://www.leebyron.com/else/shapetween/. Sur cette page, vous trouverez le lien intitulé shapetween.zip :

Download & Install
shapetween.zip
Unzip into the Processing "libraries" folder
Sketch → Import Library → shapetween

Téléchargez l'archive et décompressez le fichier zip. Placez son contenu dans le répertoire *libraries* de votre répertoire de sketch. Vous pouvez ouvrir ce dossier en utilisant le menu **Sketch › Show sketch folder**. Le dossier *libraries* se trouve dans le même dossier que vos sketchs. Vous devez y placer le répertoire extrait de l'archive téléchargée ayant le nom de la librairie. Si ce dossier *libraries* n'existe pas déjà, créez le tout simplement. Une fois ces opérations effectuées, quittez l'application Processing et lancez-la à nouveau.

ESSAYER UN EXEMPLE

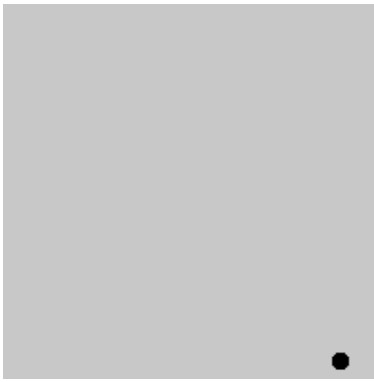
Une bonne librairie fournit également une documentation de chacune des fonctionnalités et instructions offertes (classes, méthodes et attributs). Elle comporte également le plus souvent des exemples. Si c'est le cas, il suffit d'y accéder en parcourant le menu **File › Sketchbook › Examples**. Le nom de notre librairie devrait s'y trouver. Notez que cette procédure fonctionne uniquement sur Mac OS X et Windows. Sur GNU/Linux, les exemples des librairies installées par nous-mêmes ne s'affichent pas dans ce menu.



Si la librairie ne contient pas d'exemples, nous pouvons essayer d'en trouver sur la Toile afin de le copier-coller directement de la page du site vers la fenêtre d'édition de Processing. Comme exercice, nous allons copier-coller un exemple présenté sur le site internet de ShapeTween.

Sur le site Web de la librairie, on peut voir une liste d'exemples sous la rubrique *Example Code*. Cliquez sur *Basic Tween* afin de voir un exemple très simple de mouvement contrôlé à l'aide de cette librairie. Copiez le code proposé sur la page et collez-le dans votre fenêtre d'édition de Processing. Avec la librairie ShapeTween, il est également possible de créer des mouvements qui accélèrent, puis décélèrent, mais les autres exemples proposés sur le site de cette librairie contiennent des erreurs au moment d'écrire ces lignes (le répertoire consacré aux exemples comporte la mention « en cours de construction »).

Après avoir vérifié que l'exemple proposé fonctionne, vous pouvez modifier l'exemple, le simplifier et l'utiliser comme base pour des développements ultérieurs.



```
import megamu.shapetween.*;

Tween ani;

void setup(){
  size( 200, 200 );
  ani = new Tween(this, 2, Tween.SECONDS);
  ani.start();
}

void draw(){
  background(200);

  float x1 = 20;
  float y1 = 30;

  float x2 = 180;
  float y2 = 190;

  float x = lerp( x1, x2, ani.position() );
  float y = lerp( y1, y2, ani.position() );

  fill(0);
  ellipse( x, y, 8, 8 );
}

void mousePressed(){
  ani.start();
}
```

LES LIBRAIRIES PROPRIÉTAIRES

Parfois on télécharge une librairie et on constate que l'archive ne contient pas de code source (des fichiers Java). Puis parcourant le site web, on voit soudain que celui-ci contient la mention « Tous droits réservés », sans plus d'explication. Dans ce cas, cela signifie que la librairie n'est pas libre : elle est propriétaire. Vous pouvez l'utiliser gratuitement, mais sans avoir accès à son code ! Vous ne pouvez pas la modifier, ni étudier son fonctionnement. Il se peut fort bien qu'elle ne soit pas compatible avec la prochaine version de Processing, et dans ce cas, vous ne serez pas en mesure de régler ce problème.

Si ces considérations sont importantes pour vous, soyez vigilants. Vérifiez toujours que le code source est inclus avec une librairie, et que son code comporte bien un en-tête avec la mention d'une licence libre. Le site de la Free Software Foundation fournit une liste de licences libres compatibles avec la GPL http://www.gnu.org/licenses/license-list.fr.html.

35. LA DOCUMENTATION EN LIGNE

Le site web de Processing propose de nombreux contenus pour débuter : tutoriaux, exemples de programmes, etc. On y trouve également une page listant l'ensemble des instructions informatiques (méthodes) utilisées par ce logiciel : leur fonctionnement y est décrit précisément et illustré par des exemples. D'autres sources d'information très utiles existent sur Internet.

LES TUTORIAUX

La section "*learning*" du site de Processing (http://www.processing.org/learning/) propose deux types de contenus : un apprentissage du logiciel pas à pas à travers des exemples basiques et une série de codes plus complets qui illustrent des cas pratiques.

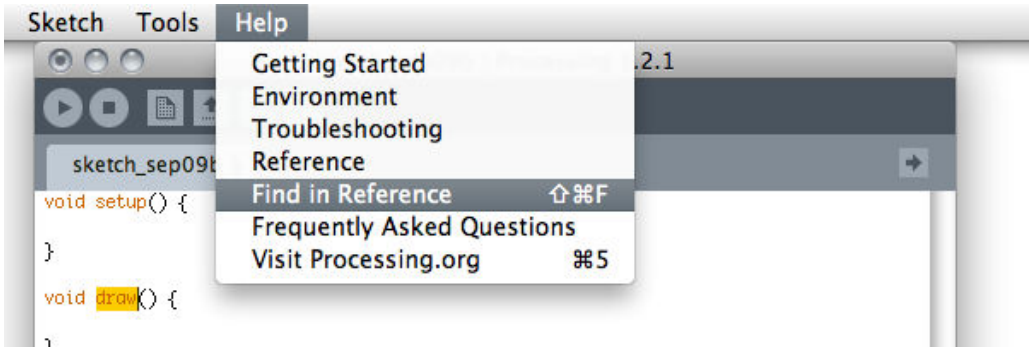
Les codes de ces exemples sont, pour la plupart, fournis avec le logiciel. On les retrouve via le menu **File › Examples**.

LES RÉFÉRENCES

La section "*reference*" du site (http://www.processing.org/reference/) propose une liste complète des méthodes disponibles sous Processing. Elles sont regroupées par thème. Chacune d'elles est décrite au moyen d'un visuel et d'un texte précisant notamment sa syntaxe et la liste de ses arguments.

Dans ces pages de référence, ce sont d'ailleurs les rubriques *Syntax* et *Parameters* qui sont les plus intéressantes. Elles permettent de savoir ce qu'il faut mettre dans les ( ) pour faire fonctionner la méthode et sa syntaxe.

Vous pouvez aussi accéder aux références directement depuis l'environnement de Processing. Pour ce faire, au niveau de la fenêtre d'édition du logiciel, sélectionnez une instruction dont vous souhaitez connaître davantage la syntaxe et les fonctionnalités (attention le mot recherché doit être sélectionné en entier) puis cliquez dans le menu **Help › Find in Reference**. Vous pouvez également effectuer cette opération avec le raccourci clavier CTRL + MAJ + F (Windows & Linux) ou MAJ + CMD + F (Mac Os).



LE FORUM DE PROCESSING.ORG

Le forum du site Processing (http://forum.processing.org/) constitue une autre source d'aide et de contribution. De nombreuses personnes l'utilisent pour exposer leurs problèmes et obtenir des réponses. On y trouve aussi de nombreux morceaux de code que vous pourrez intégrer à vos programmes.

Mentionnons également le site OpenProcessing (http://www.openprocessing.org) qui propose également un espace de partage de morceaux de code entre utilisateurs.

LES RESSOURCES FRANCOPHONES

En plus du présent manuel consultable en ligne (http://fr.flossmanuals.net), il existe plusieurs ressources francophones sur la Toile. En voici une liste non exhaustive :

- Consacré aux langages de programmation dédiés à la création d'images, de vidéo, de son et de musique, le forum Codelab comporte une section consacrée à Processing :
  http://codelab.fr
- Articles d'introduction à Processing de Jean-Noël Lafargue (Université Paris 8) :
  http://www.hyperbate.com/dernier/?page_id=2482
- Cours en ligne de Douglas Edric Stanley (École d'Art d'Aix-en-Provence) :
  http://www.ecole-art-aix.fr/rubrique81.html
- Cours en ligne de Jeff Guess (École nationale supérieure d'art de Paris-Cergy) :
  http://www.guess.fr/processing-tutoriaux/fr/
- Cours en ligne de Marc Wathieu (Ecole de Recherche Graphique - ERG à Bruxelles et Haute Ecole Albert Jacquard - HEAJ à Namur) :
  http://www.multimedialab.be/cours/logiciels/processing.htm

36. PROCESSING DANS L'ENSEIGNEMENT

Processing est un formidable environnement pour l'enseignement. Il est aujourd'hui utilisé par de nombreuses institutions, centres de ressources, associations et donne lieu à divers scénarii pédagogiques : cours, formations, ateliers. Nous nous concentrerons principalement sur l'intérêt de Processing dans les cursus artistiques et créatifs, tels que l'art, le design, le design graphique ou l'architecture. Les contributions d'enseignants d'autres domaines sont les bienvenues, le processus continu de co-rédaction de ce manuel permettant l'ajout de nouveaux paragraphes.

Encore marginaux dans le paysage francophone au début de l'aventure Processing, les initiatives autour de la création algorithmique se sont considérablement développées ces dernières années dans les écoles. Certains établissements dont l'identité repose sur un positionnement affirmé d'ouverture au numérique comme l'Ecole d'Art d'Aix en Provence, ou encore l'Ecole de Recherche Graphique à Bruxelles, proposent à leurs étudiants des programmes où les technologies et des outils tels que Processing sont au coeur des apprentissages. Mais de plus en plus d'écoles d'art et de filières généralistes commencent à intégrer ces pratiques à leurs cursus, alors que dans un premier temps priorité avait été donnée aux grands logiciels propriétaires dans la mise en place de l'apprentissage des outils numériques de création.

APPRENDRE À PROGRAMMER DANS UNE ÉCOLE D'ART ?

L'envers du décor

Tout d'abord, et pour le plus grand nombre des étudiants qui ne vont pas forcément par la suite poursuivre dans ce champ de pratiques, aborder la programmation avec Processing, c'est approcher un langage auquel ils n'ont généralement pas accès, celui de la machine.

Les outils numériques de création sont devenus très présents sinon majoritaires dans la pratique de nombre de créatifs aujourd'hui, et il en est de même pour les étudiants. Les initier à la programmation avec pour objet la création visuelle ou graphique, c'est leur faire appréhender concrètement la nature de l'image numérique ainsi que les transformations qu'elle subit lors de ses multiples traitements par les algorithmes des ordinateurs. Les faire programmer, c'est leur montrer l'envers du décor.

Par ailleurs, les grands logiciels propriétaires sont massivement utilisés et enseignés dans les écoles, sans forcément susciter un accompagnement critique ni une réflexion

sur ces derniers. Initier les étudiants à la programmation, c’est aussi rendre possible un questionnement sur leurs autres pratiques de création numérique en favorisant une mise en perspective.

Une alternative ouverte

Outil libre, Processing offre une alternative de choix aux grands logiciels propriétaires. Si l'on ne fait pas toujours les mêmes choses et qu'on ne les fait pas de la même façon, il n'en demeure pas moins important de donner aux étudiants la possibilité d'appréhender et de pratiquer cet autre type d'outils.

D’un point de vue plus critique, apprendre à programmer ses images au lieu de les coproduire avec des logiciels sur lesquels on possède très peu de prise en terme de configuration, et au final une maitrise partielle, redonne aux créatifs dont la production s'exprime sous forme numérique une grande part d’intentionnalité, à laquelle ils avaient renoncé sans toujours en avoir conscience en travaillant avec les solutions logicielles dominantes.

