16. Classes

16.1 Introduction à la programmation orientée objet

La programmation dite “orientée objet” déroute souvent un peu les profanes. Si son principe de fonctionnement semble assez intuitif, sa mise en œuvre ne l’est pas et sa manipulation impose une gymnastique mentale que l’on met toujours un petit temps à maîtriser complètement. Il serait pourtant dommage de faire l’impasse sur un outil aussi puissant.

En devenant très populaire à la fin des années 1980, l’informatique a fait un pas de géant grâce à ce type de programmation et a pu créer des jeux vidéo toujours plus sophistiqués.

Prenons un exemple très concret.

Le chat de votre voisin s’appelle Félix, il a six ans et il est noir. Votre chat se nomme Viking, il a trois ans et il est roux.

Eh bien, Félix et Viking, bien qu’ayant des noms et des pelages distincts, ont un rapport : ce sont tous les deux des chats.

En programmation orientée objet, nous dirons que Félix et Viking sont deux instances (ou deux occurrences) de la classe (ou de l’objet) `chat`.

Ces deux chats ont, comme tous les chats, la capacité de miauler et celle de rester devant une porte ouverte sans se décider à sortir.

Voilà comment nous pourrions écrire le constructeur de l'objet `chat` :

```
class chat{
String nom, couleur;
  int age;
  chat (String n, String c, int a){
    nom = n;
    couleur = c;
    age = a;
  }
  void miaule(){
    println("miaou");
  }
  void seDemandeSilDoitSortirOuPas(){
    delay(10000);
    miaule();
  }
}
```

Ce que l'on appelle "constructeur" en programmation orientée objet, c'est un ensemble de variables et de fonctions que toutes les occurrences d'une même classe d'objets partagent. Tous les chats créés avec ce constructeur auront accès à deux variables de type `String` (chaîne de caractères) qui sont `couleur` et `nom`. Tous les chats créés avec ce constructeur auront accès à deux fonctions, `miaule()`, qui affiche le mot `miaou` dans la zone de texte, et `seDemandeSilDoitSortirOuPas()` qui exécute la commande `miaule()` après un gel du programme de dix secondes.

Nous reviendrons plus loin sur la syntaxe précise de notre constructeur, mais voici comment on "crée" des occurrences d'objets de type `chat` :

```
chat felix = new chat("félix", "noir", 6);
chat viking = new chat("viking", "roux", 3);
```

On peut réduire l'opération à ceci :

```
objet nomVariable = new objet (arguments);
```

où `objet` est le nom du constructeur que nous utilisons et où `nomVariable` est le nom par lequel nous voulons pouvoir retrouver notre instance.

Supposons que nous voulions à présent faire miauler Félix. Il suffira d'écrire :

```
felix.miaule();
```

L'instance `felix` exécutera sa fonction `miaule()`, mais pas son camarade `viking`, car chaque instance de notre classe `chat` agit indépendemment des autres.

16.2 La rédaction d'un constructeur

Dans Processing, les constructeurs se rédigent typiquement ainsi :

```
class nomObjet {
déclaration de variables ;
  nomObjet ( arguments ) {
    affectation de variables
  }
  type fonction1 () {
    /* instruction */
  }
  type fonction2 () {
    /* instruction */
  }
}
```

La classe est donc un bloc d'instructions encadré d'accolades (`{ }`) et précédé du mot-clé `class` et du nom de la classe, nom que nous choisissons à notre guise (par exemple : `chat`) mais qui ne doit pas être un mot-clé réservé par Processing.

Les lignes qui succèdent à l'accolade ouvrante sont réservées pour accueillir toutes les déclarations de variables de chaque instance de notre objet, du moins toutes les variables auxquelles chaque instance pourra faire référence à tout moment.

