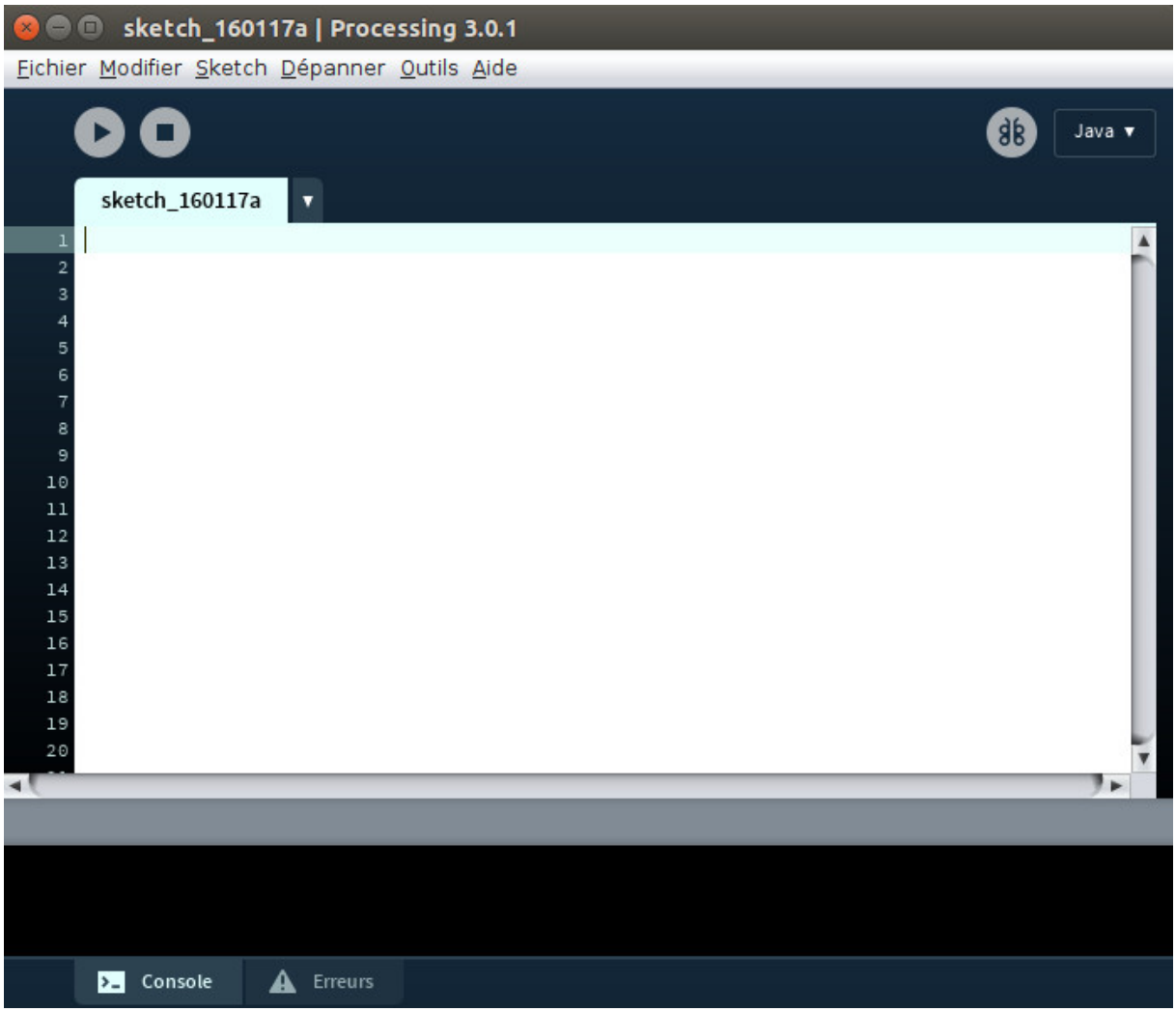
TP1 : Découverte de processing

I. Lancement

Pour démarrer processing, il suffit de lancer le programme **processing**, soit dans le terminal, soit en double cliquant dessus (c'est selon votre système d'exploitation). Normalement l'éditeur de processing devrait s'ouvrir :



Si vous avez déjà utilisé un éditeur pour programmer ou même un logiciel de traitement de texte quelconque vous devriez être en terrain connu : la majeur partie de l'éditeur de processing, se compose d'une zone blanche où vous allez écrire votre programme. En haut, vous avez un ensemble de menus :

- **Fichier** : pour toutes les actions sur votre projet, typiquement le sauvegarder ;
- **Modifier** : pour agir sur votre code, par exemple en commenter rapidement une portion ou l'indenter rapidement ;
- **Sketch** : pour agir sur votre programme, par exemple le lancer ou l'arrêter
- **Dépanner** : pour traquer les erreurs dans votre programme
- **Outils** : regroupe un ensemble d'outils annexes mais souvent très utiles
- **Aide** : regroupe des ressources pour vous aider à démarrer avec processing : documentation, tutoriaux etc.

Enfin deux boutons très importants : le gros triangle qui vous permet de lancer un programme et le carré qui vous permet de l'arrêter.

Créez un nouveau projet en le sauvegardant : `Fichier -> enregistrer`. Vous pouvez l'enregistrer dans votre dossier **Documents**, par exemple sous le nom **tp1**. Vous remarquez que Processing crée un nouveau dossier dans document, qu'il appelle **tp1**, avec dedans un unique fichier **tp1.pde**.

II. Exercice 1 : premier programme

Dans la suite de ces tps nous allons systématiquement utiliser deux fonctions de base de processing : la fonction **setup** et la fonction **draw**. Le code contenu dans la première est exécuté une seule fois, au lancement du programme. Le code contenu dans la seconde est exécuté en boucle, plusieurs fois par seconde, jusqu'à l'arrêt du programme.

Commençons avec la fonction **setup**. Nous allons débuter par la création d'un programme très simple : une fenêtre gris foncé, de largeur 500 pixels et de hauteur 250 pixels.

```
void setup(){
size(500,250);
background(20,20,20);
}
```

Vous pouvez le tester en cliquant sur le triangle. Notre programme utilise deux fonctions disponibles dans le langage processing :

- la fonction size qui définit la taille de votre fenêtre.
- la fonction background qui peint le fond dans une couleur particulière.

