

Programmation graphique avancée et animations (suite)

Une animation

Inspirée de celle écrite par Arthur van Hoff
qui se trouve à se trouve :

<http://www.javaworld.com/jw-03-1996/animation/Example7Applet.html>

écrite par Arthur van Hoff extrait de son
article sur la programmation d'animations
en Java disponible à :

<http://www.javaworld.com/jw-03-1996/animation>

J'ai ajouté le chargement par le
MediaTracker, une réécriture de `update()`
et `paint()`.

On dispose des 2 images :



2 voitures avancent de droite à gauche sur
l'image du monde.

Une animation (A. V. Hoff)

Netsite:

[Return to article](#)

Moving an Image Across the Screen: Example7Applet



le fichier html contient :

```
<applet code=Example7Applet.class width=200 height=200>  
<param name=fps value=20>  
</applet>
```

applet d'animation

```
import java.awt.*;

public
class Example7Applet extends
java.applet.Applet implements Runnable {
    int frame;
    int delay;
    Thread animator;

    Dimension offDimension;
    Image offImage;
    Graphics offGraphics;

    Image world;
    Image car;

    /**
     * Initialisation de l'applet et calcul du delai
     * entre "trames".
     */
    public void init() {
        String str = getParameter("fps");
        int fps = (str != null) ? Integer.parseInt(str) : 10;
        delay = (fps > 0) ? (1000 / fps) : 100;

        tracker = new MediaTracker(this);
        world = getImage(getCodeBase(),
"world.gif");
        car = getImage(getCodeBase(), "car.gif");
        tracker.addImage(world, 0);
        tracker.addImage(car, 0);
    }
}
```

```
/**
 * La methode start() de l'applet. On crée la
 * thread d'animation et on la lance.
 */

public void start() {
    animator = new Thread(this);
    animator.start();
}

/**
 * Le corps de la thread.
 */
public void run() {
    // stocker la date de lancement
    long tm = System.currentTimeMillis();
    while (Thread.currentThread() == animator) {
        // lance l'affichage de l'animation
        repaint();

        // Delai d'attente ajuste pour avoir la
        // meme attente entre chaque trame.
        try {
            tm += delay;
            Thread.sleep(Math.max(0, tm -
System.currentTimeMillis()));
        } catch (InterruptedException e) {
            break;
        }

        // numero de trame incremente pour
        // pouvoir afficher la trame
        // suivante.
        frame++;
    }
}
```

```
/**
 * La methode stop() de l' applet.
 * Elle arrete la thread et desalloue
 * les entites necessaires au double buffering.
 */
public void stop() {
    animator = null;
    offImage = null;
    offGraphics = null;
}

/** dessine la trame courante. */
public void paint(Graphics g) {
    Dimension d = size();

// Cree le double buffering si ce n'est pas deja
fait.
    if ((offGraphics == null)
        || (d.width != offDimension.width)
        || (d.height != offDimension.height)) {
        offDimension = d;
        offImage = createImage(d.width, d.height);
        offGraphics = offImage.getGraphics();
    }

// efface l'image precedente
offGraphics.setColor(getBackground());
offGraphics.fillRect(0, 0, d.width, d.height);
offGraphics.setColor(Color.black);

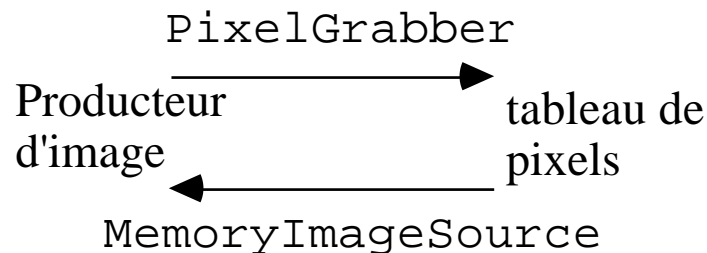
// prepare le dessin de la bonne trame
paintFrame(offGraphics);

// affiche la bonne trame a l'écran
g.drawImage(offImage, 0, 0, null);
}
```

```
public void update(Graphics g) {
    paint(g);
}
/** la creation de la trame :
 * utilise le double buffering et le MediaTracker
 */
public void paintFrame(Graphics g) {
    // Ne faire l'affichage que lorsque
    // toutes les images ont été chargées
    // au besoin provoquer le chargement
    // des images par status(..., true);
    if (tracker.statusID(0, true) ==
MediaTracker.COMPLETE) {
        Dimension d = size();
        int w = world.getWidth(this);
        int h = world.getHeight(this);
        g.drawImage(world, (d.width - w)/2,
(d.height - h)/2, this);
        w = car.getWidth(this);
        h = car.getHeight(this);
        w += d.width;
        //dessine la premiere voiture qui avance de
        // a la fois 5 pixels de droite à gauche
        g.drawImage(car, d.width - ((frame * 5) %
w), (d.height - h)/3, this);
        //dessine la seconde voiture :
        // elle avance de 7 pixels de droite à
gauche
        g.drawImage(car, d.width - ((frame * 7) %
w), (d.height - h)/2, this);
    }
    else { // dans le cas où les images n'ont
        // pas encore été chargées
g.drawString("Images en cours de chargement",
50, 50);
    }
}
}
```