Vient ensuite une première fonction qui porte exactement le même nom que la classe d'objet à laquelle elle appartient. On remarque que cette fonction, contrairement à toutes celles qui peuvent être rédigées sous Processing, n'est pas précédée d'un type tel que `int`, `float` ou encore `void`. C'est cette fonction qui est le "constructeur" proprement dit, c'est-à-dire qu'une instance de l'objet est créée lorsque l'on invoque cette fonction, précédée du mot-clé `new`, comme ceci :

```
new nom_Objet(arguments éventuels);
```

Les arguments qui sont transmis au moment de la construction de l'objet servent généralement à lui donner ses caractéristiques individuelles. Par exemple, dans le cas de nos chats, nous avions passé comme arguments le nom et la couleur du pelage de l'animal. Au constructeur, ensuite, de gérer les informations transmises en arguments et, par exemple, de les affecter à ses variables.

Reprenons notre classe `chat` en la simplifiant au maximum :

```
class chat {
String nom;
  chat ( String n ) {
    nom = n;
  }
}
```

Ici, nous n'avons plus dans notre objet qu'une fonction (le constructeur `chat`) et une variable, la chaîne de caractères `nom`. Au moment de l'instanciation, c'est-à-dire au moment où nous créons une instance de l'objet chat, nous le faisons en lui transmettons une chaîne de caractères :

```
new chat ("félix");
```

Cette chaîne de caractères (ici : `"félix"`) atterrit dans la variable nommée `n`, qui disparaîtra sitôt l'objet créé. Le programme demande ensuite d'affecter le contenu de `n` à la variable `nom`, qui a été déclarée en en-tête de la classe `chat`.

Si nous écrivons juste `new chat ("félix")`, l'objet de type `chat` sera bien construit mais nous ne pourrons pas l'utiliser. C'est pourquoi on doit stocker le résultat de l'opération dans une variable, variable qui sera déclarée avec le type `chat` :

```
chat chat1 = new chat ("félix");
chat chat2 = new chat ("viking");
```

Il est aussi possible de créer une liste à partir d'un objet, comme ceci :

```
chat[] tousLesChatsDuQuartier ;

void setup(){
  tousLesChatsDuQuartier = new chat[3];
  tousLesChatsDuQuartier[0] = new chat("félix");
  tousLesChatsDuQuartier[1] = new chat("viking");
  tousLesChatsDuQuartier[2] = new chat("bijou");
}
```

```
class chat {
  String nom;
  chat(String n){
    nom = n;
  }
}
```

Dans cet exemple, nous commençons par déclarer une liste d'objets de type `chat[]` (les crochets signifient qu'il s'agit d'une liste) nommée `tousLesChatsDuQuartier`.

Pour que Processing accepte cette déclaration de liste, il faut que la classe `chat` existe, ce qui est le cas.

Ensuite, dans la fonction `setup()`, nous initialisons la liste `tousLesChatsDuQuartier` en rappelant qu'il s'agit d'une liste de type `chat[]` et en indiquant que cette liste contiendra au total trois éléments.

Cela étant fait, nous affectons une valeur à chacun de ces trois éléments.

La syntaxe qui suit aura exactement le même effet mais avec une petite variante :

```
chat[] tousLesChatsDuQuartier ;

void setup(){
  tousLesChatsDuQuartier = new chat[0];
  new chat("félix");
  new chat("viking");
  new chat("bijou");
}

class chat {
  String nom;
  chat(String n){
    nom = n;
    tousLesChatsDuQuartier = (chat[]) append
(tousLesChatsDuQuartier, this);
  }
}
```

Cette fois-ci, la liste `tousLesChatsDuQuartier` est créée mais avec zéro élément. C'est chaque instance qui, dans son constructeur, s'ajoutera à la liste à l'aide de la commande `append()` et du mot-clé `this`, qui est la manière dont chaque objet se décrit lui-même.

Pour l'objet créé avec le nom "félix", par exemple, `this.nom` sera `félix`. Si l'on écrit la commande:

```
println(this);
```

dans une fonction de l'objet `chat`, le résultat ressemblera à ceci :

```
nomDuProgramme$chat@9cfec1
```

Ce résultat illisible nous indique que `this` est l'instance numéro `9cfec1` de l'objet `chat` du programme nommé `nomDuProgramme` (le nom sous lequel vous avez enregistré le programme).