Apprendre à programmer dans une école d’art, c’est en quelque sorte reprendre le contrôle de sa propre production numérique. D’autre part, la nature libre de Processing détermine aussi la vitalité de sa communauté d’utilisateurs, et des échanges générés sur le web autour de cette plateforme. Enseigner Processing, c’est aussi initier à un état d’esprit, à un mode de fonctionnement ouvert, collaboratif et favorisant l’auto-apprentissage.

Un vaste champ de possibles

Apprendre à programmer avec Processing ouvre un champ de création très vaste, puisque l’utilisateur n’est pas limité aux possibilités offertes par un logiciel, mais seulement par sa propre capacité à mettre en oeuvre ses idées dans l’écriture d’un logiciel. Ce retournement de situation invite à l’expérimentation et à la recherche. L’utilisateur doit trouver les moyens, de mettre en ouvre ses intentions.

Dans les écoles d’art, de design ou encore d’architecture qui dispensent des cours de programmation créative, c’est le développement de cette capacité à « inventer ses moyens » qui est enseignée par l’apprentissage d’un langage, de sa syntaxe, de son vocabulaire, et de ses méthodologies spécifiques. Pour les étudiants qui poursuivent plus loin dans ce champ, l’apprentissage de la programmation pourra leur permettre d’inventer leurs propres outils de création graphique afin qu'ils répondent spécifiquement à leurs besoins et intentions.

Une plateforme souple et intuitive

La simplicité d'utilisation de Processing en fait un médium idéal pour initier à la programmation un public qui a priori n'y était pas prédestiné. Conçu dès le départ pour favoriser une démarche de recherche et d'expérimentation par la pratique, voire l'expression plastique (le terme sketch servant à désigner les programmes créés dans Processing renvoie bien à la notion d'esquisse), ce logiciel offre un environnement adapté à l'apprentissage et à la capacité des créatifs à utiliser du code informatique.

Le langage de programmation utilisé est relativement compréhensible pour qui parle anglais et sa syntaxe relativement aisée à apprendre, y compris pour un non-anglophone. D'autre part, la présence de nombreuses ressources sur le web et la dynamique de la communauté des utilisateurs de Processing favorise une attitude d'auto-apprentissage de la part de l'apprenant qui n'attend pas tout de l'enseignant.

OpenProcessing, un site de ressources pédagogiques

Pour terminer, présentons la plate-forme OpenProcessing (http://www.openprocessing.org), très bon exemple de la vitalité de la communauté des utilisateurs de ce logiciel. Ce site permet de créer des porte-folios pour publier en ligne ses sketchs, les discuter avec d'autres utilisateurs, mais aussi de visualiser les résultats, de les télécharger, ou de les inclure dans une page web.

Si nous citons cette ressource en ligne dans ce chapitre, c'est pour sa rubrique intitulée « Classes » où sont regroupés des porte-folios de projets réalisés en classes ou en ateliers ainsi que des cours. Cette initiative permet à la fois de se rendre compte du nombre croissant de contenus pédagogiques et de résultats d'activités d'apprentissage dédiés à Processing, mais encore et surtout de la diversité des approches.

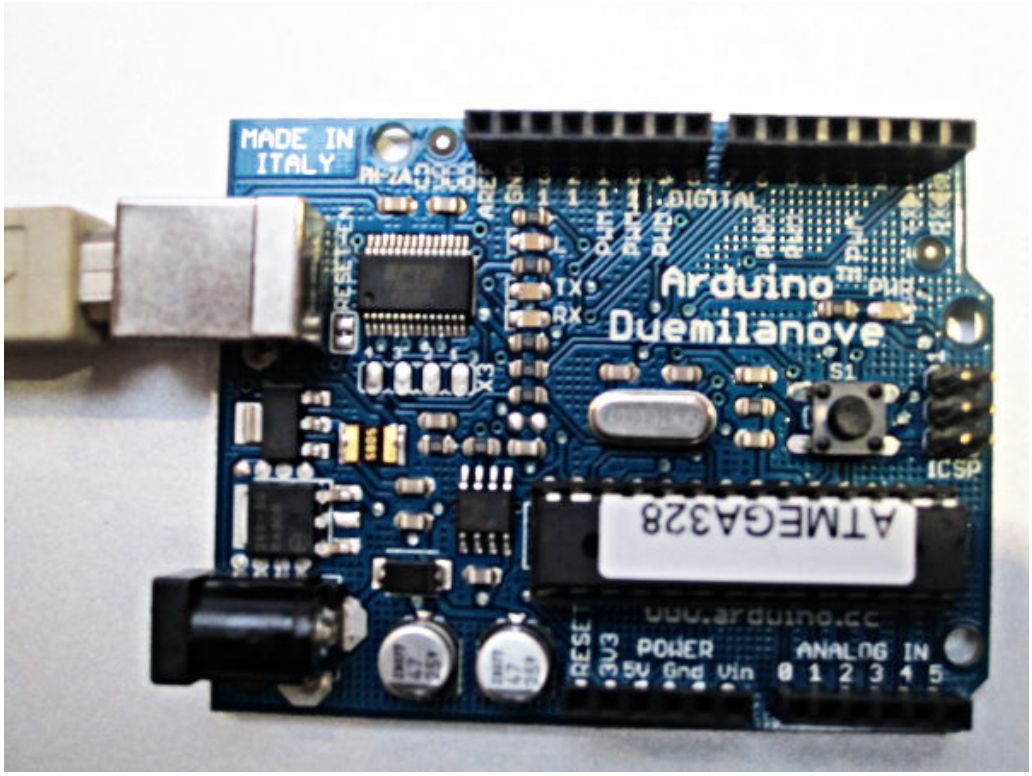
37. ARDUINO

Arduino est une plate-forme libre de création d'objets électroniques à des fins artistiques, éducatives ou de recherche via la conception de prototypes. Elle repose sur l'utilisation d'un **circuit électronique** (un mini-ordinateur, appelé également **micro-contrôleur**) comportant des entrées et sorties (des **ports**) sur lesquelles on peut brancher différents appareils :

- côté entrées, des **capteurs**, appareils qui collectent des informations sur leur environnement comme la variation de température via une sonde thermique, le mouvement via un détecteur de présence, le contact via un bouton poussoir, etc.,
- côté sorties, des **actuateurs**, des appareils qui agissent sur le monde physique, telle une petite lampe qui produit de la lumière, un moteur qui actionne un bras articulé, etc.

Arduino comporte également un environnement de développement logiciel calqué sur celui-ci de Processing, qui permet de programmer le circuit électronique. Arduino étant un projet dérivé de Processing, il apparaît donc comme une ouverture logique à la création interactive. Une fois que vous maîtrisez Processing, Arduino vous sera déjà familier.

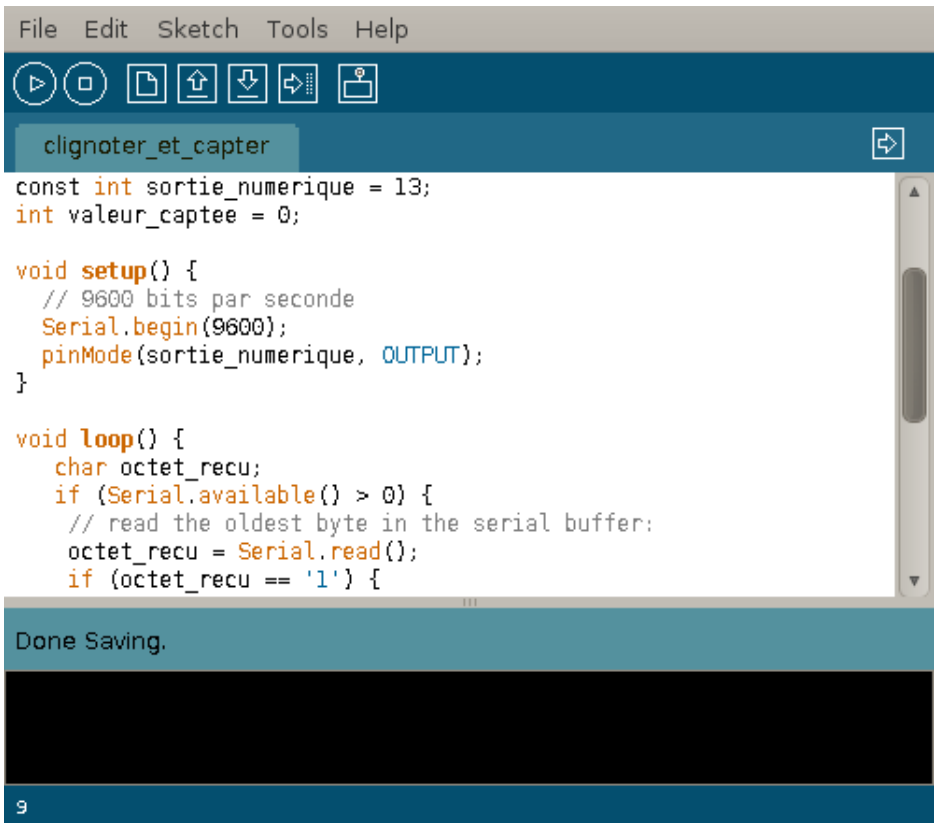
L'APPAREIL ARDUINO



L'appareil Arduino est une plaquette électronique que l'on peut connecter par USB à un ordinateur afin de téléverser du code sur le micro-contrôleur qui s'y trouve. Un micro-contrôleur, c'est un processeur (un mini-ordinateur) de petite taille qui fournit des entrées et sorties analogiques et qui fonctionne selon le programme que l'on aura enregistré dessus.

Cet appareil est autonome. Après avoir reçu le programme que vous aurez conçu, vous pouvez le déconnecter de votre ordinateur et il fonctionnera tout seul (sous réserve de l'alimenter en énergie électrique avec une pile ou mieux une cellule photovoltaïque). Comme le logiciel Arduino, ce circuit électronique est libre. On peut donc étudier ses plans, et créer des dérivés. Plusieurs constructeurs proposent ainsi différents modèles de circuits électroniques programmables utilisables avec le logiciel Arduino.

LE LOGICIEL ARDUINO



L'environnement de programmation vous est certainement familier. On y retrouve les éléments de la fenêtre de travail de Processing : intitulés des menus, boutons, zone d'édition, console, etc.

Comme dans Processing, le programme est constitué d'une série d'instructions saisie dans la fenêtre du logiciel. La spécificité d'Arduino se situe au niveau de la sauvegarde du code qui s'enregistre et s'exécute habituellement depuis le micro-contrôleur et non pas sur votre ordinateur. Le programme se lance dès que l'appareil Arduino est mis sous tension.

Une fois que vous maîtrisez la programmation Arduino, vous pouvez aller plus loin en connectant le micro-contrôleur à d'autres logiciels, notamment à un sketch Processing que vous aurez conçu pour interagir avec cet appareil électronique.

UN DIALOGUE ENTRE LE VIRTUEL ET LE PHYSIQUE

Le circuit électronique Arduino comporte des entrées et sorties sur lesquelles on peut brancher différents appareils. Les capteurs servent à mesurer des informations sur le monde, alors que les actuateurs permettent d'agir sur celui-ci. On peut utiliser des capteurs afin d'influencer ce qui se passe dans un sketch Processing. À l'inverse, on peut utiliser des actuateurs pour que les instructions qui s'exécutent dans le sketch influencent le monde extérieur.

Par exemple, notre programme pourrait lire les informations obtenues de capteurs météorologiques, de capteurs de mouvement ou de simples boutons de contrôle. Ensuite, à partir de ces informations et du traitement qu'en ferait notre sketch, on pourrait contrôler des lumières, des moteurs, ou même des radiateurs ou des ventilateurs. Les applications sont multiples. Tout dépend de l'objectif visé et de l'expérience interactive recherchée.