Manipulations d'images

Java propose des classes permettant de manipuler les images. Plus précisément de faire correspondre des tableaux de pixels à des images. Ce sont les classes `PixelGrabber` et `MemoryImageSource`.



Le tableau de pixels est un tableau d'entiers contenant les valeurs RGB des pixels.

La classe `PixelGrabber` implémente l'interface `ImageConsumer`.

La classe `MemoryImageSource` implémente l'interface `ImageProducer`.

Ces classes font partie du paquetage `java.awt.image`.

La classe PixelGrabber

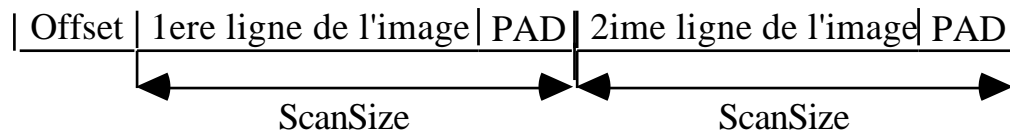
principales méthodes

```
public PixelGrabber(Image img, int x,  
int y, int w, int h, int tabpixels[ ],  
int offset, int scansize)
```

Les pixels de la sous image x, y, w, h de img seront stockés dans le tableau `tabpixels` de sorte que le pixel (i, j) ait sa valeur stockée à

`tabpixels[(j-y)*scansize + (i-x) + offset]`
`offset` donne l'indice où commence véritablement les données pixels.

`scansize` indique la taille d'une ligne de l'image : c'est souvent la largeur de l'image.



```
public boolean grabPixels() throws  
InterruptedException
```

C'est la méthode qui effectue véritablement le chargement des pixels de l'image vers le tableau. L'exception `InterruptedException` est levée si une autre thread interrompt celle-ci.

La classe

MemoryImageSource

principales méthodes

```
public MemoryImageSource(int w, int h,  
int tabpixels[ ], int offset, int  
scansize)
```

construit un producteur d'image à partir du tableau de pixels `tabpixels`.

On obtient un objet de la classe `Image` à partir d'un producteur d'image grâce à la méthode

```
Image createImage(MemoryImageSource)
```

de la classe `Component`.

Un exemple

```
import java.awt.image.*;
import java.applet.*;

public class TraitelImage extends Applet {
    private img, imgtraite;

    public void init() {
        img = getImage(getDocumentBase(),
"bellImage.gif");
        int w = img.getWidth(this);
        int h = img.getHeight(this);
        int tabpix[ ] = new int [w * h];
        PixelGrabber grab = new PixelGrabber(
            img, 0, 0, w, h, tabpix, 0, w);
        try {
            grab.grabPixels();
        } catch (InterruptedException e) {
            System.out.println("Pb dans
grabPixels()");
        }
        // traitement de l'image à l'aide des
// elements du tableau

        MemoryImageSource mem = new
MemoryImageSource(w, h, tabpix, 0, w);
        imgtraite = createlImage(mem);
    }
}
```

Les images filtrées

Java propose un ensemble de classes (dans le paquetage `java.awt.image`) pour faire des "filtres" sur les images. Ce sont les classes :

`FilteredImageSource`, `ImageFilter`,
`RGBImageFilter`

```
public class FilteredImageSource  
implements ImageProducer
```

permet de construire une nouvelle image à partir d'une image existante et d'un filtre.

```
public class ImageFilter implements  
ImageConsumer, Cloneable
```

permet de construire des filtres pour un producteur d'image vers un consommateur d'image. Les filtres seront des objets de classes dérivées de cette classe.

```
public abstract class RGBImageFilter  
extends ImageFilter
```

permet de construire facilement des filtres à l'aide de valeur RGB. Cette classe abstraite contient plusieurs méthodes possédant une définition et une seule méthode abstraite :

```
public abstract int filterRGB(int x,  
int y, int rgb).
```

