Compléments de dessin

Transformations, clipping, compositing, hits

autres attributs de Graphics2D

> ***transform***

-> `setTransform(AffineTransform Tx)`

> ***clip***

-> `setClip()`

> ***composite*** détermine comment les pixels du dessin source et les pixels de la destination (a priori l'écran) se combinent pour donner les pixels (couleurs) finalement modifiés sur la destination.

Transformations

> `Graphics2D` contient une méthode `setTransform` prenant en argument un objet de la classe `AffineTransform`.

> La classe `AffineTransform` permet de construire des matrices facilement, par des méthodes statiques:

- `getRotateInstance, getScaleInstance, getShearInstance, getTranslateInstance`

Transformations

> Elles sont définies en coordonnées homogènes:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Clipping

> Tout objet `Shape` peut être utilisé pour définir un contour de clipping: seul ce qui est à l'intérieur de ce contour est dessiné. Le contour de clipping devient un attribut du contexte de l'objet `Graphics2D`. On utilise `Graphics2D.setClip` pour définir le contour de clipping et on lui donne en argument la classe `Shape` qui sert de contour de clipping.

Exemple avec un dessin de texte

> Si on utilise en plus la méthode `clip` de l'objet `Graphics2D`, on obtient un contour de clipping qui est l'intersection des deux (l'ancien clipping et une forme Shape)

Composition

> La classe `AlphaComposite` encapsule la plupart des styles de composite classiques, qui déterminent le rendu d'objets se chevauchant. Elle donne aussi accès à un canal α pour la transparence: $\alpha = 1.0$ opaque, $\alpha = 0.0$ transparent. Les opérations standard de *Porter-Duff* sont gérées.

Composition

> Pour changer le style de composite, créer un objet `AlphaComposite` et le passer à la méthode `setComposite` de l'objet `Graphics2D`.

Porter-Duff: attributs de AlphaComposite

Source over: `SRC_OVER`

Source in: `SRC_IN`

Source out: `SRC_OUT`

Destination over: `DST_OUT`

Porter-Duff: attributs de AlphaComposite

Destination in: `DST_IN`

Destination out: `DST_OUT`

Clear: `CLEAR`

Contrôle de la qualité du rendu

> On peut utiliser les `RenderingHints` de l'objet `Graphics2D` pour déterminer si l'on souhaite un rendu rapide ou de haute qualité.

> Il suffit de construire un objet `RenderingHints` et de le passer en argument de la méthode de `Graphics2D`: `setRenderingHint`.

Contrôle de la qualité du rendu

> Si l'on veut appliquer un seul attribut de rendu, on aussi peut appeler la méthode `setRenderingHint` et lui passer la bonne paire *(attribut, valeur)*. Ces paires sont définies dans la classe `RenderingHints`. Par exemple, pour appliquer systématiquement de l'antialiasing toutes les fois que cela est possible:

Contrôle de la qualité du rendu

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,aliasing);
```


Contrôle de la qualité du rendu

> RenderingHints offre les types suivants de hints:
  – alpha interpolation (default, quality, ou speed),
  – antialiasing (on ou off),
  – color rendering (default, quality ou speed),
  – etc.

Hits

Constructive Area Geometry

> Il s'agit d'une technique permettant de définir une forme plane à l'aide d'opérations booléennes sur des formes simples.

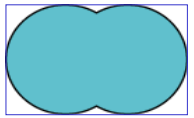
> En Java2D la classe Area supporte les opérations booléennes.

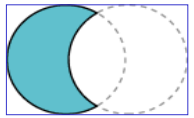
> `public class Area extends Object implements Shape, Cloneable`

Constructive Area Geometry

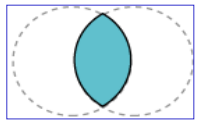
> Opérations booléennes gérées par la classe `Area`:

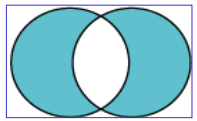
Union





Difference

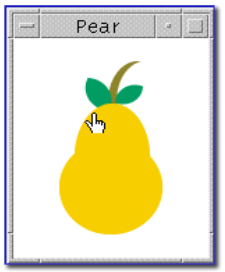




Xor

Constructive Area Geometry

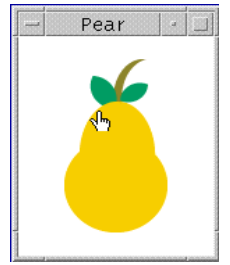
> Exemple: une poire.



Code pour une poire

> Chaque feuille est obtenue par intersection de deux cercles :

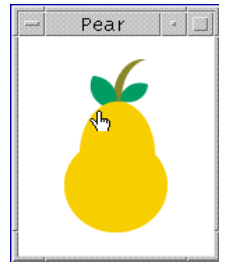
```
Ellipse2D.Double leaf
    = new Ellipse2D.Double();
...
Area leaf1, leaf2;
...
leaf.setFrame(ew-16, eh-29, 15.0, 15.0);
leaf1 = new Area(leaf);
leaf.setFrame(ew-14, eh-47, 30.0, 30.0);
leaf2 = new Area(leaf);
leaf1.intersect(leaf2);
g2.fill(leaf1);
...
leaf.setFrame(ew+1, eh-29, 15.0, 15.0);
leaf1 = new Area(leaf);
leaf2.intersect(leaf1);
g2.fill(leaf2);
```



Code pour une poire

> De même pour la queue:

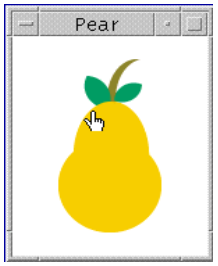
```
Ellipse2D.Double stem
        = new Ellipse2D.Double();
Area st1, st2;
stem.setFrame(ew, eh-42, 40.0, 40.0);
st1 = new Area(stem);
stem.setFrame(ew+3, eh-47,
                    50.0, 50.0);
st2 = new Area(stem);
st1.subtract(st2);
...
g2.fill(st1);
```



Code pour une poire

> Corps de la poire: union d'un cercle et d'un ovale.

```
circle = new Ellipse2D.Double();
oval = new Ellipse2D.Double();
...
circle.setFrame(ew-25, eh, 50.0, 50.0);
oval.setFrame(ew-19, eh-20,
                        40.0, 70.0);
Area circ, ov;
circ = new Area(circle);
ov = new Area(oval);
circ.add(ov);
     ...
g2.setColor(Color.yellow);
g2.fill(circ);
```



Interaction utilisateur-géométrie

> Pour permettre à l'utilisateur d'agir sur ses dessins, il faut lui donner un moyen lui permettant de savoir s'il a cliqué dans l'un d'eux.

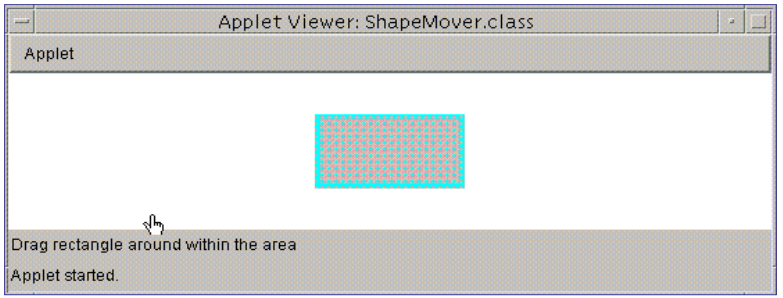
> La méthode `hit` de `Graphics2D` permet de savoir si un clic de la souris s'est produit sur un objet implémentant l'interface `Shape`.

Interaction utilisateur-géométrie

> Autre méthode équivalente: déterminer la position de la souris puis appeler la méthode `contains` sur l'objet `Shape` pour savoir si le clic de la souris est intervenu effectivement à l'intérieur de la boîte englobante de l'objet `Shape`.

Interaction utilisateur-géométrie

> Drag d'une forme géométrique au clic de la souris: déplacer une forme.



Déplacer une forme

```
public void mousePressed(MouseEvent e){
last_x = rect.x - e.getX();
last_y = rect.y - e.getY();
if(rect.contains(e.getX(), e.getY())) updateLocation(e);
...