LA COMMUNICATION SÉRIELLE

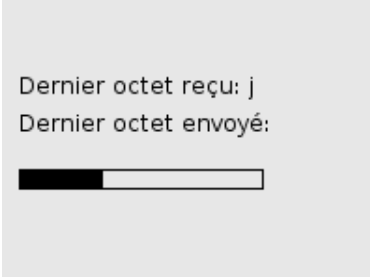
Les ordinateurs communiquent entre eux à l'aide d'impulsions électriques à travers des fils. Les impulsions sont envoyées les unes après les autres, dans une série de bits. Un bit peut être soit vrai, soit faux. C'est le chiffre un ou zéro. Les bits sont souvent groupés en paquets de huit. On appelle ces paquets des octets. Un caractère de texte utilisant le jeu de caractères ASCII est exprimé sur huit bits.

Pour pouvoir échanger ces informations sous forme de bits avec l'extérieur, votre ordinateur utilise une ou plusieurs **entrées physiques** (des **ports** en jargon informatique) lui permettant de se connecter à des appareils périphériques (imprimante, souris, clavier, modem, etc.).

Tout cela pour vous dire que les impulsions électriques et les instructions informatiques ne sont pas si éloignées que cela dans un ordinateur et qu'il est possible de communiquer facilement avec un Arduino à l'aide de simples caractères textuels.

EXEMPLE DE COMMUNICATION AU MOYEN DE LETTRES

Voici un exemple très simple d'envoi et de réception de caractères avec un Arduino : en appuyant sur la barre d'espace de votre clavier d'ordinateur, vous allumez une diode électroluminescente (led) connectée à la sortie numérique 13 de l'Arduino. Pour leur part, les variations électriques captées au niveau de l'entrée analogique 0 d'Arduino font varier la taille du rectangle affichée par votre sketch dans la fenêtre de l'espace de dessin de Processing.



Cet exemple suppose notamment d'avoir appris à connecter votre Arduino à votre ordinateur, procédure qui n'est pas présentée ici. L'objectif de ce chapitre est davantage de vous faire comprendre les potentialités offertes par cet appareil électronique et les capacités d'interaction avec Processing. Pour en savoir plus, nous vous invitons par la suite à consulter les sources d'information disponibles notamment sur Internet.

Le code pour Processing

Envoyer et recevoir du sériel avec Processing est assez facile avec la librairie `processing.serial`. Il faut spécifier le bon numéro de port série (le numéro de ce port, peut être 0, 1, 2, 3...) , ainsi que la bonne vitesse. Ici, notre vitesse est de 9600 bits par seconde.

```
/**
 * Code Processing pour communiquer avec l'Arduino simplement.
 */
import processing.serial.*;

// Attention: Changez ce numéro pour le bon port au besoin
int numero_du_port = 0;  // ...ou 1, 2, 3...

Serial mon_port; // L'objet qui gère le port série
char dernier_envoye = ' ';  // Dernier octet envoyé
char dernier_recu = ' ';  // Dernier octet reçu

void setup() {
  size(200, 150);
  println("Liste des ports: \n" + Serial.list());
  String nom_du_port = Serial.list()[numero_du_port];
  mon_port = new Serial(this, nom_du_port, 9600);
}

void draw() {
  if (mon_port.available() != 0) {
    // Conversion du int en char
    dernier_recu = char(mon_port.read());
```

```
  }
  background(231);
  fill(0);
  text("Dernier octet reçu: " + dernier_recu, 10, 50);
  text("Dernier octet envoyé: " + dernier_envoye, 10, 70);
  int largeur = int(dernier_recu - 'a');
  int multiplicateur = 5;
  stroke(0);
  noFill();
  rect(10, 90, 26 * multiplicateur, 10);
  noStroke();
  fill(0);
  rect(10, 90, largeur * multiplicateur, 10);
}

void keyPressed() {
  if (key == ' ') {
    mon_port.write('1');
    dernier_envoye = '1';
  }
}

void keyReleased() {
  if (key == ' ') {
    mon_port.write('0');
    dernier_envoye = '0';
  }
}
```

Les données sont envoyées sous la forme d'une seule lettre. En effet, la valeur lue de l'entrée analogique zéro du Arduino est convertie en une lettre alphabétique comprise entre a et z. En utilisant une échelle de 25 lettres, on perd un peu de précision par rapport à la valeur chiffrée d'origine, mais cela permet également de démontrer comment les lettres sont en fait encodées avec des chiffres.

Le code pour Arduino

Côté Arduino, on exprime donc la valeur lue au niveau de son entrée analogique zéro au moyen d'une lettre comprise entre a et z. On allume également la diode (Led) associée à la sortie 13. Cette petite lumière est intégrée directement à l'Arduino, aussi nul besoin de souder ou de monter un circuit sur une planche de prototypage.

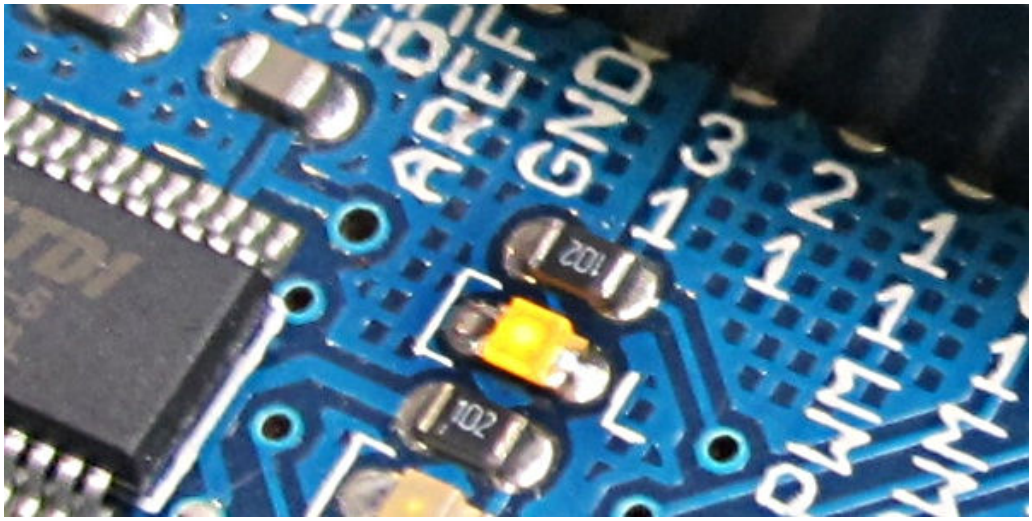
Cet exemple fonctionnera donc avec un Arduino sans aucun montage particulier. Vous pouvez toutefois connecter un potentiomètre à l'entrée analogique zéro si vous souhaitez afficher autre chose que des données aléatoires issues du « bruit » environnant. Notez que par bruit, nous désignons non pas des sons, mais les

variations électrostatiques de l'environnement dans lequel se trouve votre Arduino. Même si l'entrée analogique zéro n'est connectée à aucun fil, des phénomènes électriques se produisent à travers l'air entre ce port et le port « Gnd » de l'Arduino. Idéalement sur une carte Arduino, tous les ports analogiques inutilisés devraient être connectés directement à la borne « Gnd » (Terre) pour éviter ce type d'effets.

Pour téléverser le code qui suit sur un Arduino, il faut installer le dernier logiciel Arduino, et se munir d'une carte (Duemillanove, Uno, etc.). Pour explorer le potentiel de cette interface, il est utile d'en apprendre le fonctionnement, ainsi que ses caractéristiques techniques sur le site officiel : http://www.arduino.cc.

```
/**
 * Ce code doit être téléversé sur le Arduino.
 *
 * Communique avec l'ordinateur pour lui envoyer la valeur
 * de l'entrée analogique 0 et permettre de contrôler la
 * sortie numérique 13, qui a une petite lumière.
 */
const int entree_analogique = 0;
const int sortie_numerique = 13;
int valeur_captee = 0;

void setup() {
  // Vitesse de la communication = 9600 bauds
  Serial.begin(9600);
  pinMode(sortie_numerique, OUTPUT);
}

void loop() {
   char octet_recu;
   if (Serial.available() > 0) {
    // Lisons ce que l'on reçoit via le sériel
    octet_recu = Serial.read();
    if (octet_recu == '1') {
      digitalWrite(sortie_numerique, HIGH);
    } else {
      digitalWrite(sortie_numerique, LOW);
    }
   }
  // Lecture de l'entrée analogique 0:
  valeur_captee = analogRead(entree_analogique);
  char envoyer = 'a' + map(valeur_captee, 0, 1023, 0, 25);
  Serial.print(envoyer);
  // On attend un peu pour laisser le temps au ADC de respirer
  delay(10);
}
```

Une fois que le code pour l'Arduino est installé sur l'appareil, branchez-le en USB sur votre ordinateur, si ce n'est déjà fait, et démarrez le sketch Processing. Vérifier dans le menu **Arduino › Tools** que les configurations de « Board » et « Serial Port » sont correctes. Lorsque la fenêtre du sketch est ouverte dans Processing et que vous appuyez sur la barre d'espacement, la LED associée à la sortie 13 (intégrée à la plaquette Arduino) devrait s'allumer comme suit :



En rajoutant :

```
 println(dernier_recu);
```

...dans le `void draw()` du sketch de Processing, et avant de refermer l'accolade :

```
...
rect(10, 90, largeur * multiplicateur, 10);
println(dernier_recu);
}
```

... on peut visualiser dans la console de Processing le défilement des lettres qui correspondent aux variations électrostatiques captées entre les bornes « o » et « Gnd » de la carte Artuino.

En examinant le code destiné au logiciel Arduino, vous pouvez constater qu'il s'apparente à Processing. Ce n'est toutefois pas le même langage de programmation, il s'agit en fait de C++ et non pas de Java. La volonté de simplifier le processus de programmation se retrouve toutefois dans les deux environnements : Arduino offre aux créateurs la possibilité d'expérimenter relativement facilement la conception d'objets électroniques.

Dans notre exemple, nous avons créé un petit protocole de communication entre l'ordinateur et le micro-contrôleur, entre Processing et Arduino. Relativement simple à réaliser et à comprendre, il n'est ni très flexible ni très optimal. Si vous avez besoin d'un protocole plus robuste, nous vous invitons à visiter le site de Arduino pour trouver plusieurs propositions intéressantes de librairies et d'exemples. Le protocole Messenger (http://www.arduino.cc/playground/Code/Messenger), assez simple, pourra sans doute répondre à vos besoins.

38. A PROPOS DE CE MANUEL

La philosophie du libre inspire la rédaction et la diffusion de ce manuel d'initiation à Processing, l'objectif étant à la fois :

- d'offrir à un public débutant francophone les bases d'utilisation de Processing,
- valoriser la communauté des développeurs et experts francophones de Processing impliqués dans la rédaction et l'actualisation de ce manuel en français,
- fédérer plus largement la communauté francophone de Processing autour d'un projet commun de documentation (tant au niveau des co-auteurs que des commentateurs), sachant que l'ouvrage dans sa forme accessible en ligne (wiki) peut être amélioré et comporter de nouveaux chapitres, notamment à l'occasion de la parution d'une nouvelle version de Processing.

UN OUVRAGE COLLECTIF

Production originale en français, ce manuel est vivant : il évolue au fur et à mesure des contributions. Pour consulter la dernière version actualisée, nous vous invitons à visiter régulièrement le volet francophone de Flossmanuals sur le site http://fr.flossmanuals.net et plus particulièrement sur la page d'accueil du manuel http://fr.flossmanuals.net/processing/.

Le coeur du manuel d'environ 270 pages a été réalisé en 5 jours dans le cadre d'un Booksprint qui s'est tenu à Paris du 6 au 10 septembre 2010 à l'initiative et avec le soutien de l'Organisation internationale de la Francophonie (www.francophonie.org).

Expérimentée et popularisée par la Floss Manuals Fondation dans le cadre de ses activités de création de manuels multilingues sur les logiciels et pratiques libres, la méthodologie du Booksprint permet de rédiger en un temps très court des livres de qualité. Un groupe de 6 experts francophones de Processing originaires d'Amérique du Nord, d'Europe et d'Afrique se sont retrouvés dans un même lieu pour rédiger ce manuel. L’usage de la plate-forme de co-rédaction en ligne a permis également à d'autres personnes intéressées de s’associer à distance à l’expérience.