La classe FilteredImageSource

```
public FilteredImageSource  
(ImageProducer source, ImageFilter  
filtre)
```

construit une image filtrée à partir de le l'objet ImageProducer source à l'aide du filtre filtre.

remarque :

on obtient l'ImageProducer d'un objet img de la classe Image à l'aide de :

```
img.getSource().
```

La classe RGBImageFilter principaux champs

```
protected boolean  
canFilterIndexColorModel
```

Si égal à true, indique que la valeur retournée par la méthode filterRGB() est indépendante de la position x, y du pixel dans l'image.

Exemple

```
// fortement inspiré de Java in a Nutshell
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class GrayButton extends Applet {
    Image img, imagegrise;
    int w, h;

    public void init() {
        img = getImage(getDocumentBase(),
"images/ladybug.gif");
        w = img.getWidth(this);
        h = img.getHeight(this);
        ImageFilter filtre = new FiltreGris();
        ImageProducer prod = new
FilteredImageSource(img.getSource(), filtre );
        imagegrise = createImage(prod);
    }

    public void update(Graphics g) {
        g.drawImage(img, 0, 0, this);
    }

    public boolean mouseDown(Event e, int x,
int y) {
        Graphics legc = this.getGraphics();
        Dimension taileApplet = this.size();
        legc.clearRect(0, 0, taileApplet.width,
taileApplet.height);
        legc.drawImage(imagegrise, 0, 0, this);
        return true;
    }
}
```

programmations des animations le langage java

```
public boolean mouseUp(Event e, int x, int y)
{
    update(this.getGraphics());
    return true;
}

class FiltreGris extends RGBImageFilter {
    public FiltreGris () {
        canFilterIndexColorModel = true; }
    public int filterRGB(int x, int y, int rgb) {
        int a = rgb & 0xff000000;
        int r = ((rgb & 0xff0000) + 0xff0000)/2;
        int g = ((rgb & 0x00ff00) + 0x00ff00)/2;
        int b = ((rgb & 0x0000ff) + 0x0000ff)/2;
        return a | r | g | b ;
    }
}
```

Les boutons images

Le code proposé est en Java 1.0. On veut avoir :

```
public class AppletBoutonImage extends Applet {
    BoutonImage fauxBt1, fauxBt2;

    public void init() {
        fauxBt1 = new BoutonImage( this,
        "ladybug", "ladybug" );
        fauxBt2 = new BoutonImage( this,
        "europe", "europe" );
        add( fauxBt1 );
        add( fauxBt2 );
    }

    public boolean action(Event evt, Object arg )
    {
        if ( evt.target instanceof Button ) {
            if ( arg.equals("europe") )
                System.out.println( "ca vient de la carte
d'europe" );
            if ( arg.equals("ladybug") )
                System.out.println( "ca vient de l'image
de la coccinelle" );
            return true;
        }
        return false;
    }
}
```


Les boutons images (suite)

pour cela on construit :

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

class BoutonImage extends Canvas {
    Image          im, gray;
    Applet         appO;
    int            lg, ht;
    Event          evt;
    Button         Bt;
    String         argument;
    MediaTracker  tracker;

    BoutonImage(Applet appletOrig, String image,
String arg){
        appO          = appletOrig;
        argument      = arg;
        tracker = new MediaTracker( this );

        im =
appO.getImage(appO.getDocumentBase(),"ima
ges/"+image+".gif");
        tracker.addImage( im, 1 );
        try {
            tracker.waitForID( 1 );
        } catch( InterruptedException e ) {
            System.out.println("Erreur dans le
tracker.waitForID( 1 )");
        }
    }
}
```

```
    if ( tracker.isErrorID ( 1 ) )
        System.out.println("Erreur au
chargement de l'image");
    int resulMT;
    resulMT = tracker.statusID(1, false);
    if ((resulMT & MediaTracker.COMPLETE) !=
0)
        System.out.println("SUPER le
chargement de l'image s'est BIEN effectue");

    int l = im.getWidth ( this );
    int h = im.getHeight( this );
    resize( l, h );