Le numéro `9cfec1` (au hasard) a été attribué à la volée à l'instance au moment de sa création et ne changera plus. Chaque instance possède un identifiant numérique unique de ce genre. Vous pouvez être troublé d'apprendre que `9cfec1` est un nombre puisqu'il contient des chiffres et des lettres, mais il s'agit en fait d'un nombre noté en hexadécimal, c'est-à-dire d'un nombre en base 16 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F) et non en base 10 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). On emploie des lettres pour écrire les chiffres 10, 11, 12, 13, 14 et 15 car la numération arabe s'arrête à 9.

Rien ne se passe !

ATTENTION

Notez bien que si vous exécutez ce code, Processing le trouvera à sa convenance et ne signalera aucun message d'erreur. Mais rien ne se passera à l'écran car à aucun moment nous ne demandons d'action particulière à nos objets. Les apprentis programmeurs sont souvent un peu découragés à ce stade : des chats existent dans la mémoire vive de l'ordinateur, mais où sont-ils ? On ne les voit pas.

Comme dit en introduction, la programmation "orientée objet" réclame un certain niveau d'abstraction. Nous allons donc revenir à du concret avec le programme qui suit :

```
cercle[] tousLesCercles;

void setup() {
  size(200, 200);
  stroke(255);
  noFill();
  tousLesCercles = new cercle[0];
}

void draw() {
  background(0);
  for(int a=0;a<tousLesCercles.length;a++) {
    tousLesCercles[a].dessine();
  }
}

class cercle {
  float x, y, taille;
  cercle() {
    x=mouseX;
    y=mouseY;
    taille = 0.1;
    tousLesCercles = (cercle[]) append
(tousLesCercles, this);
  }
  void dessine() {
    if(taille<500) {
      taille*=1.01;
      taille+=1;
      ellipse(x,y,taille, taille);
    }
  }
}

void mouseReleased() {
  new cercle();
}
```

Ce programme contient une classe (`cercle`) et trois fonctions (`setup()`, `draw()` et `mouseReleased()`).

En tête de programme, nous déclarons l'existence d'une liste d'objet `cercle`, que nous avons inventé, dont le nom est `tousLesCercles`.

Dans la fonction `setup`, qui est lancée au début de l'exécution du programme, nous définissons l'environnement : une taille d'affichage de 200 × 200 pixels, une couleur blanche pour les traits et aucun remplissage pour les formes géométriques. Nous y initialisons la liste `tousLesCercles` en spécifiant qu'elle contient pour l'instant zéro élément.

La fonction `mouseReleased()`, qui se trouve en bas du programme (mais qui aurait pu être placée ailleurs, c'est indifférent), et qui est appelée chaque fois que l'on clique avec la souris, ne contient qu'une commande, invoquée sans argument :

```
new cercle()
```

Cela signifie qu'à chaque clic de la souris, un objet de type `cercle` est créé.

L'objet `cercle` n'est pas très compliqué. Il contient trois variables numériques nommées `x`, `y` et `taille`, toutes trois de type décimal (`float`), et deux fonctions, qui sont son constructeur, `cercle()`, et une fonction qui ne renvoie aucune variable et qui est nommée `dessine()`.

Dans le constructeur, nous affectons une valeur à nos variables. La variable `x` prend la valeur de la position horizontale de la souris (`mouseX`). La variable `y` prend la valeur de la position verticale de la souris (`mouseY`). Enfin, la variable taille se voit affecter la valeur 0.1.

Pour finir, notre objet s'ajoute de lui-même à la liste `tousLesCercles` grâce à cette ligne :

```
tousLesCercles = (cercle[]) append (tousLesCercles, this);
```

Nous verrons plus tard en détail l'action de la seconde fonction de notre classe d'objet, nommée `dessine()`.