Co-rédacteurs présents lors du booksprint :

- Douglas Edric Stanley (France)
- Julien Gachadoat (France)
- Horia Cosmin Samoïla (Roumanie)
- Alexandre Quessy (Canada-Québec)
- Adama Dembélé (Mali)
- Lionel Tardy (Suisse)

Facilitateurs :

- Adam Hyde
- Elisa de Castro Guerra

Co-rédacteurs en ligne et contributions externes :

- Jean-Francois Renaud
- Caroline Kassimo-Zahnd
- Jerome Saint-Clair
- Christian Ambaud

N'hésitez pas à votre tour à améliorer ce manuel en nous faisant par de vos commentaires dans la liste de diffusion francophone de Flossmanuals, ou, si vous avez des talents de rédacteur et une bonne connaissance de Processing, à vous inscrire en tant que contributeurs pour proposer la création de nouveaux chapitres. Vous trouverez en fin d'ouvrage la liste complète des personnes ayant participé jusqu'à ce jour à la co-rédaction du manuel.

UN MANUEL LIBRE DISPONIBLE SOUS PLUSIEURS FORMATS ET SUPPORTS

Ce manuel est disponible depuis le site de Flossmanuals sous plusieurs formes : livre imprimé, pages web, pdf et ePub, ce dernier format permettant de le consulter facilement sur des appareils portatifs.

Publié sous licence GPLv2, ce manuel peut être lu et copié librement.

Vous consultez l'édition révisée et augmentée du 14 octobre 2010.

39. GLOSSAIRE

Ce chapitre présente les définitions d'une série de termes techniques liés à Processing, à la programmation ou aux techniques de dessin, d'animation et d'interaction.

APPLET

En Java, un applet (ou une appliquette) est un programme que l'on peut intégrer dans une page web. En exportant votre sketch sous cette forme, Processing permet d'exécuter votre dessin, animation, jeu interactif, etc. de manière autonome sur la page d'un site internet.

APPLICATION

Voir programme.

ARDUINO

Arduino est une plate-forme libre de création d'objets électroniques composée d'un appareil d'entrée-sortie configurable (dénommé un micro-contrôleur) et d'un environnement de programmation. Cet environnement est dérivé de celui de Processing et il est possible d'exécuter des sketchs sur ce type d'appareils programmables. Voir aussi Communiquant (objet).

ARGUMENT

Voir paramètres.

ASSIGNATION

Une assignation désigne le fait d'attribuer une valeur à une variable. On utilise l'opérateur = pour attribuer une valeur à une variable.

ATTRIBUT

Un attribut, c'est une variable qui appartient à une classe ou un objet.

BLOC

Un bloc, désigné par deux accolades { } est un groupement d'instructions qui seront exécutées ensemble. Elles sont utilisées pour délimiter les conditions, les boucles, les méthodes et les classes.

BOOLÉEN

Le booléen est un type de variable qui peut être vrai ou faux. Il permet de tester des conditions.

BOUCLE

Une boucle est un bloc dont les instructions vont être exécutées plusieurs fois à la suite soit selon un nombre prédéfini de fois soit selon une condition. Les instructions `for` et `while` permettent de réaliser des boucles. Chaque étape d'une répétition est appelée itération.

CLASSE

Une classe est un modèle qui sert à créer des objets, ceux-ci partagent un ensemble de méthodes et d'attributs. Chaque objet créé selon ce modèle s'appelle une instance de cette classe. Les objets sont des variables comme les autres. Leur type, c'est le nom de la classe dont ils sont une instance. Vous n'avez rien compris ? Rassurez-vous ! Le plus simple est d'aller lire le chapitre intitulé « Les objets » du manuel.

COMMENTAIRE

Un commentaire est une instruction qui sera ignorée par l'ordinateur au moment de l'exécution du programme. Elle permet au programmeur d'ajouter des notes dans son code afin de le rendre davantage clair : cette méthode facilite la compréhension ultérieure du programme en précisant l'utilité de telle ou telle variable, méthode, etc.

COMMUNIQUANT (OBJET)

Objet technologique dont la fonction est de faire transiter des informations entre le monde physique, le monde des objets et les êtres vivants. Dans le cadre de ce manuel, ce terme s'applique notamment à des objets conçus à partir de petits circuits électroniques autonomes chargés de capter des informations sur leur environnement (mouvement, chaleur, contact...), de les transmettre et éventuellement de réagir en exécutant des actions. Voir aussi Arduino.

CONCATÉNATION

La concaténation est une opération qui consiste à assembler plusieurs variables (texte, nombre, etc.) en une seule chaîne de caractère. On utilise le symbole «« + » pour concaténer des variables.

CONDITION

Les conditions permettent d'exécuter une série d'instructions de manière conditionnelle après qu'une instruction ait été validée ou non.

CONSOLE

La console est la zone située en bas de la fenêtre de l'environnement de développement (la fenêtre d'édition) de Processing. Elle permet d'afficher du texte à des fins de test et de correction d'erreurs (débogage) pendant que notre programme fonctionne.

CONSTRUCTEUR

Dans le jargon informatique, ce terme s'applique à la notion de classe en programmation. Le constructeur est une méthode portant le même nom que la classe invoquée lorsqu'on crée une instance de cette classe. Cette définition vous semble obscure ? Rassurez-vous ! Le plus simple est d'aller lire le chapitre intitulé « Les objets » du manuel.

CONTOUR

Le contour (méthode `stroke()`) définit la couleur utilisée pour dessiner la bordure d'une forme géométrique.

COORDONNÉES

Les coordonnées servent à représenter la position d'un point ou d'un objet dans l'espace sur les axes x, y, et parfois z de l'espace de dessin.

DÉBOGUER, DÉBOGAGE

Ce terme désigne l'action de tester un programme et d'en éliminer les erreurs et les fonctionnements inadéquats. On distingue deux étapes : la suppression des erreurs de syntaxe, de langage et fautes de frappe qui empêchent l'exécution du programme ; la correction de la logique du programme et le traitement des cas de figure non prévus initialement.

DÉCLARATION

La déclaration d'une variable consiste à réserver un espace mémoire par un nom pour un type donné. Une déclaration et une assignation peuvent être combinées.

ESPACE COLORIMÉTRIQUE

Un espace colorimétrique est une manière de gérer la définition des couleurs. Par défaut Processing utilise le mode RVB (rouge, vert, bleu), mais offre aussi l'usage du mode TSV (teinte, saturation, valeur).

FOLIOSCOPE

Un folioscope est une séquence d'images imprimées et reliées sous forme de livret qui peuvent être vues en séquence animée lorsque l'on tourne les pages rapidement avec son pouce. Par analogie cela désigne les animations images par image. En anglais, on utilise le terme *flipbook*.

FONCTION

Une fonction est comme une méthode, mais à l'extérieur d'une classe. Étant donné qu'en Java, on ne peut pas définir de fonction, nous avons évité ce terme dans ce manuel. Cependant, ce terme est beaucoup utilisé sur le site Web de Processing, car le sketch se trouve en fait à l'intérieur d'une classe que nous ne voyons pas. Voir méthode.

HEXADÉCIMAL

La notation hexadécimale est une manière de noter les chiffres en base 16. Dans cette notation, les chiffres supérieurs à 9 sont représentés au moyen de lettres. On l'utilise notamment pour représenter des couleurs pour le Web.

INCRÉMENTATION

Ajouter 1 à la valeur d'une variable de type `int`.

INSTANCE

Une instance est un objet. Voir objet.

INSTRUCTION

Une instruction est un ensemble de méthodes et/ou d'opérations mathématiques, etc. destiné à effectuer une tâche. En général, on aura une instruction par ligne. Une instruction se termine toujours par un « ; ».

ITÉRATION

Voir boucle.

JAVA

Java est un langage de programmation orienté objet développé à l'origine par Sun Microsystems. Il a la particularité d'utiliser une machine virtuelle ce qui le rend utilisable sur de nombreuses plateformes sans nécessiter d'adaptations. Il est publié sous licence GNU GPL. Processing utilise le langage Java.

LIBRAIRIE

Une librairie est un ensemble de classes qui sont réunies dans un fichier afin d'être utilisables dans plusieurs sketchs.

LISSAGE

Le lissage est une technique qui permet d'éliminer les effets d'escaliers qui apparaissent lorsqu'on dessine des lignes diagonales ou des courbes.

LISTE

Une liste est un ensemble de données d'une certaine taille regroupées dans une même variable. Les valeurs sont placées l'une après l'autre. Chaque donnée est identifiée par un numéro, commençant par 0, qui rend son accès aisé.

MÉTHODE

Une méthode est un ensemble d'instructions regroupées dans un même bloc de code et accessibles par un simple mot clé.

MODULO

Le modulo est une opération qui permet d'obtenir le reste de la division euclidienne de deux nombres. Par exemple le reste de la division de 11 par 3 est 2 (11 / 3 = 3, **reste 2**). De manière plus générale, si on prend deux nombres x et y, le reste de la division de x par y est strictement inférieur à y. L'opérateur modulo permet de compter sans jamais dépasser un certain nombre (la base du modulo). Ainsi, vous pouvez avoir besoin de créer des animations cycliques dont la fréquence (le nombre d'animations par cycle) s'adapte en fonction d'autres éléments, par exemple la taille de la fenêtre de visualisation de l'espace de dessin. Pour déterminer à partir d'une valeur donnée ce nombre d'animations par cycle, la solution la plus simple consiste à utiliser l'opérateur modulo `%`.

OBJET

Dans le jargon informatique, un objet est une variable, qui est l'instance d'une classe. Un objet peut avoir des attributs et des méthodes. Les noms de ces attributs et méthodes sont les mêmes pour toutes les instances de cette classe, mais chaque objet peut avoir des valeurs différentes pour ses attributs. Vous n'avez rien compris ? Rassurez-vous ! Le plus simple est d'aller lire le chapitre intitulé « Les objets » du manuel.

OPENGL

OpenGL est une librairie graphique 2D/3D capable d'exploiter les ressources d'une carte graphique pour augmenter les performances. Processing peut au besoin utiliser cette technologie.

P3D

P3D est le mode de représentation 3D de base de Processing. Il est plus lent qu'OpenGL, mais plus simple d'utilisation et plus facile à faire fonctionner sur différents types de systèmes d'exploitation (Linux, OS X, Windows, etc.).

PARAMÈTRE

Les paramètres d'une méthode sont des valeurs d'entrée qu'on fournit à la méthode et qu'elle utilise en interne pour effectuer des actions.

PROGRAMME

Un programme est un ensemble d'instructions informatiques et de ressources (fichiers images, sons, textes, etc.) qui forme un tout et qui exécute une série d'actions prédéfinies dont le résultat peut être visuel, sonore, interactif, etc. Sketch est le nom donné à un programme dans Processing.

REMPLISSAGE

La couleur de remplissage est la couleur utilisée pour dessiner l'intérieur des formes géométriques. On la spécifie en appelant la méthode `fill()`.

SÉRIE

Le port série sert à connecter votre ordinateur à un certain nombre d'appareils périphériques. Votre ordinateur échange des informations avec ce matériel externe en envoyant des séries d'instructions sous forme de bytes.

SKETCH

Sketch est le nom donné à un programme dans Processing.

TABLEAU

Voir liste

TEST

Un test est une opération qui consiste à comparer deux valeurs et déterminer si elles sont égales, plus petites ou plus grandes l'une par rapport à l'autre. On utilise les tests dans les conditions et dans les boucles. On utilise les opérateurs ==, < et > pour faire des tests.

TIMER

En jargon informatique, un timer désigne un système (objet physique ou simple code informatique) qui exécute une action à intervalle régulier.

TRANSFORMATION

Une transformation géométrique est une modification de la position, de la rotation ou de l'échelle d'un élément du dessin (ligne, rectangle, cercle, etc.). Elle s'effectue avant l'opération de dessin de l'élément.

TWEEN

En animation, un *tween* désigne l'action de calculer les positions intermédiaires entre deux points d'une animation. Ce mot provient de l'anglais *in between*.

RADIAN

Le radian est une unité de mesure d'angle utilisée pour les calculs de trigonométrie. Toutes les méthodes de calcul d'angle de Processing fonctionnent avec cette unité. Un cercle complet (360°) vaut 2 x PI, soit environ 6,2831. La méthode `radians(NOMBRE)` permet de convertir un nombre de degrés en radians.