    ImageFilter f = new FiltreGris();
    ImageProducer producer = new
FilteredImageSource(im.getSource(), f);
    gray = this.createImage(producer);
    Bt = new Button(argument );
}

public void paint(Graphics g) {
    g.drawImage(im, 2, 2, this);
}

public boolean mouseDown(Event e, int x, int y) {
    Graphics g    = this.getGraphics();
    Dimension d = this.size();
    g.clearRect(0, 0, d.width, d.height);
    g.drawImage(gray, 2, 2, this);
    evt = new Event(Bt, Event.ACTION_EVENT,
argument );
    deliverEvent( evt );
    return true;
}
```

```
public boolean mouseUp(Event e, int x, int y) {
    update(this.getGraphics());
    return true;
}

class FiltreGris extends RGBImageFilter {
    public FiltreGris() {
        canFilterIndexColorModel = true;
    }

    public int filterRGB(int x, int y, int rgb) {
        int a = rgb & 0xff000000;
        int r = ((rgb & 0xff0000) + 0xff0000)/2;
        int g = ((rgb & 0x00ff00) + 0x00ff00)/2;
        int b = ((rgb & 0x0000ff) + 0x0000ff)/2;
        return a | r | g | b;
    }
}
```

On a associé à notre image, un **Button Java** nécessaire lorsqu'on veut gérer les événements par le constructeur

`Event(Object target, int id, Object arg)` qui crée un événement Java où `target` sera le composant graphique initiateur, `id` le type de l'événement et `arg` l'argument (qu'on peut récupérer dans `action(...)`)

Observer/Observable

La classe `Observable` et l'interface `Observer` implantent le modèle MVC ou plus précisément Model-View de Smalltalk.

A chaque objet de `Observable` est associé une liste d'objets d'une classe implémentant `Observer`. Ces objets sont informés de changement lorsque l'`Observable` lance la méthode `notifyObservers()` (après avoir indiqué un tel changement par `setChanged()`). Les observers associés lanceront leur méthode `update()`

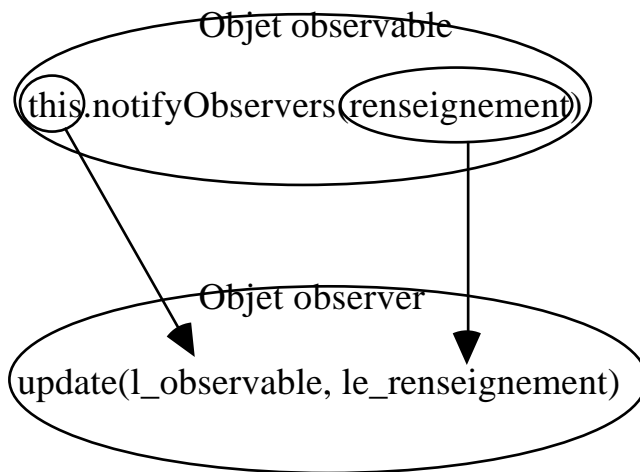
Observer

`java.util.Observer` est une interface
`public interface Observer` qui contient l'unique signature :
`public abstract void update(Observable o, Object arg)` méthode appelée lorsque l'instance appartient à la liste des observers de l'objet `o`.
`arg` est un paramètre permettant de passer une information de l'`Observable` à ses `Observer` associés.

Observable

java.util.Observable est une classe
Une notification de changement dans
l'instance est passée aux Observer associés
en lançant la méthode notifyObservers()
(qui lancera la méthode update() sur
chaque Observer associé).

On a donc :



Observable (suite)

Principales méthodes

Un seul constructeur : `Observable()`

`addObserver(Observer)` ajoute un observer à la liste des observers de l'instance.

`countObservers()` retourne le nombre d'observers.

`deleteObserver(Observer)` enlève de l'instance l'observer référencé par l'argument.

`deleteObservers()` enlève tous les observers de l'instance.

`hasChanged()` retourne `true` si un changement est survenu dans l'instance.

`notifyObservers()` indique aux observers qu'un changement est survenu.

`notifyObservers(Object)` idem que la méthode précédente où on passe une donnée référencée par l'argument aux observers.

`setChanged()` positionne le drapeau de l'instance indiquant qu'un changement s'est produit.

`clearChanged()` marque l'instance comme inchangée.

rappel important

Il faut utiliser `setChanged()` avant `notifyObservers()` pour prendre en compte les changements.

Observer/Observable (suite)

Un exemple

```
import java.util.*;

public class ListeDesConnectes extends Observable {
    private Hashtable LesConnectes = new Hashtable();

    public void login(String nom, String motPasse) throws
    BadUserException
    {
        // test du mot de passe
        if (!MotDePasseValid(nom, motPasse))
            throw new MotDePasseException(nom);

