8. Interactivité avec la souris

8.1 Déplacement

mouseX() et mouseY()

Les coordonnées de la souris sont récupérables par l'intermédiaire des variables système `mouseX` et `mouseY`. Pour la récupération de la valeur de la position de la souris sur l'axe des abscisses (sur l'axe horizontal), nous utiliserons la variable `mouseX`, et pour les ordonnées (l'axe vertical), `mouseY`. Ces données sont relatives au bord supérieur gauche de la fenêtre d'exécution du programme. Si le curseur de la souris se trouve à 10 pixels de la droite de la fenêtre et à 20 pixels du haut de la fenêtre, `mouseX` aura pour valeur 10 et `mouseY` aura pour valeur 20. Pour récupérer en continu ces informations et l'afficher dans la fenêtre de surveillance, nous pouvons écrire un programme comme celui-ci, où l'inscription des valeurs `mouseX` et `mouseY`, séparées par un double-point, est inscrite à chaque cycle du programme, c'est-à-dire à chaque appel de la fonction `draw` :

```
void draw(){
println(mouseX+ " : " +mouseY);
}
```

Pour contraindre un objet particulier à suivre notre souris, et ainsi réaliser un outil de dessin simple, nous pourrions écrire ce programme :

```
void draw(){
  smooth();
  ellipse(mouseX, mouseY, 10, 10);
}
```

Dans cet autre programme, une ligne est tracée à chaque cycle entre la position de la souris et le point supérieur gauche de la fenêtre.

```
void draw(){
 line(0, 0, mouseX, mouseY);
}
```

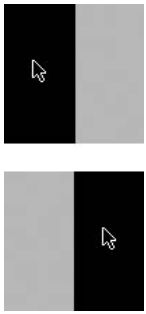
Les valeurs récupérées peuvent servir à un grand nombre d'utilisations. Ainsi nous pouvons donner à un objet une taille proportionnelle à la coordonnée horizontale du curseur de la souris (voir Figure 8.1).

Si son abscisse est 400, le diamètre de notre ellipse sera 200. Si cette abscisse est 200, le diamètre sera 100, etc. :

```
void draw(){
smooth() ;
  float taille= mouseX/2;
  ellipse(mouseX, mouseY, taille, taille);
}
```

Figure 8.1
Illustration obtenue à l'aide d'un outil de dessin dont l'épaisseur du tracé est proportionnel à la coordonnée verticale du curseur de la souris.

L'un des grands intérêts de ces commandes, vous l'aurez deviné, est la possibilité de définir des zones sensibles, pouvant être activées par la position de notre souris. On nomme communément cette action le *roll-over*, ou survol souris. Ainsi, notre premier exemple permet de dessiner un rectangle noir de 50 pixels de largeur sur l'une des moitiés d'écran que parcourt notre pointeur.



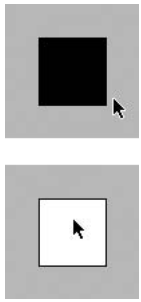
```
void draw(){
  background(204);
  fill(0);
  if (mouseX<width/2){
    rect(0, 0, 50, height);
  }else{
    rect(50, 0, 50, height);
  }
}
```

Ce second exemple, toujours relatif au *roll-over*, utilise une structure itérative afin de disposer dix zones sensibles à intervalle régulier. Dès que la position de la souris correspond aux coordonnées de l'une des zones, celle-ci est dessinée en noir, tandis que les autres, abandonnées par une opération dite *roll-out*, s'effacent progressivement.

```
void draw(){
  noStroke();
  fill(204, 10);
  rect(0, 0, width, height);
  fill(0);
  //Dispose les éléments
  for(int i=0; i<width;i+=10){
    //Vérifie la position de la souris
    if((mouseX>=i)&&(mouseX<=i+10)){
      //Dessine rectangle si la condition est vraie
      rect(i, 0, 10, height);
    }
  }
}
```

