SMCanvas : augmenter la boîte à outils Java Swing pour prototyper des techniques d'interaction avancées

Caroline Appert & Michel Beaudouin-Lafon

Laboratoire de Recherche en Informatique (LRI) & INRIA Futurs [1]
Bâtiment 490 - Université Paris-Sud
91405, Orsay Cedex, France
appert@lri.fr, mbl@lri.fr

RESUME

Cet article présente SMCanvas, une extension de la boîte à outils Java Swing destinée au prototypage et à l'enseignement de l'interaction graphique. SMCanvas utilise un graphe de scène simplifié pour le rendu graphique et des machines à états pour décrire l'interaction. L'utilisation des principes de réification et de polymorphisme permet de combiner simplicité d'usage et puissance d'expression. Nous décrivons l'utilisation de SMCanvas par des étudiants de Master pour programmer des techniques d'interaction avancées, et proposons d'utiliser des benchmarks pour évaluer les outils de développement d'interfaces.

MOTS CLES : Boîte à outils, Widgets, Machine à états, Polymorphisme, Réification, Java Swing.

ABSTRACT

This article presents SMCanvas, an extension of the Java Swing toolkit dedicated to prototyping and teaching graphical interaction. SMCanvas uses a simplified scene graph for rendering and state machines for interaction. The use of polymorphism and reification helps combine ease of use and power of expression. We describe our experience of using SMCanvas with Master level students for programming advanced interactions, and propose to evaluate user interface tools with benchmarks.

CATEGORIES AND SUBJECT DESCRIPTORS:
D.2.2 [Design tools and Techniques]: User Interfaces; H.5.2 [Information Interfaces and Presentation]: User Interfaces – Graphical User Interfaces.

GENERAL TERMS: Design, Human factors

KEYWORDS: Toolkit, Widget, State machine, Polymorphism, Reification, Java Swing.

[1]Projet in | situ | – PCRI, Pôle Commun de Recherche en Informatique du Plateau de Saclay – CNRS, Ecole Polytechnique, INRIA, Université Paris-Sud.

INTRODUCTION

Les développeurs d'applications interactives graphiques utilisent aujourd'hui des boîtes à outils d'interface fondées sur le modèle classique du *widget* [18]. Ces bibliothèques offrent un jeu de widgets couvrant les interactions standard telles que menus et boutons, et permettent de construire une interface graphique en assemblant ces widgets. De nombreuses boîtes à outils ont été développées par la recherche afin d'augmenter leur puissance d'expression, notamment pour couvrir les nouvelles techniques d'interaction telles que les *marking menus* [13], les *magnetic guidelines* [2] ou l'interaction gestuelle. Malheureusement, ces boîtes à outils sont restées expérimentales tandis que les boîtes à outils industrielles ne permettent toujours pas de mettre en œuvre ces nouvelles techniques d'interaction, au détriment de l'utilisabilité des applications graphiques.

Nous présentons dans cet article une approche différente qui consiste à augmenter une boîte à outils industrielle, en l'occurence Java Swing, afin de permettre la mise en œuvre de nouvelles techniques d'interaction avec les outils habituellement utilisés par les développeurs. L'augmentation d'un outil existant nécessite quelques compromis comme l'impossibilité de descendre à un niveau d'abstraction inférieur à celui de l'outil et le respect de ses schémas de conception. En contrepartie, l'avantage escompté est de faire adopter par les développeurs de nouveaux concepts et des outils d'interface avancés tout en capitalisant les connaissances dont ils disposent. L'objectif est d'avoir ainsi un impact réel sur le développement des applications, l'outil augmenté s'intégrant dans un environnement connu des développeurs. Conscients du fait que ce but est ambitieux puisqu'il requiert en particulier de fournir des outils de qualité industrielle, nous nous intéressons dans un premier temps à deux catégories particulières d'usages qui ne nécessitent pas des outils aussi robustes : le *prototypage* d'interfaces et l'*enseignement* de l'IHM. Ceci nous permet d'une part d'exposer et d'accoutumer les développeurs et futurs développeurs à des techniques d'interaction et des outils de développement de nouvelle génération, d'autre part d'évaluer notre extension en la soumettant à un usage *in situ*.