VARIABLE

Une variable est un espace réservé de la mémoire de l'ordinateur dans lequel l'utilisateur peut stocker des informations. On lui donne un nom, un type et une valeur.

40. LES AUTEURS

A propos de ce manuel
© FLOSSManuals 2010
Co-auteurs et contributeurs :
Alexandre Quessy 2010
Christian Ambaud 2010
Elisa Guerra 2010, 2011
Horia Cosmin Samoila 2010
Lionel Tardy 2010

L'animation de plusieurs objets
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Julien Gachadoat 2010
Lionel Tardy 2010

L'animation d'un objet
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Alexandre Quessy 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Lionel Tardy 2010

Arduino
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
Alexandre Quessy 2010
Elisa Guerra 2010

Horia Cosmin Samoila 2010
Julien Gachadoat 2010

Les astuces
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
Alexandre Quessy 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Jerome Saint-Clair 2010
Julien Gachadoat 2010
Lionel Tardy 2010

La documentation en ligne
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Christian Ambaud 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Lionel Tardy 2010

Credits
© adam hyde 2010
Co-auteurs et contributeurs :
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Lionel Tardy 2010

La vidéo
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
Alexandre Quessy 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Jerome Saint-Clair 2010
Julien Gachadoat 2010
Lionel Tardy 2010

Exemples d'utilisations
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Alexandre Quessy 2010
Caroline Kassimo-Zahnd 2010
Christian Ambaud 2010
Douglas Edric Stanley 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Jerome Saint-Clair 2010
Lionel Tardy 2010

L'exportation
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
Alexandre Quessy 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Julien Gachadoat 2010
Lionel Tardy 2010

Glossaire
© Elisa Guerra 2010

L'impression
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
Alexandre Quessy 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Julien Gachadoat 2010
Lionel Tardy 2010

L'installation de Processing
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Alexandre Quessy 2010

Caroline Kassimo-Zahnd 2010
Christian Ambaud 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Jerome Saint-Clair 2010
Lionel Tardy 2010

Introduction
© FLOSSManuals 2006
Co-auteurs et contributeurs :
adam hyde 2006, 2007, 2010
adama Dembele 2010
Christian Ambaud 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Jerome Saint-Clair 2010

La ligne de temps
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Alexandre Quessy 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Lionel Tardy 2010

La typographie
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Douglas Edric Stanley 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Lionel Tardy 2010

L'entrée microphone
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
Alexandre Quessy 2010
Elisa Guerra 2010

Horia Cosmin Samoila 2010
Lionel Tardy 2010

L'entrée vidéo
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
Alexandre Quessy 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Julien Gachadoat 2010
Lionel Tardy 2010

Les bases de Processing
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Alexandre Quessy 2010
Caroline Kassimo-Zahnd 2010
Christian Ambaud 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Julien Gachadoat 2010
Lionel Tardy 2010
TWikiGuest 2010

Les commentaires
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Alexandre Quessy 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Jerome Saint-Clair 2010
Lionel Tardy 2010

Les conditions
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010

Douglas Edric Stanley 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Lionel Tardy 2010

Les couleurs
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Alexandre Quessy 2010
Christian Ambaud 2010
Douglas Edric Stanley 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Jerome Saint-Clair 2010
Julien Gachadoat 2010
Lionel Tardy 2010

Les erreurs courantes
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Lionel Tardy 2010

Les événements clavier
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
Alexandre Quessy 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Julien Gachadoat 2010
Lionel Tardy 2010

Les événements souris
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
Alexandre Quessy 2010
Elisa Guerra 2010

Horia Cosmin Samoila 2010
Jerome Saint-Clair 2010
Julien Gachadoat 2010
Lionel Tardy 2010

Les formes
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Alexandre Quessy 2010
Christian Ambaud 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Jean-Francois Renaud 2010

Les images
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Alexandre Quessy 2010
Caroline Kassimo-Zahnd 2010
Douglas Edric Stanley 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010

Les listes
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Douglas Edric Stanley 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Lionel Tardy 2010

Les méthodes
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Alexandre Quessy 2010

Douglas Edric Stanley 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Lionel Tardy 2010

Les objets
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Lionel Tardy 2010

L'espace de dessin
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Alexandre Quessy 2010
Christian Ambaud 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Jerome Saint-Clair 2010

Les répétitions
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Lionel Tardy 2010

Les styles de bordures
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Lionel Tardy 2010

Les transformations
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Julien Gachadoat 2010

Les variables
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Alexandre Quessy 2010
Douglas Edric Stanley 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Lionel Tardy 2010

Le texte
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
Alexandre Quessy 2010
Douglas Edric Stanley 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Julien Gachadoat 2010
Lionel Tardy 2010

La lecture du son
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Douglas Edric Stanley 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Lionel Tardy 2010

La méthode draw
© FLOSSManuals 2010

Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Douglas Edric Stanley 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Jerome Saint-Clair 2010
Julien Gachadoat 2010
Lionel Tardy 2010

Processing dans l'enseignement
© FLOSSManuals 2010
Co-auteurs et contributeurs :
Caroline Kassimo-Zahnd 2010
Christian Ambaud 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Lionel Tardy 2010

Les librairies externes
© FLOSSManuals 2010
Co-auteurs et contributeurs :
adam hyde 2010
adama Dembele 2010
Alexandre Quessy 2010
Elisa Guerra 2010
Horia Cosmin Samoila 2010
Lionel Tardy 2010

FLOSS MANUALS

Manuels libres pour logiciels libres

41. LICENCE

Conformément à l'engagement des co-rédacteurs participant à la plateforme Flossmanuals, ce manuel sous copyright est sous licence GNU GPL : il peut notamment être lu et copié librement. La GNU GPL a été privilégiée par rapport à d'autres licences libres parce qu'elle permet, le cas échéant, l'intégration du manuel directement dans le logiciel libre qu'elle présente. Les détails d'utilisation de la licence sont mentionnés ci-après.

Avertissement important au sujet de cette traduction française.

Ceci est une traduction en français de la licence *"GNU General Public License"* (GPL). Cette traduction est fournie ici dans l'espoir qu'elle facilitera sa compréhension, mais elle ne constitue pas une traduction officielle ou approuvée d'un point de vue juridique.

La *Free Software Foundation* (FSF) ne publie pas cette traduction et ne l'a pas approuvée en tant que substitut valide au plan légal pour la licence authentique *"GNU General Public Licence"*. Cette traduction n'a pas encore été passée en revue attentivement par un juriste et donc le traducteur ne peut garantir avec certitude qu'elle représente avec exactitude la signification légale des termes de la licence authentique *"GNU General Public License"* publiée en anglais. Cette traduction n'établit donc légalement aucun des termes et conditions d'utilisation d'un logiciel sous licence GNU GPL — seul le texte original en anglais le fait. Si vous souhaitez être sûr que les activités que vous projetez seront autorisées par la *GNU General Public License*, veuillez vous référer à sa seule version anglaise authentique.

La FSF vous recommande fermement de ne pas utiliser cette traduction en tant que termes officiels pour vos propres programmes ; veuillez plutôt utiliser la version anglaise authentique telle que publiée par la FSF. Si vous choisissez d'acheminer cette traduction en même temps qu'un Programme sous licence GNU GPL, cela ne vous dispense pas de l'obligation d'acheminer en même temps une copie de la licence authentique en anglais, et de conserver dans la traduction cet avertissement important en français et son équivalent en anglais ci-dessous.

Important Warning About This French Translation.

This is a translation of the GNU General Public License (GPL) into French. This translation is distributed in the hope that it will facilitate understanding, but it is not an official or legally approved translation.

The Free Software Foundation (FSF) is not the publisher of this translation and has not approved it as a legal substitute for the authentic GNU General Public License. The translation has not been reviewed carefully by lawyers, and therefore the translator cannot be sure that it exactly represents the legal meaning of the authentic GNU General Public License published in English. This translation does not legally state the terms and conditions of use of any Program licenced under GNU GPL — only the original English text of the GNU LGPL does that. If you wish to be sure whether your planned activities are permitted by the GNU General Public License, please refer to its sole authentic English version.

The FSF strongly urges you not to use this translation as the official distribution terms for your programs; instead, please use the authentic English version published by the FSF. If you choose to convey this translation along with a Program covered by the GPL Licence, this does not remove your obligation to convey at the same time a copy of the authentic GNU GPL License in English, and you must keep in this translation this important warning in English and its equivalent in French above.

Préambule

La Licence Publique Générale GNU (*"GNU General Public License"*) est une licence libre, en *"copyleft"*, destinée aux œuvres logicielles et d'autres types de travaux.

Les licences de la plupart des œuvres logicielles et autres travaux de pratique sont conçues pour ôter votre liberté de partager et modifier ces travaux. En contraste, la Licence Publique Générale GNU a pour but de garantir votre liberté de partager et changer toutes les versions d'un programme — afin d'assurer qu'il restera libre pour tous les utilisateurs. Nous, la *Free Software Foundation*, utilisons la Licence Publique Générale GNU pour la plupart de nos logiciels ; cela s'applique aussi à tout autre travail édité de cette façon par ses auteurs. Vous pouvez, vous aussi, l'appliquer à vos propres programmes.

Quand nous parlons de logiciel libre (*"free"*), nous nous référons à la liberté (*"freedom"*), pas au prix. Nos Licences Publiques Générales sont conçues pour assurer que vous ayez la liberté de distribuer des copies de logiciel libre (et le facturer si vous le souhaitez), que vous receviez le code source ou pouviez l'obtenir si vous le voulez, que vous pouviez modifier le logiciel ou en utiliser toute partie dans de nouveaux logiciels libres, et que vous sachiez que vous avez le droit de faire tout ceci.

Pour protéger vos droits, nous avons besoin d'empêcher que d'autres vous restreignent ces droits ou vous demandent de leur abandonner ces droits. En

conséquence, vous avez certaines responsabilités si vous distribuez des copies d'un tel programme ou si vous le modifiez : les responsabilités de respecter la liberté des autres.

Par exemple, si vous distribuez des copies d'un tel programme, que ce soit gratuit ou contre un paiement, vous devez accorder aux Destinataires les mêmes libertés que vous avez reçues. Vous devez aussi vous assurer qu'eux aussi reçoivent ou peuvent recevoir son code source. Et vous devez leur montrer les termes de cette licence afin qu'ils connaissent leurs droits.

Les développeurs qui utilisent la GPL GNU protègent vos droits en deux étapes : (1) ils affirment leurs droits d'auteur (*"copyright"*) sur le logiciel, et (2) vous accordent cette Licence qui vous donne la permission légale de le copier, le distribuer et/ou le modifier.

Pour la protection des développeurs et auteurs, la GPL stipule clairement qu'il n'y a pas de garantie pour ce logiciel libre. Aux fins à la fois des utilisateurs et auteurs, la GPL requiert que les versions modifiées soient marquées comme changées, afin que leurs problèmes ne soient pas attribués de façon erronée aux auteurs des versions précédentes.

Certains dispositifs sont conçus pour empêcher l'accès des utilisateurs à l'installation ou l'exécution de versions modifiées du logiciel à l'intérieur de ces dispositifs, alors que les fabricants le peuvent. Ceci est fondamentalement incompatible avec le but de protéger la liberté des utilisateurs de modifier le logiciel. L'aspect systématique de tels abus se produit dans le secteur des produits destinés aux utilisateurs individuels, ce qui est précisément ce qui est le plus inacceptable. Aussi, nous avons conçu cette version de la GPL pour prohiber cette pratique pour ces produits. Si de tels problèmes surviennent dans d'autres domaines, nous nous tenons prêts à étendre cette restriction à ces domaines dans de futures versions de la GPL, autant qu'il sera nécessaire pour protéger la liberté des utilisateurs.

Finalement, chaque programme est constamment menacé par les brevets logiciels. Les États ne devraient pas autoriser de tels brevets à restreindre le développement et l'utilisation de logiciels libres sur des ordinateurs d'usage général ; mais dans ceux qui le font, nous voulons spécialement éviter le danger que les brevets appliqués à un programme libre puissent le rendre effectivement propriétaire. Pour empêcher ceci, la GPL assure que les brevets ne peuvent être utilisés pour rendre le programme non-libre.