        // on a besoin de 2 arguments pour put() de l'objet
        // LesConnectes de la classe java.util.Hashtable
        Utilisateur utilaux = new Utilisateur(nom);
        LesConnectes.put(nom, utilaux);

        // notifyObservers() n'informe les Observers
        // que si il y a eu des changements et il y a
        // des changements que si on les fait
        // explicitement par setChanged()
        setChanged();
        notifyObservers(utilaux);
    }

    public void logout(Utilisateur unUtil) {
        // utilisation de remove() de Hashtable
        LesConnectes.remove(unUtil.getNom());
        setChanged();
        notifyObservers(unUtil);
    }

    public Hashtable getLesConnectes() {
        return LesConnectes;
    }
}
// ...
}
```

programmations des animations le langage java

```
public class Controleur implements Observer {
    ListeDesConnectes liste;

    public Controleur(ListeDesConnectes tousLesConnectes) {
        liste = tousLesConnectes;
        // au moment de la création (i.e. dans le constructeur),
        // on prévient l'observable qu'on est la pour l'observer
        // et qu'il nous informe de ses changements
        liste.addObserver(this);
    }

    public void update(Observable lesConnectes, Object
renseignements)
    {
        Utilisateur unUtil = (Utilisateur) renseignements;
        if (liste.getLesConnectes().contains(unUtil))
            ajouterUtilisateur(unUtil);
        else
            enleverUtilisateur(unUtil);
    }
}
```


Animation : Neko le chat

C'est une animation graphique écrite à l'origine pour macintosh par Kenjo Gotoh en 1989. Elle montre un petit chat (Neko en japonais), qui court de gauche à droite, s'arrête, baille, se gratte l'oreille, dort un moment puis sort en courant vers la droite. On utilise pour cela les images suivantes :



Le code de l'animation

Neko

```
import java.awt.Graphics;
import java.awt.Image;
import java.awt.Color;

public class Neko extends java.applet.Applet
implements Runnable {

    Image nekopics[] = new Image[9];
    String nekosrc[] = { "right1.gif", "right2.gif",
"stop.gif", "yawn.gif", "scratch1.gif", "scratch2.gif",
"sleep1.gif", "sleep2.gif", "awake.gif" };
    Image currentimg;
    Thread runner;
    int xpos;
    int ypos = 50;

    public void start() {
        if (runner == null) {
            runner = new Thread(this);
            runner.start();
        }
    }

    public void stop() {
        if (runner != null) {
            runner.stop();
            runner = null;
        }
    }
}
```

```
public void run() {
    // chargement des images
    for (int i=0; i < nekopics.length; i++) {
        nekopics[i] = getImage(getCodeBase(),
"images/" + nekosrc[i]);
    }
    setBackground(Color.white);
    int milieu = (this.size().width) / 2;
    System.out.println("milieu = " + milieu);

    // court de gauche a droite
    flipflop(0, milieu, 150, 1, nekopics[0],
nekopics[1]);

    // s'arrete
    flipflop(milieu, milieu, 1000, 1, nekopics[2],
nekopics[2]);

    // baille
    flipflop(milieu, milieu, 1000, 1, nekopics[3],
nekopics[3]);

    // se gratte 4 fois
    flipflop(milieu, milieu, 150, 4, nekopics[4],
nekopics[5]);

    // dort i.e. ronfle 5 fois
    flipflop(milieu, milieu, 250, 5, nekopics[6],
nekopics[7]);

    // se reveille
    flipflop(milieu, milieu, 500, 1, nekopics[8],
nekopics[8]);

    // et part
    flipflop(milieu, this.size().width + 10, 150, 1,
nekopics[0], nekopics[1]);
}
```

programmations des animations le langage java

```
void flipflop(int debut, int end, int timer, int
nbFois, Image img1, Image img2) {
    for (int j = 0; j < nbFois; j++) {
        for (int i = debut; i <= end; i+=10) {
            this.xpos = i;
            // swap images
            if (currentimg == img1)
                currentimg = img2;
            else if (currentimg == img2)
                currentimg = img1;
            else currentimg = img1;

            repaint();
            pause(timer);
        }
    }
}

void pause(int time) {
    try { Thread.sleep(time); }
    catch (InterruptedException e) { }
}

public void paint(Graphics g) {
    g.drawImage(currentimg, xpos, ypos,this);
}
}
```

Les sons dans les applets

Java propose la classe

`java.applet.AudioClip` permettant de manipuler les sons.

Pour l'instant les seuls fichiers sons disponibles (jusqu'à Java 1.2 et Java media Framework) sont les `.au`, un format développé par Sun et jouable sur diverses plateformes.

On peut charger facilement des sons dans une applet grâce aux méthodes

