**Projet Labyrinthe**

**Olivier Guibert, Arnaud Pêcher et Paul Rénier**

Ce projet est à réaliser en binôme.

# 1 Préambule : réutiliser les bibliothèques de Processing en Java

Le langage `Processing` est une version simplifiée du langage `Java`, facilitant la programmation d'applications multimédia.

## 1.1 Présentation de l'environnement et du langage Processing

Le développement de `Processing` a été lancé en 2001 par Casey Reas et Benjamin Fry (MIT).

Il contient un environnement de développement très basique. En raison de sa simplicité, il est également très utilisé à des fins pédagogiques.

Outre ses vertus pédagogiques, Processing possède plusieurs qualités :
- bibliothèques multimédia performantes (2D, OpenGL, son) ;
- multi-cible : un même code permet de générer des binaires java, JavaScript/HTML5 ou même Android ;

Voici les principaux sites consacrés à `Processing`
- `http://processing.org/` : le site officiel ;
- `http://processing.org/tutorials/` : une sélection de tutoriels ;
- `http://processing.org/reference/libraries/` : des bibliothèques supplémentaires ;
- `http://fr.flossmanuals.net/processing/` : un manuel en français rédigé de manière collaborative ;
- `http://www.openprocessing.org/` : un dépôt centralisant des applications libres développées avec `Processing`.

Pour les autres langages, il existe des projets similaires : mentionnons `OpenFrameworks` (C++) ou encore `iProcessing` (pour iOS).

## 1.2 Réutiliser les bibliothèques de `Processing`

Il est facile de convertir une application `Processing` vers `Java` : il suffit de l'exporter. Le répertoire généré contient un sous-répertoire `source` qui contient le code source `Java` "pur" de l'application.

Le code `Java` ainsi généré contient une classe principale qui hérite (mot-clef `extends`) de la classe `PApplet`. Cela signifie qu'une application `Processing` s'exécute sous la forme d'une applet (une application prise en charge par un navigateur web).

Cette observation est suffisante pour pouvoir écrire directement une application `Java` tirant partie des bibliothèques de `Processing`.

Voici un extrait de code d'une applet :

```
void draw(){
  textSize(32); // taille de la police: 32
  text("Bonjour",50,50); // affichage de "Bonjour",
      coordonnees: 50,50
}

void setup(){
  size(300,100); // taille de la fenetre: 300x100
  background(0); // couleur de fond: noir
  fenetreBonjour();
}
```

Ce code met en avant deux méthodes importantes dans le fonctionnement d'une `applet`. Pendant l'exécution du programme, un moteur graphique tourne en parallèle : il se charge de redessiner le contenu des fenêtres. Au début du programme, le moteur graphique parcourt la méthode `setup` dans laquelle on peut initialiser les principaux paramètres (ici, la taille de la fenêtre et la couleur du fond). Ensuite, à intervalles réguliers (plusieurs fois par seconde), il exécute la méthode `draw` (ici, affichage d'un texte). Une application `Processing` est donc naturellement de type multiprocessus (légers) : un processus est dédié à la prise en charge de l'affichage.

Il faut bien garder à l'esprit que la méthode `draw` est exécutée plein de fois et ne doit contenir que ce qui relève de l'affichage.

Prenons maintenant pour exemple le code de démonstration (sous Processing) par Keith Peters (Examples/Topics/Motion/Follow1) :

```
**
 * Follow 1
 * based on code from Keith Peters.
 *
 * A line segment is pushed and pulled by the cursor.
 */

float x = 100;
float y = 100;
float angle1 = 0.0;
float segLength = 50;

void setup() {
  size(640, 360);
  strokeWeight(20.0);
  stroke(255, 100);
}

void draw() {
```

```
  background(0);

  float dx = mouseX - x;
  float dy = mouseY - y;
  angle1 = atan2(dy, dx);
  x = mouseX - (cos(angle1) * segLength);
  y = mouseY - (sin(angle1) * segLength);

  segment(x, y, angle1);
  ellipse(x, y, 20, 20);
}

void segment(float x, float y, float a) {
  pushMatrix();
  translate(x, y);
  rotate(a);
  line(0, 0, segLength, 0);
  popMatrix();
}
```