Le reste de cet article présente un état de l'art des boîtes à outils d'interface, introduit les principes qui ont guidé la conception de notre extension, appelée SMCanvas, puis décrit celle-ci. Enfin nous abordons l'évaluation de cette approche et concluons avec quelques perspectives.

ETAT DE L'ART

Il existe de nombreuses boîtes à outils d'interface, aussi nous listons ici seulement celles qui ont directement inspiré notre travail. Au premier rang de celles-ci figure Tcl/Tk [21], dont nous avons beaucoup utilisé le widget *canvas* pour le prototypage et l'enseignement. Le principe de ce widget, la puissance d'expression des *tags* et la concision de la syntaxe ont été repris dans SMCanvas en relevant le défi du passage d'un langage de scripts (Tcl) à un langage à objets typé (Java). GmlCanvas [4] est une extension à Tcl/Tk qui reprend également le principe du *canvas* Tk en augmentant son modèle graphique grâce à un rendu OpenGL. Notre approche est similaire à celle de GmlCanvas dont nous reprenons des éléments du modèle graphique, mais sur la base d'une boîte à outils plus largement répandue chez les développeurs et en ajoutant un modèle élaboré pour la gestion de l'interaction. Ce modèle est basé sur l'utilisation de machines à états, un modèle ancien [9] tombé en désuétude et récemment repris par HsmTk [6]. Par ailleurs, nous exploitons les capacités du langage hôte afin de rendre la syntaxe simple et efficace, à l'image de la boîte à outils Ubit [16] qui utilise la surcharge des opérateurs en C++.

SubArctic [12] est une boîte à outils extensible, en Java, qui suit une longue tradition de boîtes à outils expérimentales basées sur les widgets telles que InterViews [15], Garnet et Amulet [19]. A l'inverse de SMCanvas, aucune ne s'appuie sur une boîte à outils existante. Par ailleurs, outre Java Swing, GTk [17] et Qt [8] sont largement utilisées pour le développement d'interfaces, de même que les boîtes à outils natives de MacOS et Windows. Mais ces boîtes à outils n'offrent qu'un jeu de widgets réduit au vocabulaire classique des interfaces graphiques et sont difficilement extensibles. Enfin, à notre connaissance, la seule boîte à outils d'interface spécifiquement développée pour l'enseignement est SUIT [22].

APPROCHE DE CONCEPTION

Notre objectif de conception est de satisfaire la fameuse phrase d'Alan Kay : *simple things should be simple, complex things should be possible*. A cet effet, nous avons utilisé comme guide de conception deux principes issus des langages de programmation, qui ont également été employés dans le cadre de la conception d'interfaces [1] : le *polymorphisme* et la *réification*. La réification consiste à transformer des concepts abstraits en objets concrets, tandis que le polymorphisme consiste à faire en sorte que des fonctions, objets ou méthodes puissent être utilisés dans différents contextes. Ces principes se prêtent particulièrement bien aux langages à objets puisque d'une part la conception par objets consiste précisément à identifier les classes d'objets pertinentes pour un problème donné et d'autre part les langages à objets offrent plusieurs formes de polymorphisme. Les sections suivantes illustrent comment ces principes sont mis en œuvre, notamment avec les "tags" et les machines à états.

Comme notre approche consiste à compléter une boîte à outils existante plutôt que d'en créer une nouvelle, nous devons atteindre un bon compromis entre l'intégration dans l'environnement hôte d'une part et la puissance d'expression de l'extension proposée d'autre part. L'environnement hôte est constitué du langage de programmation (ici, Java), de la boîte à outils existante (ici, Swing) et des bibliothèques qu'elle utilise (ici, notamment, la bibliothèque graphique Java2D). Nous nous sommes donc attachés à exploiter au mieux la syntaxe et la sémantique du langage Java pour représenter des constructions complexes de façon concise et expressive. D'autre part nous avons utilisé autant que possible les fonctionalités existantes, notamment la couche graphique Java2D, afin de capitaliser au mieux les connaissances des développeurs.