Nous indiquons à la fonction **size** la largeur puis la hauteur de notre fenêtre. Ensuite, nous donnons à la fonction **background** la couleur souhaitée, sous la forme de trois entiers : un pour la quantité de rouge, un pour la quantité de vert et un pour la quantité de bleu. Il existe de nombreuses manières de préciser la couleur (code hexadécimal, simple nuance de gris etc.). Pour le moment je m'en tiendrai à celle-ci. Pour connaître les quantités de rouge, de vert et de bleu pour une couleur particulière, je vous recommande d'utiliser le sélecteur de couleur de processing : `Outils -> Sélecteur de couleurs`.

- À l'aide de la documentation de la fonction size trouvez deux manières pour obtenir une fenêtre de la taille de votre écran.
- Quelle est la manière recommandée par Processing ?
- Changez la couleur de fond de votre application.

III. Quelques fonctions

L'objectif de ce TP est de créer un programme générant des carrés de tailles et de couleurs aléatoires. Processing offre un grand nombre de fonctions permettant de créer rapidement des affichages à partir de formes géométriques. Nous procèderons toujours de la même façon :

- nous commencerons par préciser la couleur de la forme, que ce soit pour son contour ou son intérieur ;
- nous décrirons la forme à dessiner, notamment sa taille et sa position.

Nous allons avoir besoin des fonctions suivantes:

- stroke
- noStroke
- fill
- noFill

- rect
- random

Avec la fonction **stroke** nous indiquons la couleur pour le contour. Par exemple pour un contour rouge

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

Avec la fonction **noStroke** nous indiquons que le contour de la forme tracée sera transparent.

```
noStroke();
```

Avec la fonction **fill** nous indiquons la couleur de l'intérieur de la forme. Par exemple pour une forme jaune avec un contour rouge :

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

Avec la fonction **noFill** nous indiquons que l'intérieur de la forme sera transparent.

```
noFill();
```

Avec la fonction **rect** nous dessinons un rectangle. Nous commençons par donner les coordonnées de son coin supérieur gauche puis sa largeur et enfin sa hauteur. Par exemple pour un rectangle rouge dont le coin supérieur gauche se situe aux coordonnées (10,20), de largeur 100 et de hauteur 50, on écrira :

```
noStroke();
stroke(255,0,0);
rect(10,20,100,50);
```

Avec la fonction **random** nous obtenons un nombre aléatoire situé dans un intervalle donné. Par exemple pour un nombre entre 1 et 6, on écrira :

```
float n = random(1,6);
```

Par ailleurs, nous utiliserons deux variables fournies par Processing :

- Width : la largeur de notre application ;
- Height : la hauteur de notre application.

Ces variables sont à votre disposition, vous n'avez pas à leur donner de valeurs, simplement à les utiliser

Enfin vous voudrez peut être sauvegarder les résultats obtenus. Vous pouvez utiliser la fonction save.

IV. Exercice 2 : rectangles

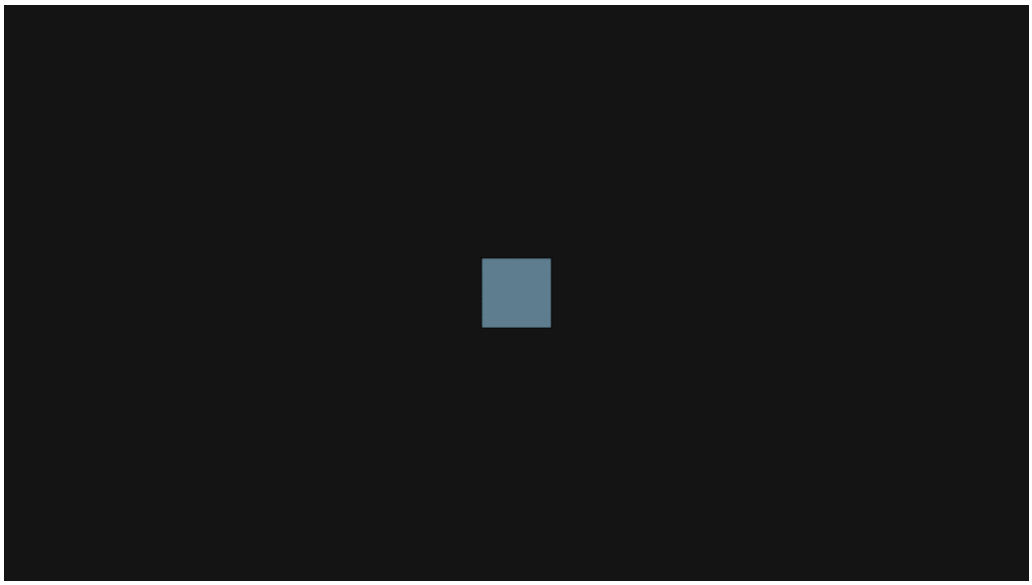
- Recopiez et testez le code suivant :

```
void setup() {
   fullScreen();
   background(20,20,20);
   stroke(0,0,0);
   float rouge=255;
   float vert=255;
   float bleu=0;
   fill(rouge,vert,bleu);
   float largeurRect=100;
   float hauteurRect=random(10,100);
   float xRect=width/2;
   float yRect=height/2;
   rect(xRect,yRect,largeurRect,hauteurRect);
 }

 //Pour sauvegarder
 void keyPressed() {
   save("tp0_result.png");
```

```
}
```

- Modifiez le calcul de xRect et yRect pour que le centre du rectangle corresponde au centre de l'application.
- Modifiez le calcul de la largeur du rectangle, pour quelle soit générée aléatoirement et comprise entre **10** pixels et **100** pixels.
- Modifiez le programme pour que la couleur soit déterminée de manière aléatoire : les variables rouge, vert et bleu doivent prendre des valeurs générées de manière aléatoire et comprise entre 0 et 255.
- Modifiez le programme pour que les formes tracées soient des carrés et non des rectangles

Vous devriez obtenir quelque chose dans ce goût-là :



V. Exercice 3 : boucle de dessin

Nous allons mainteant utiliser la fonction **draw** et y déplacer le code pour dessiner nos rectangles. La principale conséquence sera que notre programme ne va plus tracer un mais plusieurs rectangles.