Le programme contient par ailleurs une fonction `draw()`, fonction redondante qui est exécutée à une cadence régulière. Il contient une boucle dont le but est de parcourir un à un les éléments de la liste `tousLesCercles` pour lui demander d'exécuter son propre script `dessine()`. En effet, en écrivant `tousLesCercles[0].dessine()`, on veut dire que le premier élément (numéro zéro) de la liste `tousLesCercles` doit activer son programme nommé `dessine()`. L'utilisation de la boucle s'avère pratique car elle fonctionne quel que soit le nombre d'éléments que contient la liste `tousLesCercles` (`tousLesCercles.length` signifie : le nombre d'éléments de la liste `tousLesCercles`). Si ce nombre est zéro, aucun appel de fonction ne se lancera.

Le programme `dessine()` commence par vérifier la valeur de la variable taille.

Si cette valeur n'excède pas 500, on lui fait subir deux traitements :

- on la multiplie par 1.01 (ce qui revient à lui ajouter 1 %) ;
- on l'incrémente de 1.

Entre ces deux augmentations, l'une absolue et l'autre proportionnelle, nous obtenons une croissance régulière (+1) qui connaît une petite accélération (+1 %), ce qui donnera à `taille` des valeurs successives telles que : 1.101, 2.112, 3.133, 4.164, 5.206, 6.258 [...] 471.050, 476.760, 482.528, 488.353, 494.237, 500.179. Après la valeur 500.179, puisque nous aurons dépassé la valeur 500, cette partie du programme ne s'exécutera plus.

Nous appliquons enfin cette variable `taille` aux dimensions d'une ellipse que nous dessinons sur l'écran et dont le point d'origine est le point (`x`, `y`), c'est-à-dire l'endroit où on avait cliqué au départ.

Dans la pratique, en exécutant notre programme, nous voyons apparaître un cercle à l'endroit où nous cliquons, et la taille de ce cercle croît de manière apparemment régulière. Lorsqu'il atteint une certaine taille (enfin lorsque sa variable `taille` atteint une certaine valeur), il disparaît.

L'accélération de l'augmentation de la taille des cercles permet de compenser l'impression d'un ralentissement que nous aurions sinon.

Dans cet exemple, nous voyons que chaque occurrence de notre classe `cercle` connaît une croissance indépendante des autres occurrences puisque dépendante de l'instant où elle a été créée. Elles pourraient aussi avoir des couleurs différentes mais nous en resterons là pour ne pas compliquer inutilement notre code.

Ce programme a tout de même un petit défaut : la liste des instances ne cesse de s'allonger et les instances de `cercle` qui ne provoquent plus d'affichage (puisque leur variable `taille` a dépassé une certaine valeur) sont toujours incluses à cette liste.

Nous pouvons vider progressivement `tousLesCercles` en complexifiant un peu la fonction `draw()` :

```
void draw() {
  background(0);
  for(int a=0;a<tousLesCercles.length;a++) {
    tousLesCercles[a].dessine();
  }
  /* si tousLesCercles contient au moins un élément et que la
variable "taille" du premier élément de tousLesCercles a une
valeur supérieure à 500, alors on supprime le premier élément
de tousLesCercles */
  if(tousLesCercles.length>0 && tousLesCercles[0].taille>500)
{
    tousLesCercles = (cercle[]) subset(tousLesCercles, 1);
  }
}
```

Cette méthode fonctionne car tous les cercles ont la même progression, et donc ils doivent être supprimés de la liste en commençant par le premier, à l'aide de la commande `subset()`.

Une seconde méthode nous permet de supprimer toutes les instances qui ont une caractéristique donnée (par exemple, ici, la caractéristique d'avoir une variable nommée `taille` dont la valeur est supérieure à 500).

Nous commençons par créer une nouvelle liste de type `cercle`, nommée `provisoire`.