```
public AudioClip getAudioClip(URL url)
```

ou

```
public AudioClip getAudioClip(URL url,  
String nom)
```

de la classe `Applet`.

Après avoir récupéré un objet de la classe `AudioClip` on peut utiliser les 3 méthodes de cette classe : `play()`, `loop()`, `stop()`.

Il est conseillé de lancer la méthode `stop()` de la classe `AudioClip` dans la méthode `stop()` de la classe `Applet`.

Programme de Sons

```
import java.awt.Graphics;
import java.applet.AudioClip;

public class AudioLoop extends
java.applet.Applet implements Runnable {

    AudioClip bgsound;
    AudioClip beep;
    Thread runner;

    public void start() {
        if (runner == null) {
            runner = new Thread(this);
            runner.start();
        }
    }

    public void stop() {
        if (runner != null) {
            if (bgsound != null)
                bgsound.stop();
            runner.stop();
            runner = null;
        }
    }
}
```

```
public void init() {
    bgsound = getAudioClip(getCodeBase(),
"audio/loop.au");
    beep = getAudioClip(getCodeBase(),
"audio/beep.au");
}

public void run() {
    if (bgsound != null) bgsound.loop();
    while (runner != null) {
        try { Thread.sleep(5000); }
        catch (InterruptedException e) { }
        if (bgsound != null) beep.play();
    }
}

public void paint(Graphics g) {
    g.drawString("Execution de musique....", 10,
10);
}
}
```

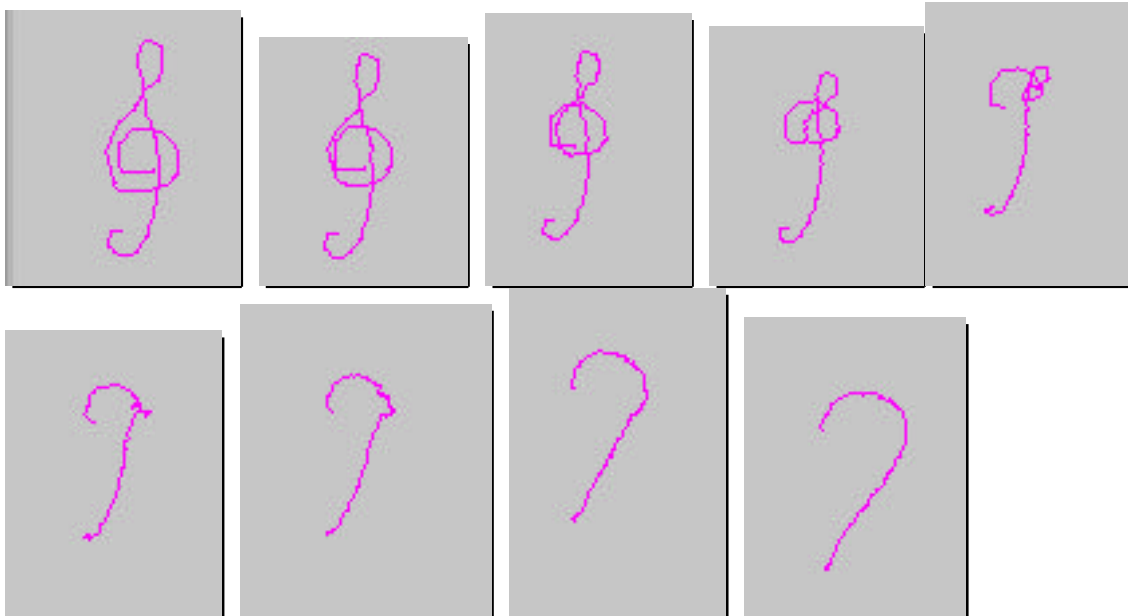
Morphing 2D en Java

Une applet écrite par Pat Niemeyer

(pat@pat.net) utilisable à :

<http://www.ooi.com/tween/editor.html>

On fait 2 dessins et le programme passe de l'un à l'autre par "morphing".



"... feel free to use it for your class
but please don't redistribute it"

code morphing P. Niemeyer

L'applet utilise essentiellement 2 classes dans le paquetage tween : la classe Editor et la classe Canvas. L'applet présente tout d'abord un éditeur de dessin :



dont une partie du code est :