- Recopiez et testez le code suivant :

```
void setup() {
  fullScreen();
  background(20, 20, 20);
}
void draw() {
  float rouge=255;
  float vert=255;
  float bleu=0;
  stroke(0, 0, 0);
  fill(rouge, vert, bleu);
  float largeurRect=100;
  float hauteurRect=random(10, 100);
  float xRect=random(0,width);
  float yRect=random(0,height);
  rect(xRect, yRect, largeurRect, hauteurRect);
}


//Pour sauvegarder
void keyPressed() {
  save("tp0_result.png");
}
```

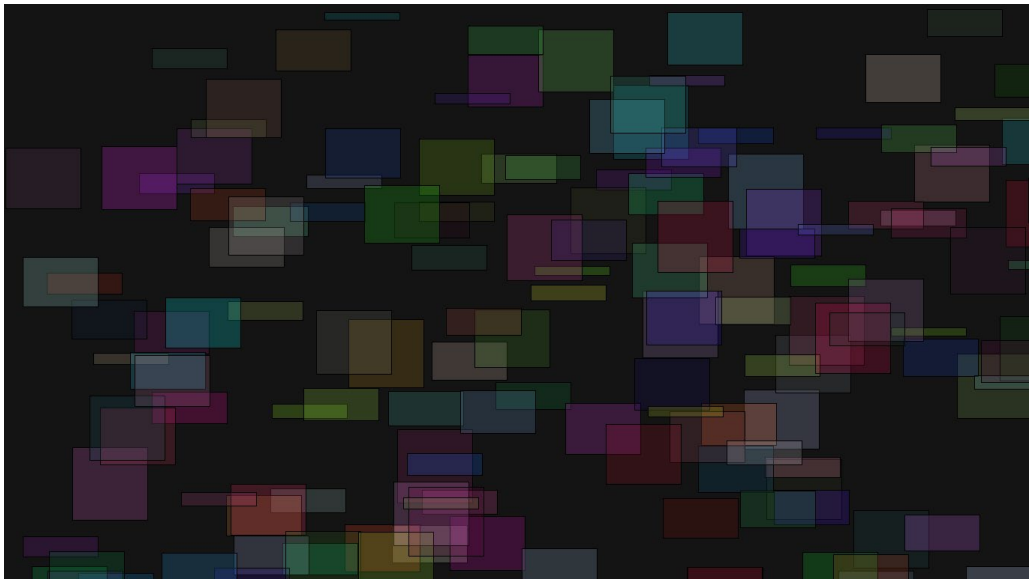
- En vous inspirant de ce que vous avez fait précédemment, modifiez le programme pour que la couleur de chaque rectangle soit déterminée de manière aléatoire.
- Lisez la documentation de la fonction frameRate et utilisez-là pour ralentir légèrement votre programme, histoire que personne ne fasse de crise d'épilepsie.

Pour le moment, nous nous contentons de donner des couleurs parfaitement opaques. Il est cependant possible d'indiquer à processing la transparence d'une couleur, grâce un quatrième paramètre : alpha. Le paramètre **alpha** prend une valeur entre 0 (invisible) et 255 (opaque). Par exemple pour un jaune transparent, on écrira :

```
float rouge=255;
float vert=255;
float bleu=0;
float alpha=50;
fill(rouge,vert,bleu,alpha);
```

- Modifiez votre programme pour que l'intérieur des rectangles soit légèrement transparent. Utilisez la même valeur alpha pour tous les carrés.

Vous devriez obtenir quelque chose dans ce goût-là :



VI. Exercice 4 : Une autre manière de définir la couleur.

Si vous ouvrez le sélecteur de couleurs (`Outils -> Sélecteur de couleurs`), vous remarquez qu'une couleur peut être soit définie par sa quantité de rouge, de vert et de bleu, soit par sa teinte **H**, sa saturation **S** et sa luminosité **B**

- La teinte correspond à la couleur générale et va de 0 à 360.
- La saturation correspond à l'intensité de la couleur et va de 0 (terne) à 100 (les couleurs des premiers sites web).
- La luminosité correspond à la quantité de lumière de la couleur et va de 0 à 100.

En vous aidant de l'outil sélecteur de couleur, répondez aux questions suivantes :

- Quelle est la teinte pour avoir une couleur plutôt verte ?
- Que se passe-t-il quand la luminosité est égale à 0 ?
- Quelle est la valeur de saturation adéquate pour obtenir des couleurs pastel ?
- Que se passe-t-il si la saturation vaut 0 ?

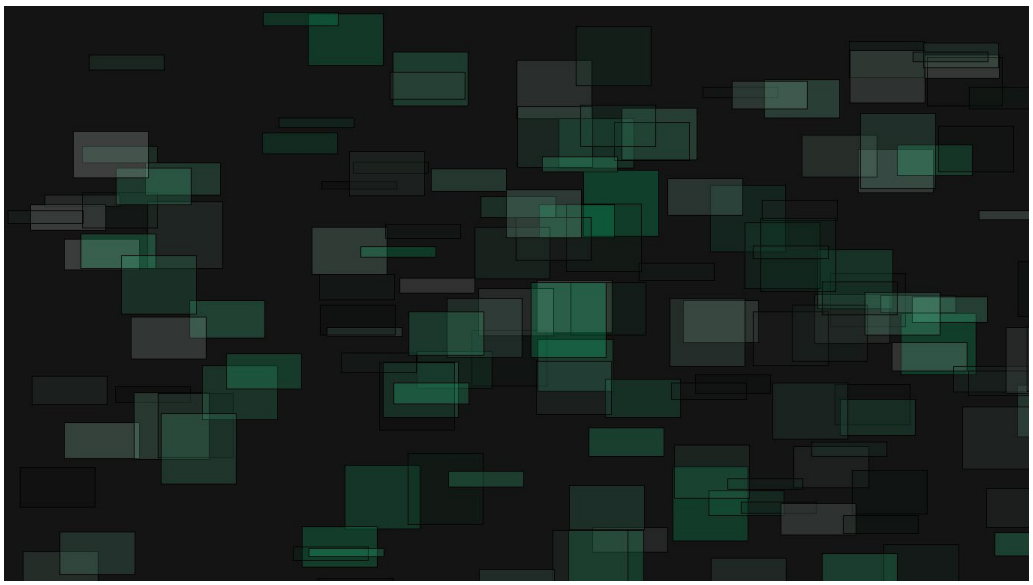
- Quelles sont les valeurs de teinte, de saturation et de luminosité pour notre gris foncé (rouge=20, vert=20 et bleu=20) ?

Dans la fonction **setup** ajoutez la ligne suivante, qui permet de générer les couleurs en mode teinte, saturation, luminosité :

```
colorMode(HSB,360,100,100,255);
```

- Dans la fonction **draw** supprimez les variables rouge, vert et bleu, puis remplacez-les par les variables teinte,sat,lum.
- Modifiez votre programme pour dessiner des rectangles de couleur verte, pour chaque carré des valeurs de saturation (variable **sat**) et de luminosité (variable **lum**) générée de manière aléatoire, entre 0 et 100.