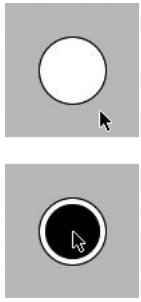
Une zone sensible donc est définie grâce à ses attributs de position et de dimensions. Ici, nous faisons en sorte qu'un carré de 50 pixels situé au centre de notre programme ne change de teinte que si la souris passe sur

une zone correspondant à notre forme. Ce code nécessitera donc de vérifier la validité de 4 conditions :



```
void draw(){
  if ((mouseX>25)&&(mouseX<75) &&
    (mouseY>25) && (mouseY<75)){
    fill(255);
  }else{
    fill(0);
  }
  rect(25, 25, 50, 50);
}
```

Vos zones sensibles, ou boutons, prendront selon vos projets une apparence probablement plus sophistiquée qu'un simple rectangle. Voyons à travers cet exemple comment réaliser un bouton rond, réactif au survol de notre pointeur. Nous emploierons la commande `dist()` afin de vérifier la distance qui nous sépare du centre de notre forme circulaire. Si cette distance est inférieure ou égale au rayon de cette même forme alors nous ordonnerons à notre programme de dessiner un cercle noir au milieu de notre bouton circulaire.



```
//position
int x= 50;
int y= 50;
//Taille
int t=50;
//Distance
float d=100;

void draw(){
  smooth();
  background(204);
  fill(255);
  ellipse(x, y, t, t);
  d= dist(mouseX, mouseY, x, y);
  /* Le diametre étant égal à 50,
  t/2 permet d'obtenir le rayon */
  if (d<=t/2){
    fill(0);
    ellipse(x, y, 40, 40);
  }
}
```

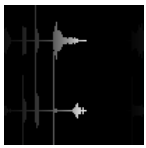
pmouseX() et pmouseY()

Les variables système pmouseX et pmouseY contiennent les coordonnées du curseur tel qu'il était positionné au cycle précédent. Ainsi, si notre curseur se situait aux points 10 sur l'axe des abscisses et 12 sur l'axe de ordonnées il y a 1/20e de seconde, et si la valeur du frameRate (la fréquence d'actualisation) est 20, et qu'il se trouve désormais aux points 15 et 30, la commande pmouseX renverra à 10, pmouseY à 12. Le "p" de pmouseX et pmouseY signifie *previous* (précédent).

En employant ensemble mouseX, mouseY, pmouseX et pmouseY, nous disposons de deux points, ce qui nous permet de dessiner le tracé continu des mouvements du curseur.

```
void draw(){
  line(mouseX, mouseY, pmouseX, pmouseY);
}
```

Nous allons mettre à profit ces variables pour connaître la distance parcourue par le curseur entre deux cycles, et donc, de déterminer la vélocité des déplacements de la souris. Dans l'exemple qui suit, nous figurons la vitesse du déplacement de la souris avec deux graphiques animés. Celui du haut dessine des traits proportionnels à la vélocité latérale de la souris, et celui du bas, des traits proportionnels à sa vélocité verticale.



```
float x=0;

void draw() {
  noStroke();
  fill(0, 10);
  rect(0, 0, width, height);
  stroke(255);
  x = (x + 1)%width;
  //Variable locale déterminant la distance
  //parcourue en x
  float dx = mouseX - pmouseX;
  //Variable locale déterminant la distance
  //parcourue en y
  float dy = mouseY - pmouseY;
  line(x, 25 + dx, x, 25 - dx);
  line(x, 75 + dy, x, 75 - dy);
}
```

dist()

Dans la démonstration précédente, nous avons effectué une opération de calcul nous permettant d'estimer la vélocité de la souris sur une seule échelle normée. Il semble judicieux d'introduire ici une fonction de calcul qui permet de déterminer la distance qui sépare deux points, chacun de ces points, rappelons-le, étant défini par deux éléments. Le premier élément est la position en x (abscisse), le deuxième la position en y (ordonnée). Par commodité, nous nommerons le point d'ancrage du premier x1 et y1, pour le second x2, y2. Puisqu'il est possible de dessiner un rectangle (et donc, deux triangles-rectangles) entre deux points, on peut recourir au théorème de Pythagore pour connaître la distance qui sépare deux points et qui est égale à la racine carrée de x2 moins x1 au carré additionné à y2 moins y1 au carré, soit $\sqrt{((x2 - x1)^2 + (y2 - y1)^2)}$