```
package tween;
...
class Editor extends Panel {
    tween.Canvas canvas;
    Graphics canvasGr;
    int xpos, ypos, oxpos, oypos;
    private int [][] track = new int [2048][2];
    private int trackEnd = 0;
    private Vector tracks = new Vector();
    static boolean standalone;

    Editor() {
        ...
        add( "Center", canvas = new tween.Canvas() );
        Panel p = ...
        p.add( new Button("Clear") );
        p.add( new Button("Tween") );
        add( "South", p );
    }
}
```

```
final public boolean handleEvent( Event e ) {
    int x = e.x, y = e.y;
    if ( e.id == Event.MOUSE_DRAG ) {
        xpos = x; ypos = y;
        addPoint( x, y );
        if ( canvasGr == null )
            canvasGr = canvas.getGraphics();
        canvasGr.setColor( Color.red );
        canvasGr.drawLine( oxpos, oypos,
oxpos=xpos, oypos=ypos );
        return true;
    } else
    if ( e.id == Event.MOUSE_DOWN ) {
        oxpos = x; oypos = y;
        trackEnd = 0;
        addPoint( x, y );
        return true;
    } else
    if ( e.id == Event.MOUSE_UP ) {
        int l;
        int [][] tr = new int [ trackEnd ][2];
        for ( int i=0; i< tr.length; i++) {
            tr[i][0] = track[i][0];
            tr[i][1] = track[i][1];
        }
        tracks.addElement( tr );
        return true;
    }
    return true;
}

public boolean action( Event e, Object o ) {
    ••• if ( o.equals("Tween") ) {
        canvas.setTracks( tracks );
        canvas.startTweening(); •••
    }
    return true;
}

private void addPoint( int x, int y ) {
    try {
        track[trackEnd][0] = x;
        track[trackEnd][1] = y;
        trackEnd++;
    } catch ( ArrayIndexOutOfBoundsException e2 ) {
        System.out.println("too many points!!!");
        trackEnd = 0;
    }
}
}
```

morphing P. Niemeyer (suite)

L'éditeur est un code classique de "rubber band" avec les 3 cas à traiter de bouton souris : appui, déplacer bouton enfoncé, relachement. À chaque manipulation, le code remplit un tableau `track` (indice `trackEnd` incrémenté) qui va "contenir la courbe tracé par l'utilisateur".

Lorsque le tracé est fini ce tableau est alors mis dans le vecteur `tracks` qui constitue alors le premier élément du vecteur. Le tracé suivant sera le second élément du vecteur, (etc. car on peut mettre plusieurs tracés).

Un tracé est constitué d'un ensemble de points qui ont eux même 2 coordonnées d'où la déclaration de tableau a deux indices de la forme : `track[numeroDuPoint][0` pour `x`, `1` pour `y]`

morphing P. Niemeyer (suite)

La seconde classe importante est

tween.Canvas :

```
package tween;

import java.awt.*;
import java.util.Vector;
import java.io.*;

class Canvas extends java.awt.Canvas implements Runnable
{
    private Vector tracks = new Vector();
    private Thread runner;
    private Image drawImg;
    private Graphics drawGr;
    private boolean loop;

    void setTracks( Vector tracks ) {
        if ( runner != null )
            stopTweening();
        this.tracks = tracks;
    }

    synchronized public void startTweening( ) {
        if ( runner != null )
            stopTweening();
        if ( tracks == null || tracks.size() == 0 )
            return;
        runner = new Thread( this );
        runner.setPriority( runner.getPriority() + 1 );
        this.loop = false;
        runner.start();
    }

    synchronized public void stopTweening() {
        if ( runner == null )
            return;
        runner.stop();
        runner = null;
    }

    public void run() {
        do {
            tweenTracks();
        } while ( loop );
    }
}
```