Notre extension se présente sous la forme d'une nouvelle classe de widget Swing appelée *SMCanvas*, qui peut être utilisée comme n'importe quelle autre classe de widgets. Nous nous sommes également attachés à rendre SMCanvas *ouvert* et *extensible* afin de pouvoir notamment aisément enrichir le modèle graphique et le vocabulaire d'actions, comme décrit plus loin. Les deux sections suivantes décrivent le modèle graphique qui définit l'aspect visuel du widget et le modèle d'interaction qui définit la manière dont il réagit aux actions de l'utilisateur.

LE MODELE GRAPHIQUE DE SMCANVAS

Comme indiqué dans l'état de l'art, le modèle graphique de SMCanvas est largement inspiré des *canvas* de Tcl/Tk et de GmlCanvas, notre contribution principale sur ce point étant l'intégration dans l'environnement Java.

Les objets graphiques

Un widget SMCanvas contient un ensemble d'*objets graphiques* stockés dans une *liste d'affichage*. Tous les objets graphiques héritent de SMShape, qui décrit par des attributs de style et de géométrie une forme graphique stockée dans un objet Shape de Java2D. Le rendu graphique affiche les objets dans l'ordre de la liste d'affichage, le dernier objet étant donc au-dessus des autres.

Les classes d'objets graphiques prédéfinies sont `SMEllipse`, `SMRectangle`, `SMSegment`, `SMPolyline`, `SMText`, `SMImage` et `SMWidget` (Figure 1). Elles héritent de SMShape et respectent le contrat suivant : l'attribut hérité *shape* est de type Shape (Java2D). Elles disposent de méthodes générales pour manipuler les attributs de style et de géométrie et de méthodes spécifiques, comme `setText(t)` pour spécifier le texte d'un objet SMText ou

`processEvent(evt)` pour envoyer un événement à un `SMWidget`. Le développeur peut aisément définir ses propres classes d'objets graphiques dès lors qu'il respecte le contrat ci-dessus.

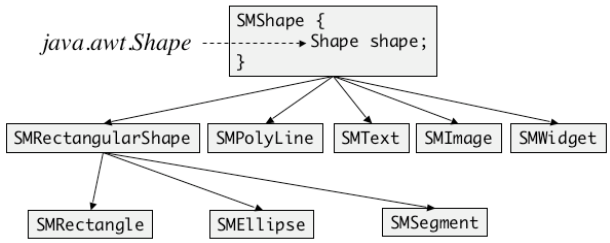


Figure 1 : diagramme de classes des objets graphiques.

Les objets peuvent être ajoutés ou retirés de la liste d'affichage à tout moment, et leur ordre peut être modifié. Chaque objet graphique a un booléen qui spécifie s'il est affiché ou non. Cela est utile lorsque l'on veut cacher un objet momentanément, par exemple un menu qui n'apparaît que lorsque l'on clique. Un autre boolén spécifie s'il reçoit ou non les événements d'entrée. Cela permet par exemple d'avoir des objets décoratifs.

Les coordonnées d'un objet sont, par défaut, relatives à la surface de dessin qui le contient (le SMCanvas), l'origine étant en haut à gauche. Ces coordonnées, et donc la géométrie de l'objet, peuvent être transformées par translation, rotation et mise à l'échelle, de manière relative ou absolue (par exemple, `translateBy(tx, ty)` et `translateTo(tx, ty)`). Par défaut, le point de référence de la rotation et du changement d'échelle est au centre de l'objet. Il peut être modifié en le spécifiant relativement à la boîte englobante de l'objet : (0, 0) correspond au point haut gauche et (1, 1) au point bas droite Par défaut il vaut donc (0.5, 0.5). De plus, un objet graphique peut spécifier un *parent*, qui est un autre objet graphique de la même liste d'affichage. Les coordonnées de l'objet sont alors interprétées relativement à ce parent, c'est-à-dire que les tranformations du parent et de l'objet sont cumulées, et ceci récursivement si le parent a lui-même un parent. Enfin, un objet graphique peut être *masqué* par un autre objet de la même liste d'affichage, c'est-à-dire que seule la partie de l'objet qui intersecte son masque est affichée. Comme dans GmlCanvas [4] et à l'inverse d'autres graphes de scène comme Inventor [27], l'ordre de la liste d'affichage est indépendant des relations entre objets, ce qui permet par exemple à un objet d'être derrière son parent.