public void updateLocation(MouseEvent e){
rect.setLocation(last_x + e.getX(), last_y + e.getY());
...
repaint();
```

> En utilisant `Swing`, on peut gérer le réaffichage à l'aide du double-buffering.

BufferedImage

> La classe `Image` représente une image comme un tableau rectangulaire de pixels.

> La sous-classe `BufferedImage` gère en sus les opérations sur les pixels. La classe `Graphics2D` permet ensuite de dessiner l'image à l'écran.

Interface BufferedImageOp

> L'API Java2D propose un certain nombre d'opérations de filtre sur les objets `BufferedImage` .

> Chaque opération de traitement d'image est encapsulée dans une classe qui implémente l'interface `BufferedImageOp`.

Interface BufferedImageOp

> L'opération de traitement d'image est réalisée par invocation de la méthode `filter` de la classe représentant l'opération de traitement d'image.

> L'interface `BufferedImageOp` permet la réalisation des opérations suivantes:

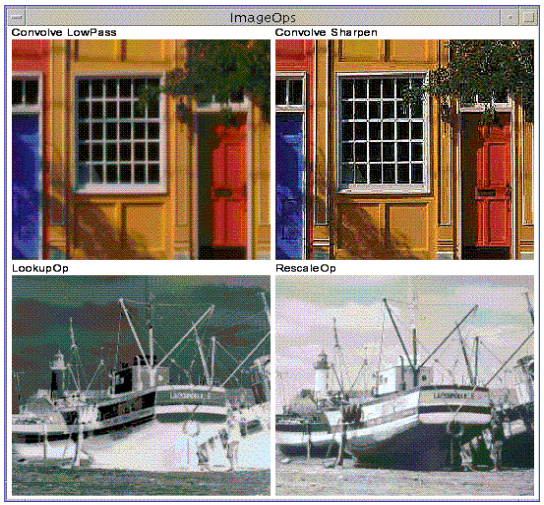
***Transformations affines, corrections de niveaux de gris, modifications de la look-up table, conversion de couleurs, convolution*.**

Comment fait-on ?

> Construire une instance de l'une des classes implémentant l'interface `BufferedImageOp`: `AffineTransformOp`, `BandCombineOp`, `ColorConvertOp`, `ConvolveOp`, `LookupOp`, ou `RescaleOp`.

> Appeler la méthode `filter` de la classe, en lui donnant la `BufferedImage` devant subir l'opération et la `BufferedImage` devant contenir le résultat.

Exemples typiques



Double buffering

> On dessine *off-screen* dans un buffer, et quand le dessin est terminé, on affiche à l'écran ce buffer de travail.

> Systématiquement utilisé par Swing.

> Courant en animation.

> Un objet `BufferedImage` peut être utilisé comme buffer *off-screen*.

Charger des images

Charger une image avec ImageIO

```
BufferedImage img = null;

try {
        img = ImageIO.read(new File("strawberry.jpg"));
}
catch (IOException e) {    }
```


Classe URL

Pour charger une image depuis un fichier distant, on utilise la classe URL

URL signifie Uniform Resource Locator

La classe URL de java.net encapsule 4 données :

> Un protocole d'accès (http ou ftp)
> Un nom de domaine (java.sun.com)
> Un numéro de port
> Un chemin d'accès au fichier

Classe URL

> `URL sourceURL = new URL («http://www.wrox.fr/»);`
> `getCodeBase():` renvoie l'URL de la source où le fichier .class d'une applet est stocké.

```
URL illustre = new URL (getCodeBase(),
«dessin.gif»);
```

> `URL doc = getDocumentBase()`

Classe URL

> `URL sourceURL = new URL («http», «www.ncsa.uiuc.edu», -1,// default port «/demoweb/url-primer.html»);`
> `String getProtocol(), getHost(), getFile(); int getPort().`
> `Boolean equals(URL)`
> `InputStream openStream()// retourne // un InputStream sur le fichier défini par // l'URL`

Charger une image avec ImageIO

```
// si le code tourne dans une applet
BufferedImage img = null;
try {
   URL url = new URL(getCodeBase(),
                     // url du repertoire contenant l'applet
                                "strawberry.jpg");
   img = ImageIO.read(url);
}
catch (IOException e) {    }
```


Sauver une image avec ImageIO

La classe ImageIO permet aussi d'écrire une image dans un fichier:

```
static boolean ImageIO.write(RenderedImage im,
                             String formatName,
                             File output) throws IOException
```


Classe ImageIcon

> 9 constructeurs `ImageIcon()`

> contient aussi une référence à un objet `Image` (de java.awt) récupérable par `getImage()`.

> formats lus : GIF, JPEG, PNG.

- GIF: Graphics Interchange Format. Limité à 256 couleurs.
- JPEG: Joint Photographic Experts Group.Plus perfectionné. Comprime les données avec perte.
- PNG: Portable Network Graphics. Pour des images crées par ordinateur.

Classe ImageIcon

`ImageIcon` utilise un **`MediaTracker`** pour suivre le chargement de l'image. C'est un peu plus simple qu'un `ImageObserver` et en général suffisant.

On crée une instance de `MediaTracker`, on lui demande de suivre une (ou plusieurs images) et on peut l'interroger avec :

> `int getImageLoadStatus()`

```
// ABORTED, COMPLETE, ERRORED, LOADING
```

interface ImageObserver

Pour un suivi plus fin que le mediaTracker, on peut aussi utiliser l'interface `ImageObserver` :

il faut alors implanter une méthode `imageUpdate` et s'assurer de l'enregistrement de l'objet comme observateur (habituellement quand on spécifie un `ImageObserver` à une méthode `drawImage`, voir plus loin).

`imageUpdate` sera appelée chaque fois qu'une nouvelle information sur l'image devient accessible.

```
>void setImageObserver(ImageObserver)
>ImageObserver getImageObserver()
```


Dessiner des images

Dessiner des images

> `drawImage(Image im, AffineTransform xform, ImageObserver observer)`
> Remarque: il existait déjà des méthodes `drawImage(Image im, int x, int y,[int width, int height], [Color background], ImageObserver observer)` dans awt.Graphics.

`L'observer` implémente `imageObserver` et sera notifié quand une nouvelle info sur l'image deviendra accessible. On peut néanmoins passer `null`.

Dessiner des images

> On peut aussi filtrer au préalable l'image avec une opération :
> `drawImage (BufferedImage img, BufferedImageOp op, int x, int y)`

ImageObserver

`imageUpdate(Image img, int infoflags, x, y, width, height)` est la méthode invoquée dans `observer` quand une information requise par une interface asynchrone devient accessible. ex:

`getWidth(ImageObserver), drawImage(img, x, y, ImageObserver)`.

`imageUpdate` retourne true si d'autres appels de mise à jour peuvent encore se produire sur l'image.

imageUpdate(Image img, int infoflags, x, y, width, height)

> `infoflags`: indique quelles informations sont maintenant disponibles sur l'image par un OU bit à bit sur les constantes suivantes:

`WIDTH, HEIGHT, PROPERTIES, SOMEBITS, FRAMEBITS, ALLBITS, ERROR, ABORT.`

> `imageUpdate` retourne false si toutes les informations sur l'image sont maintenant disponibles.

Dessiner dans des images

Dessiner dans une image

```
> private BufferedImage bufim;// dans le JPanel
> private Graphics2D g2D;
> bufim= ImageIO.read(new File("images/0a71.jpg"));
>      g2D = (Graphics2D) bufim.getGraphics();
> protected void paintComponent(Graphics g) {
      super.paintComponent(g);
      Insets inset = getInsets();
      g2D.setColor(Color.blue);
   g2D.fillRect(0,0,100,200);
```

Threads et animations

Threads : Rappels

> Thread = flot d'exécution dans un processus.

Multi-processing ≠ multi-threading

> un seul thread est exécuté à la fois.

> le scheduler (ordonnanceur) java gère la vie des threads avec des procédures qui permettent de passer d'un état à un autre.

Gestion des threads

> start(): lance la méthode run() du thread

> stop(): détruit le thread

> suspend()/resume(), sleep(long[,int]) …

☹ Mais attention: 2 threads accédant à une même donnée doivent être synchronisés.

Threads and Swing

☹ Les composants Swing ne peuvent être accédés que par un thread à la fois.

☞ Or, on a vu qu'un thread swing gère les événements sur les composants quand ils sont réalisés (=après un *appel à setVisible(), show() ou pack()*), donc tout code qui affecte ou dépend d'un composant *réalisé* doit être effectué *dans* ce thread (=via des événements). Donc on ne peut accéder aux composants Swing que dans ce thread, via des événements

> Exceptions: les procédures dites *thread-safe* (elles peuvent être appelées de n'importe où).

par exemple: repaint() ou revalidate()

Threads and Swing

- ☞ On construit l'interface dans le thread principal (`main` ou `init`). Une fois le cadre réalisé (`setVisible, show ou pack`) on ne touche plus aux composants que dans les gestionnaires d'événements.
- ☺ On peut cependant appeler `repaint` ou `revalidate`, modifier la liste des écouteurs, appeler `invokeLater` ou `invokeAndWait` depuis un autre *thread*.

invokeAndWait: exemple 1

showHelloThereDialog est appelée depuis un écouteur d'événement, et elle peut générer une exception dans le thread de dispatching des événements :