```
private void tweenTracks() {
    int n = tracks.size();
    for (int i=0; i<n-1; i++) {
        tweenTrack( (int [][])tracks.elementAt(i),
                    (int [][])tracks.elementAt(i+1) );
    }
    // draw the final track precisely...
    // (we've tweened "up to" it)
    clearGr();
    drawTrack( (int [][])tracks.elementAt(n-1),
Color.magenta );
}

private void tweenTrack(
    int [][] fromTrack, int [][] toTrack ) {
    int tweens = averageDistance( fromTrack, toTrack
)/2;
    if ( tweens < 1 )
        tweens = 1;
    for ( int tweenNo=0; tweenNo < tweens;
tweenNo++) {
        int len = (int)( fromTrack.length
+ 1.0*(toTrack.length -
fromTrack.length)/tweens * tweenNo );
        int [][] tweenTrack = new int [len][2];

        for ( int i = 0; i < tweenTrack.length; i++ ) {
            int from =
(int)(1.0*fromTrack.length/tweenTrack.length * i);
            int to =
(int)(1.0*toTrack.length/tweenTrack.length * i);
            int x1 = fromTrack[from][0];
            int x2 = toTrack[to][0];
            int y1 = fromTrack[from][1];
            int y2 = toTrack[to][1];

            tweenTrack[i][0] = (int)(x1 + 1.0*(x2-
x1)/tweens * tweenNo);
            tweenTrack[i][1] = (int)(y1 + 1.0*(y2-
y1)/tweens * tweenNo);
        }

        clearGr();
        drawTrack( tweenTrack, Color.magenta );
        try {
            Thread.sleep( 1000/24 );
        } catch ( Exception e ) { }
    }
}
```

```
/*
    Sample 1/10 of the (average) number of points to
    calculate an
    average overall distance
*/
private int averageDistance( int [][] track1, int [][] track2 )
{
    int averages = (track1.length + track2.length)/2/10;
    if ( averages < 3 )
        averages = 3;
    int t = 0;
    for ( int i=0; i< averages; i++ ) {
        int [] p1 = track1[ (int)( 1.0 * track1.length /
averages * i ) ];
        int [] p2 = track2[ (int)( 1.0 * track2.length /
averages * i ) ];
        int dx = p2[0] - p1[0];
        int dy = p2[1] - p1[1];
        t += (int)Math.sqrt( dx*dx + dy*dy );
    }
    return t/averages;
}

public void update( Graphics g ) {
    paint(g);
}

public void paint( Graphics g ) {
    if ( drawImg == null )
        return;
    g.drawImage(drawImg, 0, 0, null);
}

public void drawTrack(int [][] track, Color color ) {
    Graphics gr = getOffScreenGraphics();
    gr.setColor( color );

    for ( int i=0; i < track.length-1; i++ )
        gr.drawLine( track[i][0], track[i][1],
            track[i+1][0], track[i+1][1] );

    repaint();
}
```

programmations des animations le langage java

```
private void clearGr() {
    getOffScreenGraphics().clearRect(0, 0,
size().width, size().height);
}

public Graphics getOffScreenGraphics() {
    if ( drawGr == null ) {
        drawImg = createImage( size().width,
size().height );
        drawGr = drawImg.getGraphics();
    }
    return drawGr;
}
```

classe `tween.Canvas` P. Niemeyer

Dans l'éditeur après appui sur le bouton `tween`, la méthode `setTracks()` est lancée puis "l'animation morphing" par `startTweening()`.

`setTracks()` positionne le champ `tracks` qui est le vecteur des formes à "joindre" par morphing.

`startTweening()` crée la thread de morphing dont le corps est `tweenTracks()`.

`tweenTracks()` va faire le morphing entre les 2 formes finales en parcourant le vecteur des formes finales (on peut supposer que ce vecteur n'a que 2 éléments !!).

Morphing (P. Niemyer)

Finalemment c'est donc la méthode

```
private void tweenTrack(int [][]  
fromTrack, int [][] toTrack) qui est la  
clé de tout l'ensemble.
```

Cette méthode cherche d'abord la distance moyenne entre les 2 courbes. Cette distance donne alors le nombre de courbes intermédiaires à dessiner. Il faut alors construire et dessiner chaque courbe intermédiaire. Pour une courbe donnée (i.e. pour une valeur de `tweenNo`, on construit cette courbe à `len` points. `len` est calculé de sorte à être proche du nombre de point de la courbe finale pour les dernières courbes et vice-versa pour les premières courbes.

On repère les points correspondants au 2 courbes finales pour le point à construire, puis on construit ce point ce qui initialise les 2 composantes de `tweenTrack[i]`.

Les dessins sont fait en double buffering et on dessine 24 courbes par secondes.

Bibliographie

<http://www.javaworld.com/javaworld/jw-03-1996/jw-03-animation.html>

<http://java.sun.com:81/applets/Fractal/1.0.2/example1.html>

Teach yourself Java in 21 days : Laura Lemay, Charles L.Perkins ; ed Sams.net traduit en français ed S&SM "Le programmeur Java"