Suivant cette formule nous serions à même de calculer en pixels la distance qui sépare le bord supérieur gauche de notre programme à son bord inférieur droit, et donc la longueur d'un segment traversant de part en part ce même programme:

```
line(0, 0, width, height);
print(sqrt(((width-0)*(width-0)) + ((height-0)*(height-0))));
// Renvoie la valeur 141.42136
```

Les créateurs de Processing pour éviter aux programmeurs ce calcul ont inclus la commande `dist()` qui nous permet d'obtenir simplement le produit de ce même calcul : `dist(x1, y1, x2, y2)`. Nous pouvons désormais écrire, et donc aisément calculer, la distance qui sépare un point à un autre, par exemple la distance de la souris au milieu de notre scène :

```
void draw() {
  println(dist(mouseX, mouseY, width/2, height/2));
}
```

L'exemple qui suit est similaire à celui que nous avons programmé plus haut, avec `pmouseX` et `pmouseY`, à la différence que nous testons la vélocité générale de la souris à l'aide de la commande `dist()`.

```
float x=0;
void draw() {
  noStroke();
  fill(0, 10);
  rect(0, 0, width, height);
  stroke(255);
  x = (x + 1)%width;
```

```
  float distance = dist(mouseX, mouseY, pmouseX,
  pmouseY);
  line(x, height/2 + distance, x, height/2 - distance);
}
```

Clôturons enfin cette section en réalisant un outil de dessin dont l'épaisseur du trait sera relative à la distance parcourue :

```
void draw() {
  smooth();
  //Variable locale déterminant la distance
  //parcourue en x et y
  float distance = dist(mouseX, mouseY, pmouseX,
  pmouseY);
  strokeWeight(distance);
  line(mouseX, mouseY, pmouseX, pmouseY);
}
```

constrain()

La commande `constrain()` est invoquée avec 3 arguments et permet de contraindre une valeur donnée à un minimum et à un maximum. Elle se rédige ainsi : `constrain(valeur, minimum, maximum)`.

Dans l'exemple qui suit, nous dessinons une ellipse qui suit l'abscisse du curseur de la souris, à condition que cette abscisse ne soit pas inférieure à 20 ni supérieure à 80.

```
void draw(){
smooth() ;
float x = constrain(mouseX, 20, 80);
float y = constrain(mouseY, 20, 80);
ellipse(x, y, 20, 20);
}
```

8.2 Clic

mousePressed() et mouseReleased()

Un clic de souris est décomposée en deux actions distinctes. La première est consécutive de l'action de presser le bouton de la souris. On nomme cette action le demi-clic. Le gestionnaire d'événement `mousePressed()` permet

de récupérer cette information. La seconde est consécutive au relâchement du bouton de notre souris. C'est ce que nous appelons généralement le clic. Le gestionnaire d'événements `mouseReleased()` est invoqué à l'instant où le relâchement a été effectué.

Enfin, la variable `mousePressed`, associée à la structure conditionnelle `if` permet de vérifier si le bouton de notre souris est maintenu appuyé. L'exemple qui suit permet de vérifier dans la fenêtre de surveillance les trois états que nous venons de décrire.

```
void draw(){
  if (mousePressed){
    print("Maintenu appuyé ");
  }
}
void mousePressed(){
  println("Appuyé à la position: "+mouseX);
}
void mouseReleased(){
  println("Relaché à la position: "+mouseX);
}
```

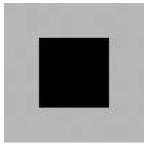
```
void setup(){
  smooth();
}

void draw(){
  if (mousePressed){
    fill(255);
    float d = dist(mouseX, mouseY, pmouseX, pmouseY);
    ellipse(mouseX, mouseY, d*2, d*2);
  }
}

void mousePressed(){
  stroke(0);
  line(mouseX, mouseY, width/2, height);
}

void mouseReleased(){
  noStroke();
  fill(255, 50);
  rect(0, 0, width, height);
}
```