Toutes les méthodes qui permettent de manipuler les objets graphiques de SMCanvas ont la particularité de renvoyer l'instance sur laquelle elles sont appelées. Ceci permet une *syntaxe enchaînée* des appels qui réduit la taille du code et le nombre de variables. Par exemple, l'instruction : `(new SMEllipse(10,10,20,30)).rotate(45).setFillPaint(red).addTo(c)` crée une ellipse, la tourne, modifie sa couleur de fond et l'ajoute au canvas `c`. Elle remplace une déclaration de variable et 4 affectations par une seule ligne. De plus, on peut utiliser une telle expression comme paramètre d'un appel de méthode.

Les "tags" et les groupes d'objets

Les *tags* (ou étiquettes) permettent de marquer les objets graphiques et de représenter ainsi aisément des *groupes d'objets*. Dans la boîte à outils Tk [21] on peut associer un ou plusieurs *tags* (des chaînes de caractères) à chaque objet du *canvas* afin de manipuler tous les items qui ont le même tag par ce seul identifiant. Dans SMCanvas, par application du principe de réification, les tags sont des objets à part entière, instances de la classe `SMTag` et, par application du principe de polymorphisme, la plupart des opérations applicables à un objet graphique le sont aussi à un tag. Les tags sont également des itérateurs (interface java.util.Iterator), ce qui permet de les énumérer aisément, comme illustré ci-dessous :

```
SMCanvas c = new SMCanvas(300, 300);
// étiqueter trois objets avec le tag "g1"
c.newEllipse(10, 10, 30, 20).addTag("g1");
c.newRectangle(50, 50, 5, 10).addTag("g1");
c.newRectangle(100, 50, 25, 20).addTag("g1");
// tous les objets étiquetés sont translatés de 10 pixels horizontalement
SMTag group1 = SMNamedTag.getTag("g1");
group1.translateBy(10, 0);
// seuls les objets assez larges sont colorés en rouge
for(group1.reset();group1.hasNext(); ) {
   SMShape s = group1.nextShape();
   if(s.getWidth() > 10)
      s.setFillPaint(Color.RED);
}
```

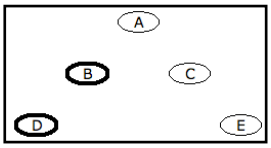
De même que `SMShape`, la classe `SMTag` peut être étendue en respectant le contrat suivant : les méthodes `reset()`, qui initialise l'itérateur, `hasNext()`, qui renvoie vrai s'il reste un objet, et `nextShape()`, qui renvoie le prochain objet, doivent être implémentées. Nous avons défini deux grandes catégories de tags : les tags *extensionnels*, qui sont apposés explicitement sur les objets graphiques, et les tags *intentionnels*, qui étiquettent tous les objets graphiques qui satisfont une propriété.

Un exemple de tag extensionnel est la classe `SMNamedTag` qui permet, à la manière de Tk, d'apposer une chaîne de caractères sur un objet, comme ci-dessus : les trois objets sont étiquettés "g1". Un exemple de tag intentionnel est la classe `SMHierarchyTag` qui étiquette le sous-arbre d'un objet, comme illustré ci-dessous[1] :

```
SMShape a = c.newEllipse(100, 100, 40, 20);
SMShape b = c.newEllipse(50, 150, 40, 20);
SMShape c = c.newEllipse(150, 150, 40, 20);
SMShape d = c.newEllipse(0, 200, 40, 20);
```