Pour la porter sous `eclipse`, il faut ajouter la bibliothèque `core` (fichier `core.jar` du répertoire `core/library` de `Processing`) et modifier la méthode `main` pour qu'elle lance l'application en mode `applet` :

```
import processing.core.*;

public class Application extends PApplet {
// idem : float x = ...

public void setup() {
  // idem : size(640,360); ...
}

public void draw() {
  // idem : background(0); ...
}

void segment(float x, float y, float a) {
  // idem : pushMatrix(); ...
}

public static void main(String[] args) {
   PApplet.main("Application");
  }
}
```

**Exercice 1** *(⋆⋆) Testez ce portage.*

Pour des explications plus complètes sur la réutilisation des bibliothèques de `Processing`, voir le tutoriel de Daniel Shiffman : `http://processing.org/tutorials/eclipse/`

**Exercice 2** *(⋆⋆) Faire un programme déplaçant le texte "Bonjour" de gauche à droite dans la fenêtre. Ajouter ensuite un rebond sur le bord droit de la fenêtre.*

**Exercice 3** *(⋆) Idem en ajoutant le contrôle de l'animation par la souris (la variable `mousePressed` est vraie lorsque l'utilisateur fait un clic sur la fenêtre).*

**Exercice 4** *(⋆⋆⋆) Idem en faisant un rebond progressif (ralentissement lors du rebond puis accélération) sur chaque bord. Indication : utiliser la fonction mathématique `sin`.*

**Exercice 5** *(⋆⋆⋆) Afficher le texte `Bonjour` via une animation constituée du déplacement des lettres une par une de la droite à la gauche.*

**Exercice 6** *(⋆⋆) Prenez l'exemple Topics/Motion/BouncyBubbles et recréez directement cette application en `Java` dans `eclipse`.*

# 2 Description du travail à réaliser

L'objectif est de réaliser une application permettant de se déplacer dans un labyrinthe 2D dont les salles carrées puis de greffer dessus des extensions.

Le format utilisé pour stocker un labyrinthe dans un fichier texte est le suivant :

```
37 37 // nombre de lignes et de colonnes de la grille
1 1 // coordonnees de l'entree
11 35 // coordonnees de la sortie
1 3 // coordonnees des autres salles
...
```

## 2.1 Socle minimal - 12 points

Il s'agit de réaliser une application Java exploitant les bibliothèques graphiques de `Processing` permettant de se déplacer de manière fluide dans le labyrinthe, avec un éclairage centré sur le héros (seule une vue partielle du labyrinthe est donc affichée).

L'archive `dist.zip` contient une démonstration du résultat attendu et des exemples de fichiers de labyrinthes sont dans l'archive `labys.zip`.

Pour vous faciliter dans la réalisation de votre projet, voici une suite d'exercices et d'indication, que vous êtes libre de suivre.

Ces exercices sont prévus pour être réalisés directement sous `Processing`. N'oubliez pas que le résultat final doit être une application `Java` sous la forme d'un projet sous Eclipse ...

**Exercice 7** *(⋆) Pour vous familiariser avec le format ci-dessus, ouvrez un fichier de labyrinthe dans un éditeur de texte et dessinez sur une feuille la partie du labyrinthe formée des salles dont les coordonnées sont inférieures ou égales à 10.*

On souhaite afficher le plan de ce labyrinthe dans une fenêtre graphique programmée avec `Processing`. Chaque case du labyrinthe sera de taille `taille` par `taille`, où `taille` est un paramètre de l'application. L'application possèdera également un paramètre `laby` qui stocke le nom du fichier du labyrinthe à afficher.

Lors de la lecture du fichier, vous allez récupérer des chaines de caractères représentant un nombre. Pour convertir une chaine `s` en un nombre `n`, vous pouvez faire :

```
int n = Integer.parseInt(s); // conversion de s (String) dans
    le nombre n(int)
```

**Exercice 8** *(⋆⋆) Dans la méthode `setup`, faites en sorte que la fenêtre soit d'une taille adaptée au labyrinthe, en prévoyant une marge de 10 pixels tout autour. La fenêtre aura un fond noir.*

**Exercice 9** *(⋆) Ecrire une méthode `dessinerSalle(int x, int y)` qui dessine en blanc la salle de coordonnées $x$ et $y$ (dans le système de coordonnées du labyrinthe)*