Les termes précis et conditions concernant la copie, la distribution et la modification suivent.

Termes et conditions

Article 0. Définitions.

« Cette Licence » se réfère à la version 3 de la *"GNU General Public License"* (le texte original en anglais).

« Droit d'Auteur » signifie aussi les droits du "copyright" ou voisins qui s'appliquent à d'autres types de travaux, tels que ceux sur les masques de semi-conducteurs.

« Le Programme » se réfère à tout travail qui peut être sujet au Droit d'Auteur ("copyright") et dont les droits d'utilisation sont concédés en vertu de cette Licence. Chacun des Licenciés, à qui cette Licence est concédée, est désigné par « vous. » Les « Licenciés » et les « Destinataires » peuvent être des personnes physiques ou morales (individus ou organisations).

« Modifier » un travail signifie en obtenir une copie et adapter tout ou partie du travail d'une façon nécessitant une autorisation d'un titulaire de Droit d'Auteur, autre que celle permettant d'en produire une copie conforme. Le travail résultant est appelé une « version modifiée » du précédent travail, ou un travail « basé sur » le précédent travail.

Un « Travail Couvert » signifie soit le Programme non modifié soit un travail basé sur le Programme.

« Propager » un travail signifie faire quoi que ce soit avec lui qui, sans permission, vous rendrait directement ou indirectement responsable d'un délit de contrefaçon suivant les lois relatives au Droit d'Auteur, à l'exception de son exécution sur un ordinateur ou de la modification d'une copie privée. La propagation inclut la copie, la distribution (avec ou sans modification), la mise à disposition envers le public, et aussi d'autres activités dans certains pays.

« Acheminer » un travail signifie tout moyen de propagation de celui-ci qui permet à d'autres parties de réaliser ou recevoir des copies. La simple interaction d'un utilisateur à travers un réseau informatique, sans transfert effectif d'une copie, ne constitue pas un acheminement.

Une interface utilisateur interactive affiche des « Notices Légales Appropriées » quand elle comprend un dispositif convenable, bien visible et évident qui (1) affiche une notice appropriée sur les droits d'auteur et (2) informe l'utilisateur qu'il n'y a pas de garantie pour le travail (sauf si des garanties ont été fournies hors du cadre de cette Licence), que les licenciés peuvent acheminer le travail sous cette Licence, et comment voir une copie de cette Licence. Si l'interface présente une liste de commandes utilisateur ou d'options, tel qu'un menu, un élément évident dans la liste présentée remplit ce critère.

Article 1. Code source.

Le « code source » d'un travail signifie la forme préférée du travail permettant ou facilitant les modifications de celui-ci. Le « code objet » d'un travail signifie toute forme du travail qui n'en est pas le code source.

Une « Interface Standard » signifie une interface qui est soit celle d'une norme officielle définie par un organisme de normalisation reconnu ou, dans le cas des interfaces spécifiées pour un langage de programmation particulier, une interface largement utilisée parmi les développeurs travaillant dans ce langage.

Les « Bibliothèques Système » d'un travail exécutable incluent tout ce qui, en dehors du travail dans son ensemble, (a) est inclus dans la forme usuelle de paquetage d'un Composant Majeur mais ne fait pas partie de ce Composant Majeur et (b) sert seulement à permettre l'utilisation du travail avec ce Composant Majeur ou à implémenter une Interface Standard pour laquelle une implémentation est disponible au public sous forme de code source ; un « Composant Majeur » signifie, dans ce contexte, un composant majeur essentiel (noyau, système de fenêtrage, etc.) du système d'exploitation (le cas échéant) d'un système sur lequel le travail exécutable fonctionne, ou bien un compilateur utilisé pour produire le code objet du travail, ou un interprète de code objet utilisé pour exécuter celui-ci.

Le « Source Correspondant » d'un travail sous forme de code objet signifie l'ensemble des codes sources nécessaires pour générer, installer et (dans le cas d'un travail exécutable) exécuter le code objet et modifier le travail, y compris les scripts pour contrôler ces activités. Cependant, cela n'inclut pas les Bibliothèques Système du travail, ni les outils d'usage général ou les programmes libres généralement disponibles qui peuvent être utilisés sans modification pour achever ces activités mais ne sont pas partie de ce travail. Par exemple le Source Correspondant inclut les fichiers de définition d'interfaces associés aux fichiers sources du travail, et le code source des bibliothèques partagées et des sous-routines liées dynamiquement, pour lesquelles le travail est spécifiquement conçu pour les requérir via, par exemple, des communications de données ou contrôles de flux internes entre ces sous-programmes et d'autres parties du travail.

Le Source Correspondant n'a pas besoin d'inclure tout ce que les utilisateurs peuvent regénérer automatiquement à partir d'autres parties du Source Correspondant.

Le Source Correspondant pour un travail sous forme de code source est ce même travail.

Article 2. Permissions de base.

Tous les droits accordés suivant cette Licence le sont jusqu'au terme des Droits d'Auteur ("copyright") sur le Programme, et sont irrévocables pourvu que les conditions établies soient remplies. Cette Licence affirme explicitement votre permission illimitée d'exécuter le Programme non modifié. La sortie produite par l'exécution d'un Travail Couvert n'est couverte par cette Licence que si cette sortie, étant donné leur contenu, constitue un Travail Couvert. Cette Licence reconnait vos propres droits d'usage raisonnable ("fair use" en législation des États-Unis d'Amérique) ou autres équivalents, tels qu'ils sont pourvus par la loi applicable sur le Droit d'Auteur ("copyright").

Vous pouvez créer, exécuter et propager sans condition des Travaux Couverts que vous n'acheminez pas, aussi longtemps que votre licence demeure en vigueur. Vous pouvez acheminer des Travaux Couverts à d'autres personnes dans le seul but de leur faire réaliser des modifications à votre usage exclusif, ou pour qu'ils vous fournissent des facilités vous permettant d'exécuter ces travaux, pourvu que vous vous conformiez aux termes de cette Licence lors de l'acheminement de tout matériel dont vous ne contrôlez pas le Droit d'Auteur ("copyright"). Ceux qui, dès lors, réalisent ou exécutent pour vous les Travaux Couverts ne doivent alors le faire qu'exclusivement pour votre propre compte, sous votre direction et votre contrôle, suivant des termes qui leur interdisent de réaliser, en dehors de leurs relations avec vous, toute copie de votre matériel soumis au Droit d'Auteur.

L'acheminement dans toutes les autres circonstances n'est permis que selon les conditions établies ci-dessous. La concession de sous-licences n'est pas autorisée ; l'article 10 rend cet usage non nécessaire.

Article 3. Protection des droits légaux des utilisateurs envers les lois anti-contournement.

Aucun Travail Couvert ne doit être vu comme faisant partie d'une mesure technologique effective selon toute loi applicable remplissant les obligations prévues à l'article 11 du traité international sur le droit d'auteur adopté à l'OMPI le 20 décembre 1996, ou toutes lois similaires qui prohibent ou restreignent le contournement de telles mesures.

Si vous acheminez un Travail Couvert, vous renoncez à tout pouvoir légal d'interdire le contournement des mesures technologiques dans tous les cas où un tel contournement serait effectué en exerçant les droits prévus dans cette Licence pour ce Travail Couvert, et vous déclarez rejeter toute intention de limiter l'opération ou la modification du Travail, en tant que moyens de renforcer, à l'encontre des utilisateurs de ce Travail, vos droits légaux ou ceux de tierces parties d'interdire le contournement des mesures technologiques.

Article 4. Acheminement des copies conformes.

Vous pouvez acheminer des copies conformes du code source du Programme tel que vous l'avez reçu, sur n'importe quel support, pourvu que vous publiiez scrupuleusement et de façon appropriée sur chaque copie une notice de Droit d'Auteur appropriée ; gardez intactes toutes les notices établissant que cette Licence et tous les termes additionnels non permissifs ajoutés en accord avec l'article 7 s'appliquent à ce code ; et donnez à chacun des Destinataires une copie de cette Licence en même temps que le Programme.

Vous pouvez facturer un prix quelconque, y compris gratuit, pour chacune des copies que vous acheminez, et vous pouvez offrir une protection additionnelle de support ou de garantie en échange d'un paiement.

Article 5. Acheminement des versions sources modifiées.

Vous pouvez acheminer un travail basé sur le Programme, ou bien les modifications pour le produire à partir du Programme, sous la forme de code source suivant les termes de l'article 4, pourvu que vous satisfassiez aussi à chacune des conditions requises suivantes :

- a) Le travail doit comporter des notices évidentes établissant que vous l'avez modifié et donnant la date correspondante.
- b) Le travail doit comporter des notices évidentes établissant qu'il est édité selon cette Licence et les conditions ajoutées d'après l'article 7. Cette obligation vient modifier l'obligation de l'article 4 de « garder intactes toutes les notices. »
- c) Vous devez licencier le travail entier, comme un tout, suivant cette Licence à quiconque entre en possession d'une copie. Cette Licence s'appliquera en conséquence, avec les termes additionnels applicables prévus par l'article 7, à la totalité du travail et chacune de ses parties, indépendamment de la façon dont ils sont empaquetés. Cette licence ne donne aucune permission de licencier le travail d'une autre façon, mais elle n'invalide pas une telle permission si vous l'avez reçue séparément.
- d) Si le travail a des interfaces utilisateurs interactives, chacune doit afficher les Notices Légales Appropriées ; cependant si le Programme a des interfaces qui n'affichent pas les Notices Légales Appropriées, votre travail n'a pas à les modifier pour qu'elles les affichent.

Une compilation d'un Travail Couvert avec d'autres travaux séparés et indépendants, qui ne sont pas par leur nature des extensions du Travail Couvert, et qui ne sont pas combinés avec lui de façon à former un programme plus large, dans ou sur un volume de stockage ou un support de distribution, est appelée un « agrégat » si la compilation et son Droit d'Auteur résultant ne sont pas utilisés pour limiter l'accès ou les droits légaux des utilisateurs de la compilation en deçà de ce que permettent les travaux individuels. L'inclusion d'un Travail Couvert dans un agrégat ne cause pas l'application de cette Licence aux autres parties de l'agrégat.

Article 6. Acheminement des formes non sources.

Vous pouvez acheminer sous forme de code objet un Travail Couvert suivant les termes des articles 4 et 5, pourvu que vous acheminiez également suivant les termes de cette Licence le Source Correspondant lisible par une machine, d'une des façons suivantes :

- a) Acheminer le code objet sur, ou inclus dans, un produit physique (y compris un support de distribution physique), accompagné par le Source Correspondant fixé sur un support physique durable habituellement utilisé pour les échanges de logiciels.
- b) Acheminer le code objet sur, ou inclus dans, un produit physique (y compris un support de distribution physique), accompagné d'une offre écrite, valide pour au moins trois années et valide pour aussi longtemps que vous fournissez des pièces de rechange ou un support client pour ce modèle de produit, afin de donner à quiconque possède le code objet soit (1) une copie du Source Correspondant à tout logiciel dans ce produit qui est couvert par cette Licence, sur un support physique durable habituellement utilisé pour les échanges de logiciels, pour un prix non supérieur au coût raisonnable de la réalisation physique de l'acheminement de la source, ou soit (2) un accès permettant de copier le Source Correspondant depuis un serveur réseau sans frais.
- c) Acheminer des copies individuelles du code objet avec une copie de l'offre écrite de fournir le Source Correspondant. Cette alternative est permise seulement occasionnellement et non commercialement, et seulement si vous avez reçu le code objet avec une telle offre, en accord avec l'article 6 alinéa b.
- d) Acheminer le code objet en offrant un accès depuis un emplacement désigné (gratuit ou contre facturation) et offrir un accès équivalent au Source Correspondant de la même façon via le même emplacement et sans facturation supplémentaire. Vous n'avez pas besoin d'obliger les Destinataires à copier le Source Correspondant en même temps que le code objet. Si l'emplacement pour copier le code objet est un serveur réseau, le Source Correspondant peut être sur un serveur différent (opéré par vous ou par un tiers) qui supporte des facilités équivalentes de copie, pourvu que vous mainteniez des directions claires à proximité du code objet indiquant où trouver le Source Correspondant. Indépendamment de quel serveur héberge le Source Correspondant, vous restez obligé de vous assurer qu'il reste disponible aussi longtemps que nécessaire pour satisfaire à ces obligations.
- e) Acheminer le code objet en utilisant une transmission d'égal à égal, pourvu que vous informiez les autres participants sur où le code objet et le Source Correspondant du travail sont offerts sans frais au public général suivant l'article 6 alinéa d.