[1] On peut noter que Tk a deux tags intentionnels prédéfinis : *all* (tous les objets) et *current* (l'objet sous le curseur).

```
SMShape e = c.newEllipse(200,
200, 40, 20);
// créer la hiérarchie
a.addChild(b).addChild(c);
b.addChild(d);
c.addChild(e);
```




```
// colorer tout l'arbre en blanc
SMHierarchyTag tree = new SMHierarchyTag(a);
tree.setFillPaint(Color.WHITE);
// changer l'épaisseur des nœuds d'un sous-arbre
(new SMHierarchyTag(b)).setStroke(new BasicStroke(3));
```

L'ensemble des objets représentés par un tag intentionnel est calculé à chaque utilisation du tag. Ainsi, le tag hiérarchique `tree` ci-dessus désignera toujours le sous-arbre de `a`. Un autre exemple de tag intentionnel, que l'on aurait pu utiliser dans le premier exemple ci-dessus, est défini par un prédicat, ici `s.getWidth()>10`.

Par ailleurs, les tags sont *actifs* : l'ajout d'un tag à un objet `s` provoque l'appel de la méthode `added(s)` du tag et la suppression de ce tag provoque l'appel de sa méthode `removed(s)` (pour l'instant, cela ne concerne que les tags extensionnels). L'exemple ci-dessous montre comment changer l'épaisseur du contour des formes lorsqu'elles ont le tag `SelectedTag` :

```
class SelectedTag extends SMExtensionalTag {
   Stroke selectedStroke = new BasicStroke(3);
   Stroke normalStroke = new BasicStroke(1);
   ...
   public void added(SMShape s)
      { s.setStroke(selectedStroke); }
   public void removed(SMShape s)
      { s.setStoke(normalStroke); }
}
```

Enfin, les tags sont également utilisés pour associer une même interaction à plusieurs objets, comme nous allons le voir dans la prochaine section.

L'INTERACTION DANS SMCANVAS

La spécification de l'interaction dans SMCanvas se fait à l'aide de *machines à états*. Une ou plusieurs machines à états peuvent être attachées à un `SMCanvas`, et elles peuvent être activées ou désactivées indépendamment.

Machines à états

Une machine est constituée d'un ensemble d'*états*, dont un *état initial* qui est l'état de départ. Le passage d'un état à un autre est décrit par des *transitions*, qui sont déclenchées par l'arrivée d'événements tels que le déplacement de la souris, l'appui sur un bouton, l'écoulement d'un délai, etc. A chaque transition peuvent être associées :

- une *garde*, c'est-à-dire une condition booléenne qui retourne Vrai lorsque la transition peut être déclenchée. Une garde non spécifiée retourne toujours Vrai.
- une *action*, qui est du code exécuté lorsque la transition est déclenchée.

A chaque état peuvent être associées :

- une *action d'entrée*, qui est du code exécuté lorsque l'état devient l'état courant ;
- une *action de sortie*, qui est du code exécuté lorsque l'état cesse d'être l'état courant.

Au départ, l'état courant est l'état initial. Lorsqu'un événement arrive, s'il existe une transition correspondant à cet événement dans l'état courant et dont la garde retourne Vrai, cette transition est déclenchée :

- l'action de sortie de l'état courant est exécutée ;
- l'action de la transition est exécutée ;
- l'état courant devient l'état d'arrivée de la transition et son action d'entrée est exécutée.

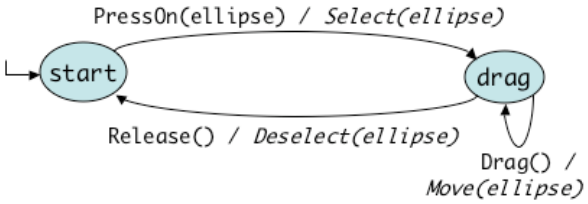


Figure 2 : Machine à états décrivant le déplacement d'un objet.

La figure 2 montre une représentation graphique d'une machine à états simple. Les états sont des ellipses et les transitions sont des flèches orientées de leur état de départ vers leur état d'arrivée. Les transitions sont étiquetées par des expressions de la forme événement / *action*. Les états contiennent leur nom et l'état initial est repéré par une flèche en forme de L.