Ensuite nous parcourons un à un tous les éléments de la liste `tousLesCercles`. Ceux qui n'ont pas dépassé la valeur 500 sont ajoutés à la liste `provisoire`. Pour finir, nous décrétons que le contenu de `tousLesCercles` devient le contenu de `provisoire` :

```
void draw() {
  background(0);
  for(int a=0;a<tousLesCercles.length;a++) {
    tousLesCercles[a].dessine();
  }
  cercle[] provisoire = new cercle[0];
  for(int a=0;a<tousLesCercles.length;a++){
    if( tousLesCercles[a].taille<500){
      provisoire=(cercle[])append(provisoire,tousLesCercles[a]);
    }
  }
  tousLesCercles = provisoire;
}
```

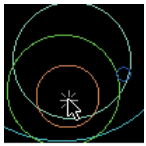
Vous remarquerez que nous parcourons deux fois les éléments de la liste `tousLesCercles` à l'aide d'une boucle. C'est une fois de trop et cela constitue une perte d'énergie pour le programme.

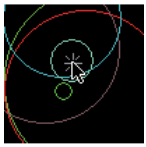
On peut donc effectuer les deux opérations en une seule passe, comme ceci :

```
void draw() {
  background(0);
  cercle[] provisoire = new cercle[0];
  for(int a=0;a<tousLesCercles.length;a++) {
    tousLesCercles[a].dessine();
    if( tousLesCercles[a].taille<500){
      provisoire=(cercle[])append(provisoire,tousLesCercles[a]);
    }
  }
  tousLesCercles = provisoire;
}
```

Revenons à notre classe `cercle`. Nous voulons à présent lui ajouter deux caractéristiques :

- chaque cercle aura une couleur différente ;
- chaque cercle changera de taille à une vitesse différente.






```
class cercle {
  int x,y;
  float taille;
  float vitesse;
  color maCouleur;
  cercle() {
    x=mouseX;
    y=mouseY;
    taille  =0.1;
    vitesse = random(1,2);
    maCouleur = color(random(255), random(255),
    random(255));
    tousLesCercles = (cercle[]) append
    (tousLesCercles, this);
  }
  void dessine() {
    if(taille<500) {
      taille*=1.01;
      taille+=vitesse;
      stroke(maCouleur);
      ellipse(x,y,taille, taille);
    }
  }
}
```

Nous avons commencé par déclarer deux nouvelles variables pour notre classe : `vitesse` (de type nombre décimal) et `maCouleur` (de type color).

Ces variables sont initialisées dans le constructeur d'objet : `vitesse` prend comme valeur un chiffre compris entre 1.0 et 2.0 (par exemple 1.432) et `maCouleur` prend comme valeur n'importe quelle couleur parmi les 2^{24} (soit plus de 16 millions) de couleurs que peut afficher l'ordinateur.

Nous utilisons ces deux variables dans la fonction `dessine()`, où `taille` n'est plus incrémenté de 1 mais de la valeur de la variable `vitesse` (entre 1 et 2) ; quant à la couleur `maCouleur`, elle est appliquée aux contours de formes juste avant le dessin de l'ellipse, grâce à la commande `stroke()`.

Cet exemple somme toute assez simple (cinquante lignes au total) montre bien la puissance de la programmation orientée objet.

Grâce à la programmation objet nous pouvons donc donner des caractéristiques individuelles à un grand nombre d'occurences d'une même classe d'objets. Cela nous permet de créer des systèmes de particules plus ou moins sophistiqués, qui peuvent imiter des phénomènes naturels ou biologiques comme les "systèmes complexes" (circulation des fluides, météorologie, bancs de poissons, nuées d'étourneaux, etc.), ou bien encore des comportements animaux (fourmilière) ou humains (circulation automobile, etc.).

Le programme qui suit reste modeste et propose une manière de simuler des explosions de feux d'artifice (voir Figure 16.1).

Nous n'allons pas le détailler car il est au fond assez proche du programme précédent, mais notons :