Une portion séparable du code objet, dont le code source est exclu du Source Correspondant en tant que Bibliothèque Système, n'a pas besoin d'être incluse dans l'acheminement du travail sous forme de code objet.

Un « Produit Utilisateur » est soit (1) un « Produit de Consommation, » ce qui signifie toute propriété personnelle tangible normalement utilisée à des fins personnelles, familiales ou relatives au foyer, soit (2) toute chose conçue ou vendue pour l'incorporation dans un lieu d'habitation. Pour déterminer si un produit constitue un Produit de Consommation, les cas ambigus sont résolus en fonction de la couverture. Pour un produit particulier reçu par un utilisateur particulier, l'expression « normalement utilisée » ci-devant se réfère à une utilisation typique ou l'usage commun de produits de même catégorie, indépendamment du statut de cet utilisateur particulier ou de la façon spécifique dont cet utilisateur particulier utilise effectivement ou s'attend lui-même ou est attendu à utiliser ce produit. Un produit est un Produit de Consommation indépendamment du fait que ce produit a ou n'a pas d'utilisations substantielles commerciales, industrielles ou hors Consommation, à moins que de telles utilisations représentent le seul mode significatif d'utilisation du produit.

Les « Informations d'Installation » d'un Produit Utilisateur signifient toutes les méthodes, procédures, clés d'autorisation ou autres informations requises pour installer et exécuter des versions modifiées d'un Travail Couvert dans ce Produit Utilisateur à partir d'une version modifiée de son Source Correspondant. Les informations qui suffisent à assurer la continuité de fonctionnement du code objet modifié ne doivent en aucun cas être empêchées ou interférées du seul fait qu'une modification a été effectuée.

Si vous acheminez le code objet d'un Travail Couvert dans, ou avec, ou spécifiquement pour l'utilisation dans, un Produit Utilisateur et l'acheminement se produit en tant qu'élément d'une transaction dans laquelle le droit de possession et d'utilisation du Produit Utilisateur est transféré au Destinataire définitivement ou pour un terme fixé (indépendamment de la façon dont la transaction est caractérisée), le Source Correspondant acheminé selon cet article-ci doit être accompagné des Informations d'Installation. Mais cette obligation ne s'applique pas si ni vous ni aucune tierce partie ne détient la possibilité d'installer un code objet modifié sur le Produit Utilisateur (par exemple, le travail a été installé en mémoire morte).

L'obligation de fournir les Informations d'Installation n'inclut pas celle de continuer à fournir un service de support, une garantie ou des mises à jour pour un travail qui a été modifié ou installé par le Destinataire, ou pour le Produit Utilisateur dans lequel il a été modifié ou installé. L'accès à un réseau peut être rejeté quand la modification elle-même affecte matériellement et défavorablement les opérations du réseau ou viole les règles et protocoles de communication au travers du réseau.

Le Source Correspondant acheminé et les Informations d'Installation fournies, en accord avec cet article, doivent être dans un format publiquement documenté (et dont une implémentation est disponible auprès du public sous forme de code source) et ne doivent nécessiter aucune clé ou aucun mot de passe spécial pour le dépaquetage, la lecture ou la copie.

Article 7. Termes additionnels.

Les « permissions additionnelles » désignent les termes qui supplémentent ceux de cette Licence en émettant des exceptions à l'une ou plusieurs de ses conditions. Les permissions additionnelles qui sont applicables au Programme entier doivent être traitées comme si elles étaient incluses dans cette Licence, dans les limites de leur validité suivant la loi applicable. Si des permissions additionnelles s'appliquent seulement à une partie du Programme, cette partie peut être utilisée séparément suivant ces permissions, mais le Programme tout entier reste gouverné par cette Licence sans regard aux permissions additionnelles.

Quand vous acheminez une copie d'un Travail Couvert, vous pouvez à votre convenance ôter toute permission additionnelle de cette copie, ou de n'importe quelle partie de celui-ci. (Des permissions additionnelles peuvent être rédigées de façon à requérir leur propre suppression dans certains cas où vous modifiez le travail.) Vous pouvez placer les permissions additionnelles sur le matériel acheminé, ajoutées par vous à un Travail Couvert pour lequel vous avez ou pouvez donner les permissions de Droit d'Auteur ("copyright") appropriées.

Nonobstant toute autre clause de cette Licence, pour tout constituant que vous ajoutez à un Travail Couvert, vous pouvez (si autorisé par les titulaires de Droit d'Auteur pour ce constituant) supplémenter les termes de cette Licence avec des termes :

- a) qui rejettent la garantie ou limitent la responsabilité de façon différente des termes des articles 15 et 16 de cette Licence ; ou
- b) qui requièrent la préservation de notices légales raisonnables spécifiées ou les attributions d'auteur dans ce constituant ou dans les Notices Légales Appropriées affichées par les travaux qui le contiennent ; ou
- c) qui prohibent la représentation incorrecte de l'origine de ce constituant, ou qui requièrent que les versions modifiées d'un tel constituant soient marquées par des moyens raisonnables comme différentes de la version originale ; ou
- d) qui limitent l'usage à but publicitaire des noms des concédants de licence et des auteurs du constituant ; ou
- e) qui refusent à accorder des droits selon la législation relative aux marques commerciales, pour l'utilisation dans des noms commerciaux, marques commerciales ou marques de services ; ou
- f) qui requièrent l'indemnisation des concédants de licences et auteurs du constituant par quiconque achemine ce constituant (ou des versions modifiées de celui-ci) en assumant contractuellement la responsabilité envers le Destinataire, pour toute responsabilité que ces engagements contractuels imposent directement à ces octroyants de licences et auteurs.

Tous les autres termes additionnels non permissifs sont considérés comme des « restrictions avancées » dans le sens de l'article 10. Si le Programme tel que vous l'avez reçu, ou toute partie de celui-ci, contient une notice établissant qu'il est gouverné par cette Licence en même temps qu'un terme qui est une restriction avancée, vous pouvez ôter ce terme. Si un document de licence contient une restriction avancée,

mais permet la reconcession de licence ou l'acheminement suivant cette Licence, vous pouvez ajouter un Travail Couvert constituant gouverné par les termes de ce document de licence, pourvu que la restriction avancée ne survive pas à une telle cession de licence ou acheminement.

Si vous ajoutez des termes à un Travail Couvert en accord avec cet article, vous devez placer, dans les fichiers sources appropriés, une déclaration des termes additionnels qui s'appliquent à ces fichiers, ou une notice indiquant où trouver les termes applicables.

Les termes additionnels, qu'ils soient permissifs ou non permissifs, peuvent être établis sous la forme d'une licence écrite séparément, ou établis comme des exceptions ; les obligations ci-dessus s'appliquent dans chacun de ces cas.

Article 8. Terminaison.

Vous ne pouvez ni propager ni modifier un Travail Couvert autrement que suivant les termes de cette Licence. Toute autre tentative de le propager ou le modifier est nulle et terminera automatiquement vos droits selon cette Licence (y compris toute licence de brevet accordée selon le troisième paragraphe de l'article 11).

Cependant, si vous cessez toute violation de cette Licence, alors votre licence depuis un titulaire de Droit d'Auteur ("copyright") est réinstaurée (a) à titre provisoire à moins que et jusqu'à ce que le titulaire de Droit d'Auteur termine finalement et explicitement votre licence, et (b) de façon permanente si le titulaire de Droit d'Auteur ne parvient pas à vous notifier de la violation par quelque moyen raisonnable dans les soixante (60) jours après la cessation.

De plus, votre licence depuis un titulaire particulier de Droit d'Auteur est réinstaurée de façon permanente si ce titulaire vous notifie de la violation par quelque moyen raisonnable, c'est la première fois que vous avez reçu une notification deviolation de cette Licence (pour un travail quelconque) depuis ce titulaire de Droit d'Auteur, et vous résolvez la violation dans les trente (30) jours qui suivent votre réception de la notification.

La terminaison de vos droits suivant cette section ne terminera pas les licences des parties qui ont reçu des copies ou droits de votre part suivant cette Licence. Si vos droits ont été terminés et non réinstaurés de façon permanente, vous n'êtes plus qualifié à recevoir de nouvelles licences pour les mêmes constituants selon l'article 10.

Article 9. Acceptation non requise pour obtenir des copies.

Vous n'êtes pas obligé d'accepter cette licence afin de recevoir ou exécuter une copie du Programme. La propagation asservie d'un Travail Couvert qui se produit simplement en conséquence d'une transmission d'égal à égal pour recevoir une copie ne nécessite pas l'acceptation. Cependant, rien d'autre que cette Licence ne vous accorde la permission de propager ou modifier un quelconque Travail Couvert. Ces actions enfreignent le Droit d'Auteur si vous n'acceptez pas cette Licence. Par conséquent, en modifiant ou propageant un Travail Couvert, vous indiquez votre acceptation de cette Licence pour agir ainsi.

Article 10. Cession automatique de Licence aux Destinataires et intermédiaires.

Chaque fois que vous acheminez un Travail Couvert, le Destinataire reçoit automatiquement une licence depuis les concédants originaux, pour exécuter, modifier et propager ce travail, suivant les termes de cette Licence. Vous n'êtes pas responsable du renforcement de la conformation des tierces parties avec cette Licence.

Une « transaction d'entité » désigne une transaction qui transfère le contrôle d'une organisation, ou de substantiellement tous ses actifs, ou la subdivision d'une organisation, ou la fusion de plusieurs organisations. Si la propagation d'un Travail Couvert résulte d'une transaction d'entité, chaque partie à cette transaction qui reçoit une copie du travail reçoit aussi les licences pour le travail que le prédécesseur intéressé à cette partie avait ou pourrait donner selon le paragraphe précédent, plus un droit de possession du Source Correspondant de ce travail depuis le prédécesseur intéressé si ce prédécesseur en dispose ou peut l'obtenir par des efforts raisonnables.

Vous ne pouvez imposer aucune restriction avancée dans l'exercice des droits accordés ou affirmés selon cette Licence. Par exemple, vous ne pouvez imposer aucun paiement pour la licence, aucune royaltie, ni aucune autre charge pour l'exercice des droits accordés selon cette Licence ; et vous ne pouvez amorcer aucun litige judiciaire (y compris une réclamation croisée ou contre-réclamation dans un procès) sur l'allégation qu'une revendication de brevet est enfreinte par la réalisation, l'utilisation, la vente, l'offre de vente, ou l'importation du Programme ou d'une quelconque portion de celui-ci.

Article 11. Brevets.

Un « contributeur » est un titulaire de Droit d'Auteur (*"copyright"*) qui autorise l'utilisation selon cette Licence du Programme ou du travail sur lequel le Programme est basé. Le travail ainsi soumis à licence est appelé la « version contributive » de ce contributeur.

Les « revendications de brevet essentielles » sont toutes les revendications de brevets détenues ou contrôlées par le contributeur, qu'elles soient déjà acquises par lui ou

acquises subséquemment, qui pourraient être enfreintes de quelque manière, permises par cette Licence, sur la réalisation, l'utilisation ou la vente de la version contributive de celui-ci. Aux fins de cette définition, le « contrôle » inclue le droit de concéder des sous-licences de brevets d'une manière consistante, nécessaire et suffisante, avec les obligations de cette Licence.

Chaque contributeur vous accorde une licence de brevet non exclusive, mondiale et libre de toute royaltie, selon les revendications de brevet essentielles, pour réaliser, utiliser, vendre, offrir à la vente, importer et autrement exécuter, modifier et propager les contenus de sa version contributive.