```
void showHelloThereDialog() throws Exception {
        Runnable showModalDialog = new Runnable() {
          public void run() {
            JOptionPane.showMessageDialog(myMainFrame,
                                          "Hello There");
          }
        };
        SwingUtilities.invokeAndWait(showModalDialog);
}
```


invokeAndWait: exemple 2

printTextField est appelée depuis un thread qui doit accéder à deux TextField (donc 2 composants Swing).

```
void printTextField() throws Exception {
        final String[] myStrings = new String[2];
        Runnable getTextFieldText = new Runnable() {
                public void run() {
                        myStrings[0] = textField0.getText();
                        myStrings[1] = textField1.getText();
                }
        };
        SwingUtilities.invokeAndWait(getTextFieldText);
        System.out.println(myStrings[0] + " " + myStrings[1]);
}
```


Autre outil: Classe SwingWorker

```
final SwingWorker worker = new SwingWorker() {

        public Object construct() {
          // code long à exécuter
          // il sera lancé dans un thread séparé
          return someValue; // récupérable par get() [bloquant]
        }
        public void finished() {
          // code court à exécuter après construct
          // placé dans le "event-dispatching thread"
        }
};
```

Classe SwingWorker

Exemple : une procédure loadImage qui charge une image peut utiliser un SwingWorker.

```
//      ici, on a des objets Photo noté pic (pour picture)
public void actionPerformed(ActionEvent e) {
        if (icon == null) {
                loadImage(imagedir + pic.filename, current);
        }       // current est un index dans un tableau de Photo
        else {
                updatePhotograph(current, pic);
        }
}
```


Classe SwingWorker

```
private void loadImage(final String imagePath, final int index) {
        final SwingWorker worker = new SwingWorker() {
            ImageIcon icon = null;

             public Object construct() {
                icon = new ImageIcon(getURL(imagePath));
                return icon;   //   obligado mais non utilisé ici
             }
        // (ImageIcon utilise un MediaTracker)  …
        // la procédure finished() sera lancée après construct
```


Classe SwingWorker

```
        // la procédure finished() est lancée dans le
        // "event-dispatching thread" quand construct() est terminée
             public void finished() {
                Photo pic = (Photo)pictures.elementAt(index);
                pic.setIcon(icon);
                if (index == current)
                        updatePhotograph(index, pic);
              }
         }; // end SwingWorker
}       // end loadImage
```


Classe Timer

La classe `Timer` de javax.swing permet de lancer une action après un délai, **via un événement d'action**. (La procédure déclenchée est ainsi lancée depuis le thread de gestion des événements.)

- Cela permet aussi d'effectuer des actions répétées (réarmer le timer avec l'action).
- Utilisée pour contrôler des animations.
- `Timer(int millisec, ActionListener thisObj)`

Classe Timer

- `setDelay(int)`
- `int getDelay()`
- `void setInitialDelay(int)`
- `int getInitialDelay()`
- `void setRepeats(boolean)// True`
- `boolean isRepeats()`
- `void setCoalesce (boolean)// True`
- `boolean isCoalesce()`

Classe Timer

- `void start()`
- `void restart()` : relance le timer. En abandonnant les événements en attente de traitement.
- `void stop()` : stoppe le timer.
- `boolean isRunning()`

Exemple de schéma d'animation

```
public class AnimatorClass
                    ... implements ActionListener {
    int number = -1;
    Timer timer;
    JLabel label; // permet d'afficher le nombre number
        ...
    timer = new Timer(delay, this);
        ...
    public synchronized void startAnimation() {
        ...
        timer.start();
        ...
    }
```


Exemple de schéma d'animation

```
    public synchronized void stopAnimation() {
        … // à appeler sur un clic de souris par ex.
        timer.stop();
        ...
    }
    public void actionPerformed(ActionEvent e) {
        number++;
        label.setText("Animation " + number);
    }

    // Affichage de l'interface utilisateur puis

    startAnimation();
}
```