Vous devriez obtenir quelque chose dans ce goût-là :



VII. Exercice 5 : de nouvelles formes

Processing ne permet pas seulement de tracer des rectangles, mais aussi des lignes, des ellipses, des triangles etc. Pour terminer ce tp, nous allons voir comment remplacer nos carrés par des cercles et des lignes. Mais n'hésitez pas à parcourir la documentation pour aller un peu plus loin. Vous verrez qu'il est même possible de faire des formes 3D.

VII. 1. Cercles

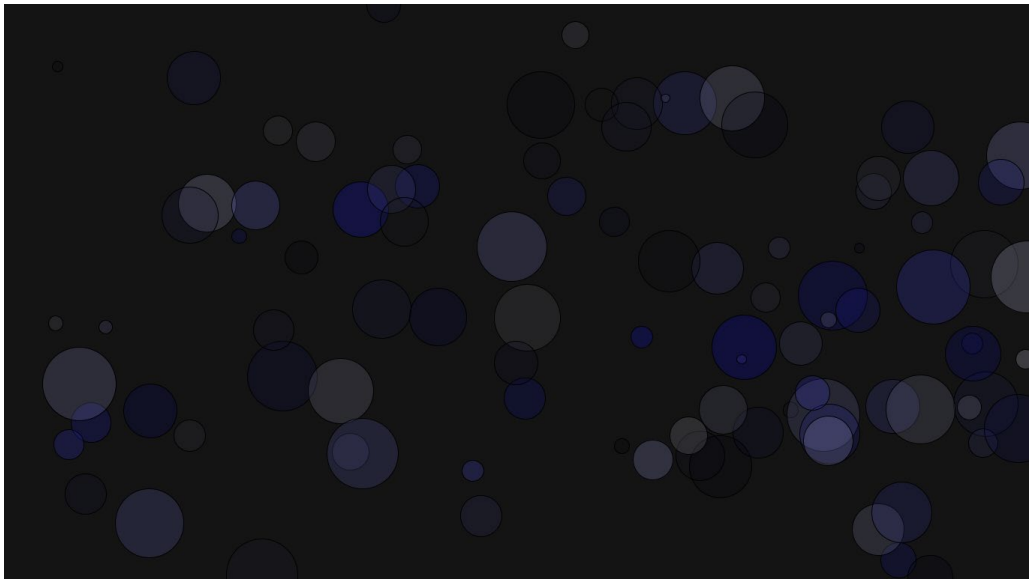
La fonction ellipse permet de tracer des ellipses et donc des cercles, qui ne sont rien d'autre que des ellipses pour lesquelles le plus petit diamètre est égal au plus grand diamètre. Il existe plusieurs manières de décrire une ellipse. Processing se sert du rectangle qui englobe l'ellipse. Vous allez fournir les quatre paramètres suivants :

- l'abscisse pour le centre du rectangle englobant l'ellipse
- l'ordonnée pour le centre du rectangle englobant l'ellipse
- la largeur du rectangle englobant l'ellipse
- la hauteur du rectangle englobant l'ellipse

Dans votre code, remplacez la fonction **rect** par la fonction **ellipse** et assurez-vous que la largeur et la hauteur du rectangle soient identiques.

- Remplacez les rectangles verts par des cercles bleus (et pas des ellipses!).

Vous devriez obtenir quelque chose dans ce goût-là :



VII. 2. Lignes

Pour Processing une ligne est un segment qui part d'un premier point et s'arrête à un second. Pour tracer une ligne, on utilise la fonction line et on lui passe les quatre paramètres suivants :

- l'abscisse pour le premier point
- l'ordonnée pour le premier point
- l'abscisse pour le second point
- l'ordonnée pour le second point

Nous allons tracer un faisceau de droite qui parcourt l'écran colonne par colonne et dont la teinte dépend de la position horizontale. Ce qui veut dire que nos lignes feront toujours la largeur de l'écran, donc que l'ordonnée du premier point sera toujours égale à 0, tandis que l'ordonnée du second point sera toujours égale à height. De plus l'abscisse du second point sera toujours égale à celle l'abscisse du premier point. Comme nous commençons à tracer nos lignes à partir de la gauche, la première valeur de l'abscisse pour nos deux points sera 0. Nous avons donc trois variables, initialisées de la manière suivante :

```
float y0;
float y1;
float x;
void setup() {
  fullScreen();
  background(20, 20, 20);
  colorMode(HSB,360,100,100,255);
  y0=0;
  y1=height;
  x=0;
}
```

J'ai déclaré ces trois variables en dehors de la fonction **setup** en tout début de mon fichier. Ainsi elles sont accessibles dans toutes les fonctions. Cela va me permettre de faire évoluer la valeur de x dans la fonction **draw** et d'accéder à cette valeur mise à jour à chaque appel de la fonction **draw**.

- De la même manière, déclarez les variables teinte,sat et lum qui vont permettre de gérer la couleur.
- Initialisez la teinte à 0.
- Initialisez sat avec un nombre aléatoire compris entre 0 et 100.
- Initialisez lum ave un nombre aléatoire compris entre 0 et 100.

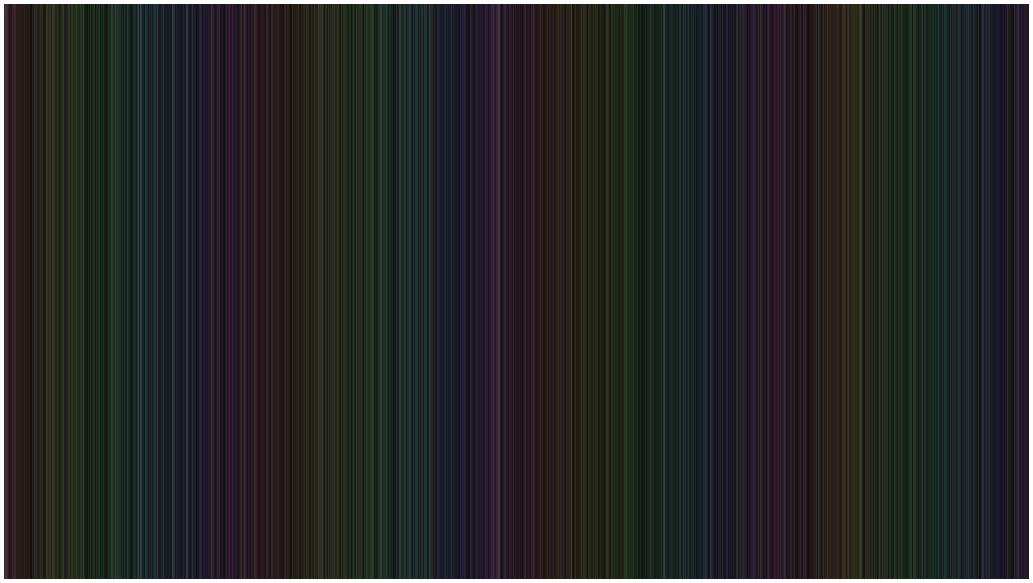
Dans la fonction **draw**, on va écrire le code pour tracer les lignes. Une ligne n'a qu'une couleur de contour et pas de couleur interne. Il est donc inutile d'utiliser la fonction **fill**.