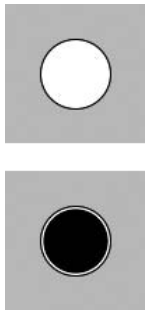
Sur cette base nous pouvons écrire deux programmes relatifs à ces événements souris. Le premier permet de modifier la teinte d'un bouton par un demi-clic, tandis que le second modifiera l'aspect de notre bouton par un clic. Dans ces deux cas de figure nous emploierons une variable booléenne que nous intitulons "clic" qui déterminera l'état du bouton ; à savoir s'il a été préalablement activé ou non. Sa valeur initiale sera `false`.



```
boolean clic=false;

void draw(){
  smooth();
  background(204);
  noStroke();
  fill(0);
  rect(25, 25, 50, 50);
  if (clic==true){
    stroke(255);
    line(30, 30, 70, 70);
    line(30, 70, 70, 30);
  }
}

void mousePressed(){
  if ((mouseX>25) && (mouseX<75) &&
    (mouseY>25) && (mouseY<75)){
    if(clic==false){
      clic=true;
    }else{
      clic=false;
    }
  }
}
```



```
boolean clic=false;
float d=100;

void draw(){
  smooth();
  background(204);
  fill(255);
  ellipse(50, 50, 50, 50);
  d= dist(mouseX, mouseY, 50, 50);
  if (clic==true){
    fill(0);
    ellipse(50, 50, 45, 45);
  }
}

void mouseReleased(){
  if(d<=25){
    if(clic==false){
      clic=true;
    }else{
      clic=false;
    }
  }
}
```

mouseMoved() et mouseDragged()

Il existe deux gestionnaires d'événement qui sont invoqués chaque fois que notre curseur est déplacé :

- `mouseMoved()` Est appelé lorsque la souris bouge mais que l'utilisateur ne clique pas.
- `mouseDragged()` Est appelé lorsque la souris bouge et que le bouton du clic est maintenu.

Dans l'exemple suivant, nous réalisons un programme qui dessine une succession de segments blancs. Leur l'épaisseur est relative à la distance parcourue par la souris lorsque le bouton est maintenu appuyé. Le programme dessine des traits noirs d'un pixel d'épaisseur lorsque le bouton de la souris est relâché.

```
void draw(){
 smooth();
}

void mouseMoved(){
  stroke(0);
  strokeWeight(1);
  line(mouseX,mouseY, pmouseX, pmouseY);
}

void mouseDragged(){
  stroke(255);
  float distance = dist(mouseX, mouseY, pmouseX,
  pmouseY);
  strokeWeight(distance);
  line(mouseX,mouseY, pmouseX, pmouseY);
}
```

mouseButton()

L'instruction `mouseButton()` permet de vérifier quel bouton de la souris est employé. Cette commande est tributaire de 3 arguments :

- `RIGHT`
- `LEFT`
- `CENTER`

```
if (mouseButton== RIGHT){
print("Bouton droit cliqué");
}
```

8.3 Apparence

cursor() et noCursor()

Terminons par les commandes `cursor()` et `noCursor()` qui modifient l'apparence du curseur.

- `cursor()` Permet de choisir le type de pointeur que l'on affiche.
- `noCursor()` Désactive l'affichage du curseur.

L'exemple qui suit montre comment transformer successivement le pointeur, en fonction de la position de la souris sur l'axe horizontal, en une flèche (valeur par défaut), une croix, une main, un pointeur blanc, un signet texte ou un sablier (ou équivalent, selon plate-forme et thème choisi) [voir Figure 8.2].

```
void draw() {
if(mouseX>= 0 && mouseX<10) { cursor(ARROW); }
else if (mouseX>= 10 && mouseX<20) { cursor(CROSS); }
else if (mouseX>= 20 && mouseX<30) { cursor(HAND); }
else if (mouseX>= 30 && mouseX<40) { cursor(MOVE); }
else if (mouseX>= 40 && mouseX<50) { cursor(TEXT); }
else if (mouseX>= 50 && mouseX<60) { cursor(WAIT); }
else
{
noCursor();
}
}
```

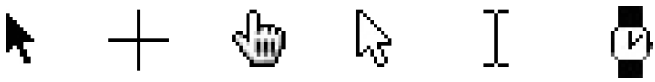


Figure 8.2
Présentation des apparences possibles du pointeur à l'écran.