Dans les trois paragraphes suivants, une « licence de brevet » désigne tous les accords ou engagements exprimés, quel que soit le nom que vous lui donnez, de ne pas mettre en vigueur un brevet (telle qu'une permission explicite pour mettre en pratique un brevet, ou un accord pour ne pas poursuivre un Destinataire pour cause de violation de brevet). « Accorder » une telle licence de brevet à une partie signifie conclure un tel accord ou engagement à ne pas faire appliquer le brevet à cette partie.

Si vous acheminez un Travail Couvert, dépendant en connaissance d'une licence de brevet, et si le Source Correspondant du travail n'est pas disponible à quiconque copie, sans frais et suivant les termes de cette Licence, à travers un serveur réseau publiquement accessible ou tout autre moyen immédiatement accessible, alors vous devez soit (1) rendre la Source Correspondante ainsi disponible, soit (2) vous engager à vous priver pour vous-même du bénéfice de la licence de brevet pour ce travail particulier, soit (3) vous engager, d'une façon consistante avec les obligations de cette Licence, à étendre la licence de brevet aux Destinataires de ce travail. « Dépendant en connaissance » signifie que vous avez effectivement connaissance que, selon la licence de brevet, votre acheminement du Travail Couvert dans un pays, ou l'utilisation du Travail Couvert par votre Destinataire dans un pays, enfreindrait un ou plusieurs brevets identifiables dans ce pays où vous avez des raisons de penser qu'ils sont valides.

Si, conformément à ou en liaison avec une même transaction ou un même arrangement, vous acheminez, ou propagez en procurant un acheminement de, un Travail Couvert et accordez une licence de brevet à l'une des parties recevant le Travail Couvert pour lui permettre d'utiliser, propager, modifier ou acheminer une copie spécifique du Travail Couvert, alors votre accord est automatiquement étendu à tous les Destinataires du Travail Couvert et des travaux basés sur celui-ci.

Une licence de brevet est « discriminatoire » si, dans le champ de sa couverture, elle n'inclut pas un ou plusieurs des droits qui sont spécifiquement accordés selon cette Licence, ou en prohibe l'exercice, ou est conditionnée par le non-exercice d'un ou plusieurs de ces droits. Vous ne pouvez pas acheminer un Travail Couvert si vous êtes partie à un arrangement selon lequel une partie tierce exerçant son activité dans la distribution de logiciels et à laquelle vous effectuez un paiement fondé sur l'étendue de votre activité d'acheminement du travail, et selon lequel la partie tierce accorde, à

une quelconque partie qui recevrait depuis vous le Travail Couvert, une licence de brevet discriminatoire (a) en relation avec les copies du Travail Couvert acheminées par vous (ou les copies réalisées à partir de ces copies), ou (b) avant tout destinée et en relation avec des produits spécifiques ou compilations contenant le Travail Couvert, à moins que vous ayez conclu cet arrangement ou que la licence de brevet ait été accordée avant le 28 mars 2007.

Rien dans cette Licence ne devrait être interprété comme devant exclure ou limiter toute licence implicite ou d'autres moyens de défense à une infraction qui vous seraient autrement disponible selon la loi applicable relative aux brevets.

Article 12. Non abandon de la liberté des autres.

Si des conditions vous sont imposées (que ce soit par décision judiciaire, par un accord ou autrement) qui contredisent les conditions de cette Licence, elles ne vous excusent pas des conditions de cette Licence. Si vous ne pouvez pas acheminer un Travail Couvert de façon à satisfaire simultanément vos obligations suivant cette Licence et toutes autres obligations pertinentes, alors en conséquence vous ne pouvez pas du tout l'acheminer. Par exemple, si vous avez un accord sur des termes qui vous obligent à collecter pour le réacheminement des royalties depuis ceux à qui vous acheminez le Programme, la seule façon qui puisse vous permettre de satisfaire à la fois à ces termes et ceux de cette Licence sera de vous abstenir entièrement d'acheminer le Programme.

Article 13. Utilisation avec la Licence Générale Publique Affero GNU.

Nonobstant toute autre clause de cette Licence, vous avez la permission de lier ou combiner tout Travail Couvert avec un travail placé sous la version 3 de la Licence Générale Publique GNU Affero (*"GNU Affero General Public License"*) en un seul travail combiné, et d'acheminer le travail résultant. Les termes de cette Licence continueront à s'appliquer à la partie formant un Travail Couvert, mais les obligations spéciales de la Licence Générale Publique GNU Affero, article 13, concernant l'interaction à travers un réseau s'appliqueront à la combinaison en tant que telle.

Article 14. Versions révisées de cette License.

La *Free Software Foundation* peut publier des versions révisées et/ou nouvelles de la Licence Publique Générale GNU (*"GNU General Public License"*) de temps en temps. De telles versions nouvelles resteront similaires dans l'esprit avec la présente version, mais peuvent différer dans le détail afin de traiter de nouveaux problèmes ou préoccupations.

Chaque version reçoit un numéro de version distinctif. Si le Programme indique qu'une version spécifique de la Licence Publique Générale GNU « ou toute version ultérieure » (*"or any later version"*) s'applique à celui-ci, vous avez le choix de suivre soit les termes et conditions de cette version numérotée, soit ceux de n'importe quelle version

publiée ultérieurement par la *Free Software Foundation*. Si le Programme n'indique pas une version spécifique de la Licence Publique Générale GNU, vous pouvez choisir l'une quelconque des versions qui ont été publiées par la *Free Software Foundation*.

Si le Programme spécifie qu'un intermédiaire peut décider quelles versions futures de la Licence Générale Publique GNU peuvent être utilisées, la déclaration publique d'acceptation d'une version par cet intermédiaire vous autorise à choisir cette version pour le Programme.

Des versions ultérieures de la licence peuvent vous donner des permissions additionnelles ou différentes. Cependant aucune obligation additionnelle n'est imposée à l'un des auteurs ou titulaires de Droit d'Auteur du fait de votre choix de suivre une version ultérieure.

Article 15. Déclaration d'absence de garantie.

IL N'Y A AUCUNE GARANTIE POUR LE PROGRAMME, DANS LES LIMITES PERMISES PAR LA LOI APPLICABLE. À MOINS QUE CELA NE SOIT ÉTABLI DIFFÉREMMENT PAR ÉCRIT, LES PROPRIÉTAIRES DE DROITS ET/OU LES AUTRES PARTIES FOURNISSENT LE PROGRAMME « EN L'ÉTAT » SANS GARANTIE D'AUCUNE SORTE, QU'ELLE SOIT EXPRIMÉE OU IMPLICITE, CECI COMPRENANT, SANS SE LIMITER À CELLES-CI, LES GARANTIES IMPLICITES DE COMMERCIALISABILITÉ ET D'ADÉQUATION À UN OBJECTIF PARTICULIER. VOUS ASSUMEZ LE RISQUE ENTIER CONCERNANT LA QUALITÉ ET LES PERFORMANCES DU PROGRAMME. DANS L'ÉVENTUALITÉ OÙ LE PROGRAMME S'AVÉRERAIT DÉFECTUEUX, VOUS ASSUMEZ LES COÛTS DE TOUS LES SERVICES, RÉPARATIONS OU CORRECTIONS NÉCESSAIRES.

Article 16. Limitation de responsabilité.

EN AUCUNE AUTRE CIRCONSTANCE QUE CELLES REQUISES PAR LA LOI APPLICABLE OU ACCORDÉES PAR ÉCRIT, UN TITULAIRE DE DROITS SUR LE PROGRAMME, OU TOUT AUTRE PARTIE QUI MODIFIE OU ACHEMINE LE PROGRAMME COMME PERMIS CI-DESSUS, NE PEUT ÊTRE TENU POUR RESPONSABLE ENVERS VOUS POUR LES DOMMAGES, INCLUANT TOUT DOMMAGE GÉNÉRAL, SPÉCIAL, ACCIDENTEL OU INDUIT SURVENANT PAR SUITE DE L'UTILISATION OU DE L'INCAPACITÉ D'UTILISER LE PROGRAMME (Y COMPRIS, SANS SE LIMITER À CELLES-CI, LA PERTE DE DONNÉES OU L'INEXACTITUDE DES DONNÉES RETOURNÉES OU LES PERTES SUBIES PAR VOUS OU DES PARTIES TIERCES OU L'INCAPACITÉ DU PROGRAMME À FONCTIONNER AVEC TOUT AUTRE PROGRAMME), MÊME SI UN TEL TITULAIRE OU TOUTE AUTRE PARTIE A ÉTÉ AVISÉ DE LA POSSIBILITÉ DE TELS DOMMAGES.

Article 17. Interprétation des sections 15 et 16.

Si la déclaration d'absence de garantie et la limitation de responsabilité fournies ci-dessus ne peuvent prendre effet localement selon leurs termes, les cours de justice qui les examinent doivent appliquer la législation locale qui approche au plus près possible une levée absolue de toute responsabilité civile liée au Programme, à moins qu'une garantie ou assumation de responsabilité accompagne une copie du Programme en échange d'un paiement.

FIN DES TERMES ET CONDITIONS.

Comment appliquer ces termes à vos nouveaux programmes

Si vous développez un nouveau programme et voulez qu'il soit le plus possible utilisable par le public, la meilleure façon d'y parvenir et d'en faire un logiciel libre que chacun peut redistribuer et changer suivant ces termes-ci.

Pour appliquer ces termes, attachez les notices suivantes au programme. Il est plus sûr de les attacher au début de chacun des fichiers sources afin de transporter de façon la plus effective possible l'exclusion de garantie ; et chaque fichier devrait comporter au moins la ligne de réservation de droit (*"copyright"*) et une indication permettant de savoir où la notice complète peut être trouvée :

‹une ligne donnant le nom du programme et une brève idée de ce qu'il fait.› Copyright (C) ‹année› ‹nom de l'auteur› — Tous droits réservés. Ce programme est un logiciel libre ; vous pouvez le redistribuer ou le modifier suivant les termes de la *"GNU General Public License"* tels que publiés par la *Free Software Foundation* : soit la version 3 de cette licence, soit (à votre gré) toute version ultérieure. Ce programme est distribué dans l'espoir qu'il vous sera utile, mais SANS AUCUNE GARANTIE : sans même la garantie implicite de COMMERCIALISABILITÉ ni d'ADÉQUATION À UN OBJECTIF PARTICULIER. Consultez la Licence Générale Publique GNU pour plus de détails. Vous devriez avoir reçu une copie de la Licence Générale Publique GNU avec ce programme ; si ce n'est pas le cas, consultez : ‹http://www.gnu.org/licenses/›.

Ajoutez également les informations permettant de vous contacter par courrier électronique ou postal.

Si le programme produit une interaction sur un terminal, faites lui afficher une courte notice comme celle-ci lors de son démarrage en mode interactif :

‹programme› Copyright (C) ‹année› ‹nom de l'auteur› Ce programme vient SANS ABSOLUMENT AUCUNE GARANTIE ; taper "affiche g" pour les détails. Ceci est un logiciel libre et vous êtes invité à le redistribuer suivant certaines conditions ; taper "affiche c" pour les détails.

Les commandes hypothétiques "affiche g" et "affiche c" devrait afficher les parties appropriées de la Licence Générale Publique. Bien sûr, les commandes de votre

programme peuvent être différentes ; pour une interface graphique, vous pourriez utiliser une « boîte À propos. »

Vous devriez également obtenir de votre employeur (si vous travaillez en tant que programmeur) ou de votre école un « renoncement aux droits de propriété » pour ce programme, si nécessaire. Pour plus d'informations à ce sujet, et comment appliquer la GPL GNU, consultez ‹http://www.gnu.org/licenses/›.

La Licence Générale Publique GNU ne permet pas d'incorporer votre programme dans des programmes propriétaires. Si votre programme est une bibliothèque de sous-routines, vous pourriez considérer qu'il serait plus utile de permettre de lier des applications propriétaires avec la bibliothèque. Si c'est ce que vous voulez faire, utilisez la Licence Générale Publique Limitée GNU au lieu de cette Licence ; mais d'abord, veuillez lire ‹http://www.gnu.org/philosophy/why-not-lgpl.html›.