- Que chaque particule lumineuse de nos explosions est une instance de la classe `particule`.
- Que les instances de la classe `particule` sont stockées dans une liste nommée `mesPoints`.
- Que lorsque nous cliquons sur l'écran une couleur est créée au hasard et que 200 instances de l'objet `particule` sont créées et à qui cette couleur est attribuée.
- Que chaque instance de particule se voit attribuer une direction au hasard (grâce à un calcul réalisé avec les fonctions cosinus et sinus).
- Que chaque élément de la liste `mesPoints` perd régulièrement en luminosité et que ceux qui sont proches de l'invisibilité sont retirés de la liste.

```
// Feu d'artifices

particule[] mesPoints;
// Liste d'instances de type "particule"

void setup(){
  //smooth();
  size(200, 200);
  background(0);
  mesPoints = new particule[0];
  fill(0,5);
  strokeWeight(2);
  colorMode(HSB);
}

void draw(){
  filter(BLUR, 2) ;
  particule[] newPoints = new particule[0];
  /* à l'issue de la boucle, la liste newPoint
  contiendra les particules actives et remplacera
  la listes mesPoints*/
  for(int a=0;a<mesPoints.length;a++){
    mesPoints[a].dessine();
    if(mesPoints[a].lux>2){
      newPoints = (particule[]) append (newPoints,
      mesPoints[a]);
    }
  }
  mesPoints = newPoints;
  /* fade progressif par application d'un rectangle
  noir transparent*/
  noStroke();
  rect(0,0,width, height);
}

class particule {
  float x, y, vx, vy, lux, teinte;
```

```
  particule(float t){
    teinte = t;
    float angle = random(TWO_PI);
    float vitesse = random(2,10);
    vx = cos(angle)*vitesse;
    vy = sin (angle)*vitesse;
    x=mouseX;
    y=mouseY;
    lux = random(240, 255);
    mesPoints = (particule[]) append(mesPoints, this);
  }
  void dessine(){
    x+=vx;
    y+=vy;
    lux*=0.99;
    stroke(teinte, 255, lux);
    point(x,y);
    vy+=0.1;
    vx*=0.98;
  }
}

void mousePressed(){
  float teinte = random(360);
  for(int a=0;a<200;a++){
    new particule(teinte);
  }
}
```

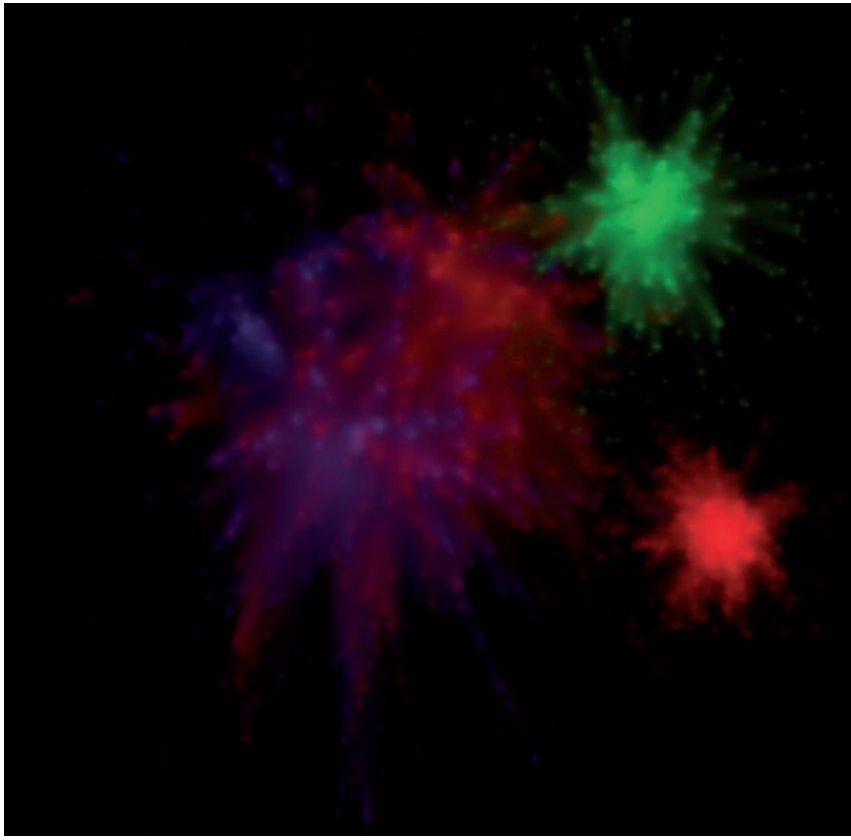


FIGURE 16.1
Système de génération de particules à la constitution d'un feu d'artifice.