- Appelez la fonction **stroke** avec comme paramètre teinte,sat et lum. Ne donnez pas de valeur de transparence.
- Ajoutez le code pour tracer une ligne dont le premier point est situé à (x,y0) et le second) (x,y1).
- Mettez à jour la saturation et la teinte en leur donnant une valeur aléatoire comprise entre 0 et 100.
- Mettez à jour l'abscisse pour la prochaine ligne en ajoutant 1 à x.
- Pour que le programme reparte à l'abscisse 0 lorsqu'il a atteint la dernière ligne, vous utilisez la fonction modulo pour mettre à jour x :

```
x=(x+1)%width;
```

- Sur le même principe mettez à jour la teinte en lui ajoutant 1 et en utilisant la fonction modulo pour que lorsque la teinte vaut 360 elle soit remise à 0.

Vous devriez obtenir quelque chose dans ce goût-là :



TP2 : Hexagones

I. Lancement

Lancez Processing et créez un nouveau projet. Vous pouvez l'enregistrer dans votre dossier **Documents**, par exemple sous le nom **Hexagones**. Vous remarquez que Processing crée un nouveau dossier dans document, qu'il appelle **Hexagones**, avec dedans un unique fichier **Hexagones.pde**.

Dans la suite de ces tps nous allons systématiquement utiliser deux fonctions de base de processing : la fonction **setup** et la fonction **draw**. Le code contenu dans la première est exécuté une seule fois, au lancement du programme. Le code contenu dans la seconde est exécuté en boucle, plusieurs fois par seconde, jusqu'à l'arrêt du programme.

Voici le code pour déclarer ces deux fonctions et afficher une fenêtre grise, de la taille de l'écran.

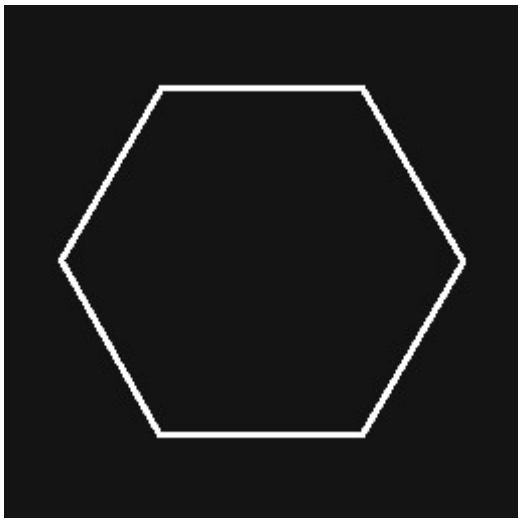
```
void setup(){
size(displayWidth,displayHeight);
background(20,20,20);
}

void draw(){
}
```

Pour rappel :

- La fonction **size** définit la taille de votre fenêtre.
- La variable **displayWidth** correspond à la largeur de votre écran en nombre de pixels.
- La variable **displayHeight** correspond à la hauteur de votre écran en nombre de pixels.
- La fonction **background** peint le fond dans une couleur particulière.

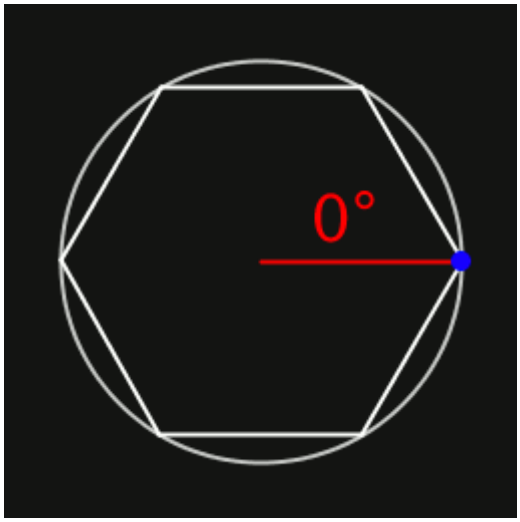
II. Premier hexagone

Tout ça, c'est bien gentil, mais le sujet de ce TP reste quand même les hexagones, et il est temps de poser la question qui fâche : comment trace-t-on un hexagone ? Alors d'abord, pour ceux qui ne s'en souviendraient pas, voici l'animal :

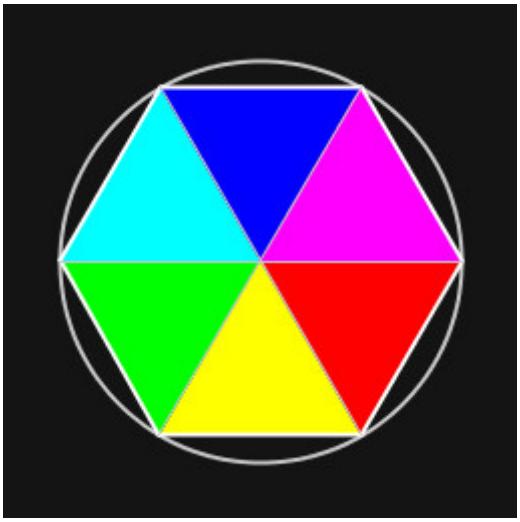


Si vous regardez un peu les propriétés de cette figure, vous allez vous rendre compte de deux choses qui vont nous permettre de le dessiner très simplement :

- l'hexagone s'inscrit dans un cercle ;
- l'angle entre deux sommets consécutifs est toujours égale à 60°



Et comme un dessin vaut mieux que de longs discours



Pour tracer notre hexagone, nous allons simplement nous déplacer sur le cercle et ses arrêtes au fur et à mesure