**Exercice 10** *(⋆) Ecrire deux méthodes `dessinerEntree` et `DessinerSortie`.*

**Exercice 11** *(⋆⋆) Ecrire une méthode `dessinerLabyrinthe` qui dessine tout le plan du labyrinthe puis ajuster la méthode `draw` pour qu'elle prenne en charge le dessin effectif du labyrinthe.*

**Exercice 12** *(⋆⋆) Modifier les méthodes de dessin pour qu'elles utilisent des images (de mur, de sol) pour dessiner le labyrinthe.*
*Voir `http://www.lemog.fr/lemog_textures/index.php` pour récupérer des images adaptées.*

Maintenant que l'affichage du plan du labyrinthe est acquis, il reste à rajouter la possibilité de déplacer un héros dans le labyrinthe, ainsi qu'un éclairage autour de sa position.

Pour l'instant, le plan du labyrinthe se dessine, mais on ne peut pas l'utiliser pour se déplacer : la structure du labyrinthe n'est pas modélisée.

**Exercice 13** (⋆⋆) *Créer une classe `Salle` : sachant qu'une salle possède des coordonnées $x$ et $y$, une couleur (de type `int`), peut se dessiner et peut indiquer si elle est adjacente à une autre.*

**Exercice 14** *(⋆⋆) Créez une classe `Labyrinthe` : un labyrinthe contient une collection des salles, la méthode `load` crée ces salles à partir des données du fichier texte stockées dans le tableau `lignes`, le labyrinthe peut se dessiner.*

**Exercice 15** *(⋆) Rajouter la prise en compte de l'entrée et de la sortie.*

**Exercice 16** *(⋆) Ajouter une classe* `Personnage` *: un personnage a une salle courante, peut se dessiner et se déplacer dans une autre salle.*

**Exercice 17** *(⋆⋆) Rajouter la gestion des déplacements du personnage, avec les touches z,q,s et d.*

**Exercice 18** *(⋆) Afficher un message lorsque le héros est sur la sortie pour l'informer qu'il a gagné.*

**Exercice 19** *(⋆⋆⋆) Modifier la gestion du déplacement pour que le déplacement soit fluide. Pour cela, il faut découpler le déplacement dans le labyrinthe (salle par salle) du déplacement graphique, pour que ce dernier se fasse pixel par pixel. Indication : avant que le héros puisse à nouveau se déplacer dans le labyrinthe, il faut que le déplacement graphique vers sa position courante soit achevé.*

`javadoc` (complète) Il manque encore la mise en place d'un éclairage autour du héros : l'affichage des salles sera de plus en plus sombre en fonction de leur distance (euclidienne) du héros.

**Exercice 20** *(⋆) Ecrire une méthode* `float distance(Salle t)` *dans la classe* `Salle` *qui retourne sa distance à une autre salle t.*

**Exercice 21** *(⋆⋆) Ecrire une méthode* `dessinEclairee(float distance)` *de la classe* `Salle` *qui dessine la salle en fonction de la valeur de distance, avec un rayon maximal de valeur 10. Modifier ensuite le moteur d'affichage pour qu'il exploite cette nouvelle méthode à la place de l'ancienne.*

### 2.2 Extensions (au choix) : au moins une - 8 points

1. (⋆) ajoutez des salles piégées (le héros perd un certain nombre de points de vie en les traversant) ;
2. (⋆⋆) ajoutez des portes et des clefs ;
3. (⋆⋆) ajoutes des monstres (qui se déplacent) ;
4. (⋆⋆⋆) réaliser un jeu de "pacman" exploitant cette architecture de labyrinthe.

## 3 Livrables

A rendre par mail à votre enseignant pour le **Vendredi 8 novembre 2013, 23h59mn59s** :
- un rapport au format `pdf` de 5 pages maximum comprenant une description des commandes pour recompiler/exécuter votre logiciel, le diagramme des classes et une explication de vos algorithmes les plus retors ;
- une archive au format `zip` de votre projet sous `eclipse` contenant :
  - un labyrinthe personnel ;
  - la `javadoc` complète ;
  - tout le code source, incluant des tests unitaires pour chacune des méthodes de chacune de vos classes ;
  - les bibliothèques nécessaires.