16.3 Récursion

On parle de récursion lorsqu'une fonction s'invoque elle-même. Ainsi, si quelqu'un avait la mauvaise idée d'invoquer cette fonction, celle-ci s'invoquerait elle-même, et à nouveau, et à nouveau... Les appels de fonctions récursives sont un sujet très abstrait de prime abord. Nous allons voir un peu plus loin une application très parlante de récursion avec la création d'un arbre dont chaque branche crée deux nouvelles branches qui créent à leur tour deux nouvelles branches, etc.

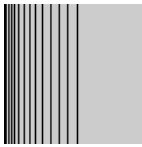
```
void maFonction () {
  maFonction ();
}
```

Mais on peut faire pire avec :

```
void maFonction () {
  maFonction ();
  maFonction ();
}
```

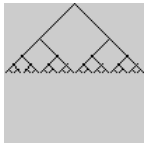
Cette fois-ci, la fonction s'invoque elle-même deux fois, qui chacune s'invoqueront deux fois, etc., ce qui aboutira à une progression exponentielle : 1 appel de fonction, puis 2, puis 4, puis 8, puis 16, 32, 64, 128, 256... au bout de peu de temps, l'ordinateur ne pourra plus suivre. Enfin théoriquement car, heureusement, l'opération est interrompue par Processing qui détecte le problème et qui signale que trop d'opérations récursives sont en cours d'exécution.

On peut placer un petit compteur pour éviter le problème. Ainsi dans l'exemple qui suit nous créons une fonction `trait`, qui dessinera une succession de ligne à un espacement `e`. Dès que cet espacement aura atteint une certaine valeur, la fonction arrêtera de s'invoquer.



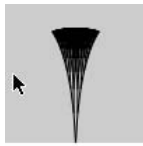
```
void setup(){
  trait(0, 0);
}

void trait (float x, float e){
  // (re)dessine une ligne à l'actuelle valeur
  // de e + x
  line (x + e,  0,  x + e , height);
  if(x + e<=50){
    e+=0.5;
    //ré invoque la fonction "trait"
    trait(x + e,  e);
  }
}
```

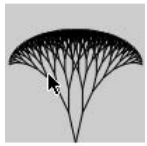
Dans notre second exemple nous écrivons une fonction dite `branche` consistant à tracer deux diagonales descendantes. À chaque récursion, la taille de ces dernières sera réduite de moitié.

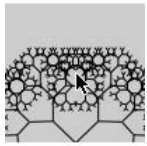


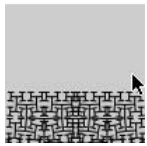
```
void setup(){
  // x, y, taille
  branche(width/2, 0, 25);
}

void branche(float x, float y, float t){
  //segment descendant vers la droite
  line(x, y, x + t , y + t);
  //segment descendant vers la gauche
  line(x, y, x - t , y + t);
  if(t>2){
    branche(x - t , y + t, t/2);
    branche(x + t , y + t, t/2);
  }
}
```

Notre dernier exemple permet, sur le principe précédemment décrit, de réaliser un arbre dont l'angle de distribution de ses branches est ordonné par la souris.









```
float angle_distributif=0.35;

void setup(){
  smooth();
  noFill();
}

void draw(){
  background(204);
  translate(width/2, height);
  //x, y, taille, angle
  branche(0, 0, 30,  angle_distributif);
  angle_distributif=map(mouseX, 0, width, 0,
  HALF_PI);
}

void branche(float x, float y, float t, float ad ){
  t*= 0.75;
  if(t>2){
    //Branche de gauche
    pushMatrix();
    rotate(ad);
    line(0, 0, 0, -t);
    translate(0, -t);
    branche(0, 0, t, ad);
    popMatrix();
    //Branche de droite
    pushMatrix();
    rotate(-ad);
    line(0, 0, 0, -t);
    translate(0, -t);
    branche(0, 0, t, ad);
    popMatrix();
  }
}
```