III. Exercice 1

- Déclarez une variable **nbCotes** de type int ;
- Initialisez-là à 6;
- Avant la fonction **setup**, déclarez une variable **theta** de type float;
- Dans la fonction **setup**, initialisez-là à 0 ;
- Avant la fonction **setup**, déclarez une variable **R** de type float ;
- Dans la fonction **setup**, initialisez-là à 50 ;
- Écrivez une boucle qui s'exécute autant de fois que la valeur stockée dans **nbCotes** ;

Nous allons maintenant donner les coordonnées de deux sommets consécutifs et tracer l'arrête qui les relie. Des années de recherches laborieuses ont permis à l'humanité de découvrir que les coordonnées d'un point situé sur un cercle de rayon *R* et de centre (centreX,centreY) selon un angle *theta* peuvent se calculer de la manière suivante :

- x = centreX + cos(theta)*R;
- y = centreY + sin(theta)*R;

Pour notre problème nous avons déjà **R** et **theta**, il ne nous manque que le centre du cercle. Je vous propose de le placer au centre de notre écran. A l'intérieur de votre boucle, vous allez donc pouvoir ajouter les deux lignes suivantes :

```
float x0=(displayWidth/2) + cos(theta)R;
float y0=(displayWidth/2) + sin(theta)R;
```

Voilà, pour notre premier sommet. Pour tracer une arrête, il nous faut un deuxième sommet, décalé par rapport au premier d'un angle de 60 degrés. Dans un monde parfait nous pourrions simplement augmenter la valeur de notre variable theta de 60. Sauf que les fonctions cosinus et sinus ne fonctionnent pas avec des degrés mais avec des radians. Pour convertir un degré en radian, il vous faut le multiplier par **2*PI/360**. Donc, notre angle doit être mis à jour de la manière suivante :

```
theta = theta + 60(2PI/360);
```

Nous pouvons calculer le deuxième sommet, de la même manière que le précédent et, enfin, tracer une ligne blanche entre les deux sommets. Pour la couleur blanche utilisez la fonction stroke et pour tracer la ligne la fonction line.

- Après la boucle, remettez la variable *R* à 0.

Si vous avez tout fait correctement, votre programme devrait tracer un hexagone.

IV. Exercice 2 : pleins d'hexagones !

Pour le moment chaque fois que la fonction **draw** s'exécute, le même hexagone est tracé. Ce que nous allons faire, c'est simplement changer le rayon, en rajoutant la ligne suivante à la fin de la fonction **draw** :

```
R=R+10;
```

Rien de bien compliqué : on se contente augmenter le rayon de 10, chaque fois qu'on a fini de tracer un hexagone.

- Testez votre programme ;
- Faites varier le rayon **R** en lui ajoutant un nombre aléatoire entre 10 et 30 ;
- Faites varier l'angle theta, en lui donnant une valeur entre **(2*PI/360)** et **50*(2*PI/360)**

Le résultat obtenu n'est pas très esthétique et surtout, nous avons vite fait d'atteindre la taille maximale pour un hexagone. Nous allons faire juste deux petites modifications qui vont considérablement améliorer le dessin produit par notre programme.

- Au début de la fonction **draw**, ajoutez les deux lignes suivantes, qui vont vous permettre de faire varier l'épaisseur du trait pour l'hexagone tracé
- `float epaisseur=random(2,6);`
- `strokeWeight(epaisseur);`
- Nous allons effectuer un test sur le rayon de l'hexagone : quand celui-ci aura atteint sa taille maximale (c'est-à-dire la moitié de la diagonale de l'écran), nous réinitialiserons le rayon *R* avec une valeur comprise entre **5** et **5**

Au début de la fonction **draw**, ajoutez la ligne suivante qui calcule la distance maximale pour le rayon

```
float maxR=sqrt(displayWidth*displayWidth + displayHeight*displayHeight)/2;
```

- Ecrivez le test (**if***) qui :
  - si le **R** est inférieur à **maxR** lui ajouter une valeur aléatoire comprise entre **10** et **30**
- sinon donne à **R** une valeur aléatoire comprise entre **5** et **50**
- Dans la fonction **setup** diminuez le frameRate : `frameRate(10)`
- Ne tracez pas votre ligne dans un blanc pur, mais dans un blanc transparent : `stroke(255,255,255,50)`

V. Exercice 3 : de la couleur

Jusqu'à présent tous nos hexagones sont tracés de la même couleur. Nous allons faire en sorte que chaque fois que notre programme atteint le rayon maximal (**maxR**) il change la couleur des hexagones. Nous allons seulement utiliser deux couleurs :

- le blanc transparent : `stroke(255,255,255,50)`
- un jaune doré : `stroke(193, 158, 18,50);`

- En dessous de la variable **R**, déclarez une variable **couleur** de type **int** ;
- Dans la fonction **setup** initialisez cette variable à **0** ;
- Remplacez la ligne `stroke(255,255,255,50)` par le test suivant :
  - si couleur est égal à 0, tracez en blanc : `stroke(255,255,255,50)`
  - sinon, tracez en jaune : `stroke(193, 158, 18,50);`
- Où faut-il changer la valeur de couleur pour changer de couleur lorsque le rayon maximal **maxR** est atteint ?

- Pourquoi la mise à jour `couleur = couleur + 1;` ne permet-elle pas d'alterner les deux couleurs ?
- Grâce à la fonction **modulo**, proposez une mise à jour de la couleur, qui permette une alternance des couleurs ;

Nous avons obtenu un processus intéressant, mais très vite l'écran est saturé de couleur. La modification qui suit va nous permettre d'avoir un programme qui s'autorégule, en traçant régulièrement des hexagones opaques de la couleur du fond : `stroke(20,20,20)`

L'astuce est la suivante :

- quand couleur == 0 : du blanc ;
- quand couleur == 1 : la couleur du fond ;
- quand couleur == 2 : du jaune ;
- quand couleur == 3 : la couleur du fond.

Pour obtenir ce comportement :

- Modifiez la mise à jour de la couleur ;
- Modifiez le test sur la couleur ;

PARTIE II. TP3 : Manipulation dimages

I. Objectif

Dans ce TP vous allez apprendre à afficher une image et chahuter ses pixels.

II. Afficher une image

Tout d'abord il va nous falloir une image. Comme cette première étape est **essentielle** au bon déroulement du TP, en voici une que j'aime bien: Je vous conseille de l'utiliser, comme ça vous aurez les mêmes résultats que moi et ça sera plus facile pour comparer.