SMCanvas utilise les classes internes (*inner class*) de Java pour décrire les machines à états avec une syntaxe familière aux développeurs. En effet, les classes internes sont très souvent utilisées pour écrire des écouteurs d'événements (interface java.util.EventListener). Par exemple, on spécifie que l'application doit quitter lorsque l'on clique sur le bouton "quit" de la manière suivante :

```
1 new JButton("quit").addActionListener(
2    new ActionListener(){
3       public void actionPerformed(ActionEvent e)
4          { System.exit(1) ; }
5    }) ;
```

Les lignes 2 à 4 spécifient une sous-classe de la classe ActionListener dont une unique instance est créée et passée en paramètre à addActionListener. De plus, le nouvel objet est créé "à l'intérieur" de l'objet qui exécute ce code et il a accès à cet objet englobant.

Les machines à états de SMCanvas exploitent cette possibilité de spécifier une sous-classe partout où l'on peut créer un objet et de créer des objets ayant directement accès à leur contexte. Ainsi, `State` est une classe interne de la machine à états `StateMachine`. `State` possède les méthodes `enter()` et `leave()` qui peuvent être surchargées pour spécifier les actions d'entrée et de sortie d'un

état. `Transition` est une classe interne de `State`. L'action de la transition et sa garde sont spécifiées en surchargeant les méthodes `action()` et `guard()`. Ces différentes classes réifient le formalisme des machines à états en objets Java et constituent une grande originalité de SMCanvas. A notre connaissance, les autres implémentations de machines à états en Java n'utilisent pas cette approche, mais traduisent une description graphique ou textuelle de la machine à états en Java.

SMCanvas définit un ensemble de classes de `Transition` pour les événements clavier (`KeyPress`, `KeyRelease` et `KeyType`) et souris (`Press`, `Release`, `Click`, `Move` et `Drag`) (voir Figure 3). Les transitions des événements souris peuvent être suffixées par `OnShape` ou `OnTag` afin de spécifier le contexte nécessaire au déclenchement de ces transitions : `ClickOnShape` définit un clic sur un objet donné et `ClickOnTag` un clic sur un objet ayant un tag donné. Ces conditions pourraient être spécifiées par des gardes mais elles sont très courantes et l'on gagne ainsi en concision.

La machine de la figure 2 s'écrit alors ainsi :

```
sm = new StateMachine("SimpleDAndD") {
   SMShape dragged = null;
   Point2D pInit = null;
   Point2D pSize = new Point2D (30, 40);

   public Point2D offset (Point2D p) {
      return new Point2D (p.getX() - pInit.getX(),
                          p.getY() - pInit.getY());
   }
   // L'état de départ avec une transition vers l'état drag
   public State start = new State() {
      Transition dragOn = new PressOnShape
                              (BUTTON1, "drag") {
         public void action() {
            dragged = getShape();
            pInit = getPoint();
         }
      };
   };

   // L'état drag avec deux transitions
   public State drag = new State() {
      // pas d'état destination : boucle sur drag
      Transition drag = new Drag(BUTTON1) {
         public void action() {
            dragged.translateBy(offset(getPoint()));
            pInit = getPoint();
         }
      };
      // retour à l'état de départ
      Transition dragOff = new Release
                              (BUTTON1, "start") {
         public void action() {
            dragged.translateBy(offset(getPoint()));
            dragged = null;
         }
      };
   };
};
```

L'intérêt de cette syntaxe pour décrire l'interaction est triple : elle permet de localiser toute l'interaction au même endroit, au lieu de la répartir dans différents objets ("handlers" ou "listeners") ; elle ne déroute pas les développeurs qui ont l'habitude des classes internes ; elle facilite la mise au point car on a directement accès au code qui est compilé et exécuté, contrairement aux approches qui utilisent une description graphique ou textuelle de la machine à états.