Nous allons commencer comme pour n'importe quel projet avec Processing, avec le code suivant :

```
void setup(){
  size(displayWidth,displayHeight);
  background(255,255,255);
}

void draw(){
}
```

Pour rappel la fonction `setup()` contient des instructions qui ne seront exécutées qu'une seule fois, au lancement de notre programme, tandis que la fonction `draw()` est appellée plusieurs fois par seconde, tout le temps que notre programme fonctionne.

Enregistrez le projet (vous pouvez lui donner un nom très original comme **tp2**). Processing va créer un répertoire du même nom que votre projet dans lequel vous trouvez un fichier **.pde** qui contient votre code. Juste à côté de ce fichier, copiez votre image. Pour la suite de ce tp, je supposerai qu'elle porte le doux nom de **img.jpg**.

Pour pouvoir l'utiliser, nous allons stocker notre image dans une variable, de type PImage, que nous appellerons **monImage**. Modifiez juste un tout petit peu le programme précédent :

```
PImage monImage; // La modification importante est ici !!!

void setup(){
  size(displayWidth,displayHeight);
```

```
  background(240);
}

void draw(){
}
```

Et maintenant attention, les choses sérieuses commencent ! Nous allons nous placer à l'intérieur de la fonction `setup` et écrire les lignes de code suivantes :

- `monImage = loadImage("im.jpg");` : pour stocker notre image, dans la variable monImage
- `copy(monImage,0,0,monImage.width,monImage.height,0,0,monImage.width,monImage.height);` : pour dessiner l'image. Vous devriez obtenir l'affichage suivant :

III. Exercice 1

La dernière ligne de code est un peu obscure :

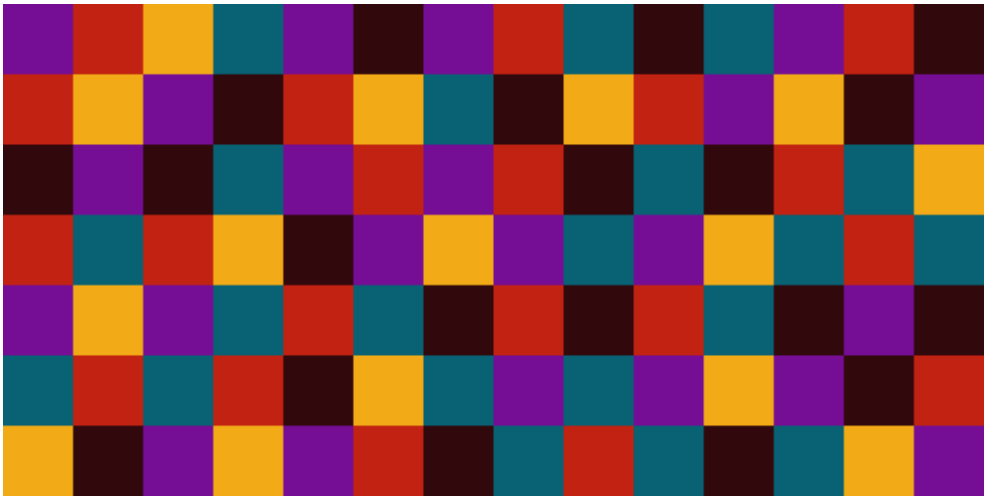
- Allez voir sa documentation
- Comment faire pour que le coin supérieur gauche de notre image se situe au pixel de coordonnée (50,10) ?
- Comment faire pour diminuer par 2 la taille de notre image (parce que c'est pas pour dire, mais là, elle déborde un peu...) ?

III. 1. Pixels

Vous le savez sans doute, mais une image est une grille rectangulaire. Chaque élément de la grille est appelé pixel et possède une couleur. Concrètement cela veut dire qu'une PImage est composée d'au moins trois choses :

- sa largeur : `monImage.width`
- sa haureur : `monImage.height`
- des pixels : `monImage.pixels`

La largeur correspond au nombre de pixels qu'on peut compter de la droite vers la gauche. La hauteur correspond au nombre de pixels qu'on peut compter du haut vers le bas. Par exemple voici une image de largeur 14 pixels et de hauteur 7 pixels.



Nous allons voir comment parcourir les pixels d'une image pour les lire ou les modifier. Enlevez cette ligne :

```
copy(monImage,0,0,monImage.width,monImage.height,0,0,monImage.width,monImage.height);
```

D'abord on va diviser par 2 la taille de notre image :

```
monImage.resize(monImage.width/2,monImage.height/2);
```

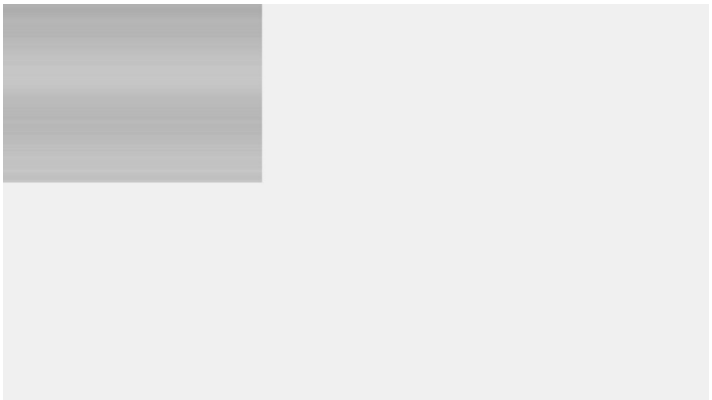
Nous allons noter **y** le numéro de ligne du pixel et **x** son numéro de colonne. Dans notre PImage, les pixels sont tous entassés les uns à la suite des autres, d'abord les pixels de la première ligne, ensuite les pixels de la seconde ligne etc. De plus le premier pixel a le numéro 0, le deuxième le numéro 1 et ainsi de suite. Un peu compliqué ? Heureusement, processing dispose d'une fonction **get** qui permet de récupérer facilement la couleur du pixel :

```
color c = monImage.get(x,y);
int rouge = red(c);
int vert = vert(c);
int bleu = bleu(c);
```

IV. Exercice 2

Maintenant à vous de travailler

- Écrivez la boucle permettant de parcourir toutes lignes de l'image
- À l'intérieur de la boucle, lisez la couleur du premier pixel de la ligne
- Utilisez ses valeurs de rouge, de vert et de bleu pour préciser la couleur de contour et des formes à tracer avec les fonctions `stroke` et `fill`
- Tracez un rectangle qui commence au point de coordonnée (0,y), dont la largeur est de 500 et la hauteur de 1, avec la fonction
- Vous devriez obtenir quelque chose comme ça:

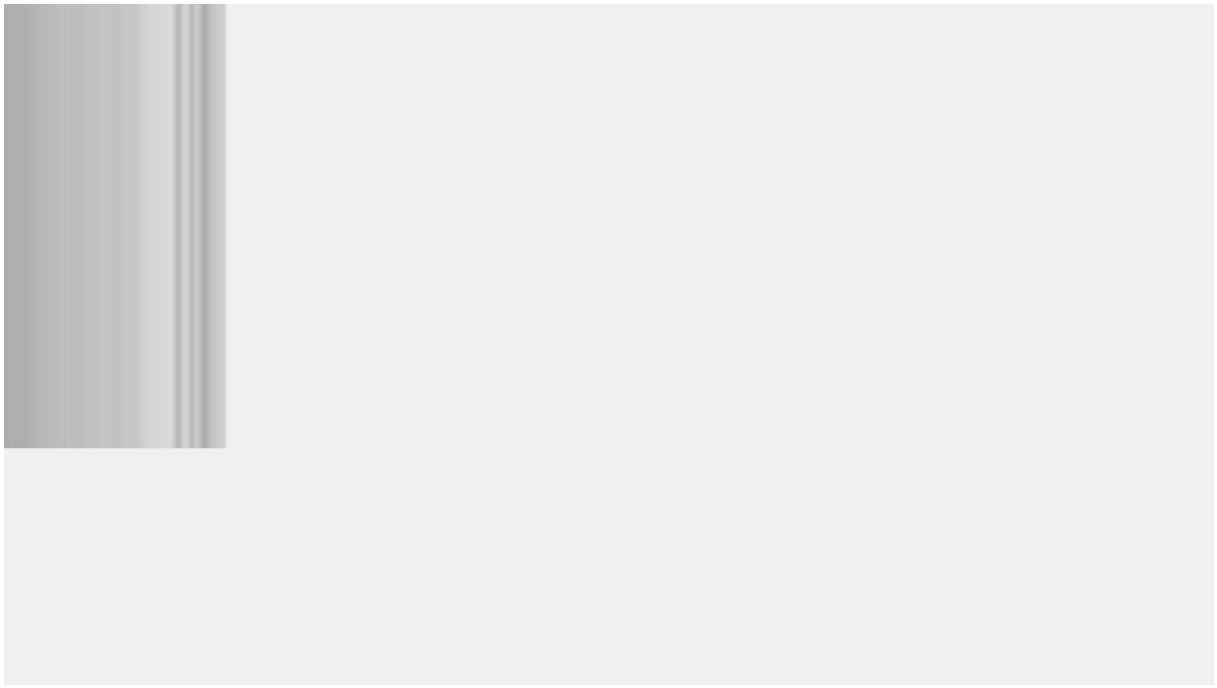


V. Exercice 3

La même chose, mais cette fois-ci en parcourant les colonnes !

- Tracez un rectangle qui commence au point de coordonnée (x,0), dont la largeur est de 1 et la hauteur de 500, avec la fonction

  Vous devriez obtenir ceci :



VI. Exercice 4

On va voir si vous suivez : que fait ce bout de code ?

```
monImage.resize(monImage.width/2,monImage.height/2);
int y=0;
int i=0;
while( y < monImage.height ){
  int x=0;
  while( x < monImage.width ){
    color c = monImage.get(x,y);
    fill(c);
    stroke(c);
    rect(x,y,1,1);
    x = x + 1;
  }
  y = y + 1;
}
```

VII. Exercice 5

- Modifiez le code de l'exercice 4 pour n'afficher qu'un pixel sur 10 et tracer des rectangles de taille 10*10.
- Remplacez les rectangles par des cercles en utilisant la fonction **ellipse**

VIII. Exercice 6

- Testez le code suivant

```
PImage monImage;

void setup() {
  size(displayWidth, displayHeight);
  background(255);
  monImage = loadImage("im.jpg");


  monImage.resize(monImage.width/4, monImage.height/4);

  int yDessin = 0;
  int y = 0;
  int i = 0;
  while ( y < monImage.height ) {
    int x = 0;
    int xDessin = 0;
    while ( x < monImage.width ) {
      color c = monImage.get(x,y);
      fill(red(c),green(c),blue(c));
      stroke(red(c),green(c),blue(c));
      int r = int(random(1,7));
      ellipse(xDessin, yDessin,r,r);
      xDessin = xDessin + 7;
      x = x + 1;
    }
    yDessin = yDessin + 7;
    y = y + 1;
  }
}

void draw() {
}

void keyPressed() {
  save("resultat.png");
}
```

- Quelles sont les différences avec le programme de l'exercice 5 ?

IX. Exercice 7 : de la couleur

Il ne vous aura pas échappé que notre image est un peu particulière : elle n'a pas de couleurs. Il s'agit d'une **image en niveaux de gris**, parce que chaque pixel de l'image est en fait une teinte de gris particulière. En fait, c'est parce que pour ce type d'image il n'y a pas vraiment de couleur, juste une intensité lumineuse allant du noir (0) au blanc (255). C'est exactement ce qui se passe avec les tous premiers appareils photographiques : ils ne captent que la lumière, toute l'image est construite à partie des contrastes entre les ombres et les zones éclairées. Et c'est précisément pour cette raison que j'aime beaucoup cette photographie : parce qu'on peut la découper facilement en trois types de zones

- *des zones sombres, avec des pixels ayant un niveau de gris entre 0 et 77*

- *des zones médianes, avec des pixels ayant un niveau de gris entre 77 et 200*

- *des zones claires, avec des pixels ayant un niveau de gris entre 200 et 255*

Nous allons modifier notre image pour afficher ces trois zones, dans trois couleurs différentes.

- Pour accéder au niveau de gris de l'image, rajoutez la ligne suivante :

```
color c = monImage.get(x,y);
int gris = int(brightness(c));
Quel test faut-il effectuer pour savoir si le niveau de gris appartient à une zone claire ?
```

- Dans ce cas-là, tracez un cercle en noir (0, 0, 0).
- Quel test faut-il effectuer pour savoir si le niveau de gris appartient à une zone médiane ?
- Dans ce cas-là, tracez un cercle en mauve (101, 49, 124).
- Quel test faut-il effectuer pour savoir si le niveau de gris appartient à une zone sombre ?
- Dans ce cas-là, tracez un cercle en orange (224, 71, 0).
- Modifiez le calcul des coordonnées pour le dessin, de manière à ce que l'image soit centrée