La seule concession requise par cette approche est de dénommer l'état d'arrivée d'une transition par son nom (les chaînes "drag" et "start") plutôt que par l'objet lui-même. En effet, les champs `drag` et `start` ne sont pas initialisés au moment où la transition est construite. Nous utilisons l'interface de réflection de Java pour transformer les chaînes de caractères en références aux objets de même nom à l'initialisation de la machine.

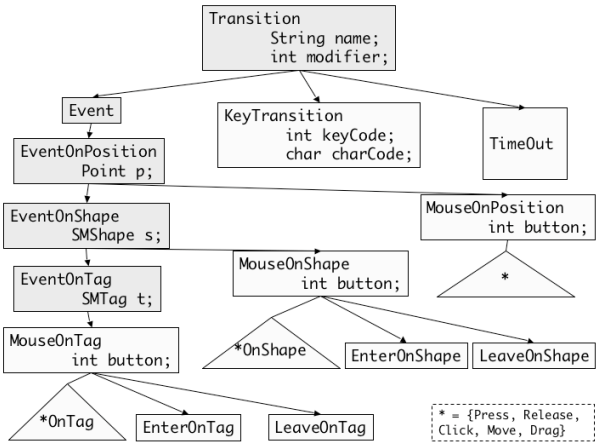


Figure 3 : Classes de transitions de SMCanvas. Les sous-classes de KeyTransition sont omises pour plus de lisibilité. Les triangles dénotent chacun cinq classes correspondant aux événements Press, Release, Click, Move et Drag de la souris.

De nouveaux événements

Se restreindre aux événements standard du clavier et de la souris est insuffisant pour développer des interfaces avancées. L'utilisation de la multimodalité requiert, par exemple, le traitement d'événements gestuels ou vocaux. Dans cette section, nous présentons les possibilités d'extension de SMCanvas avec l'ajout de la reconnaissance de gestes.

Les transitions d'une machine à états sont stockées dans une table associative dont la clé est une chaîne de caractères contenant le type de l'événement attendu et ses paramètres (bouton, shift, etc.). Pour les événements standard, SMCanvas les écoute grâce aux `Listener` Java et envoie la clé correspondante à la machine à états (par exemple, pour un appui du bouton gauche de la souris, la clé est "Press1"). Celle-ci cherche alors si son état courant dispose d'une transition correspondant à cette clé pour la déclencher. Afin d'enrichir le vocabulaire

d'événements, la classe `SMVirtualEvent` permet de définir de nouvelles clés (`new SMVirtualEvent("newKey")`. Ces événements virtuels peuvent être envoyés à une machine à états qui peut alors déclencher des transitions génériques de la forme `Event–On*("newKey"), * ∈ {Position, Shape, Tag}`.

A titre d'illustration, nous présentons une machine à états dont les transitions sont déclenchées par des gestes tracés à la souris. L'exemple ci-dessous montre son utilisation avec trois gestes, couper, copier et coller :

```
smGesture = new StateMachine("CutCopyPaste", c) {
  public State start = new State() {
    Transition copy = new EventOnShape("copy"){...};
    Transition cut = new EventOnShape ("cut"){...};
    Transition paste =
                new EventOnPosition("paste"){...};
  };
};
```

Nous définissons une deuxième machine à états, `smInk`, qui utilise un objet `Classifier` implémentant l'algorithme de reconnaissance de geste de Rubine [25]. La machine `smInk` dessine l'encre du geste et ajoute les points un par un à un objet `Gesture`. Lorsque l'utilisateur relâche le bouton de la souris, elle classifie le geste et envoie un événement virtuel du nom de la classe reconnue à `smGesture`.

```
smInk = new StateMachine("Ink", c) {
   // l'encre du geste
   SMPolyLine ink = (SMPolyLine) canvas.newPolyLine(0,
0).setPickable(false).setFilled(false);
   // le geste
   Gesture gesture = new Gesture();
   // le classifieur pour reconnaître les gestes
   Classifier classifier = Classifier.
                    newClassifier("cutcopypaste.cl");
   public State start = new State() {
      Transition begin = new Press(BUTTON1){
         public void action(){
           ink.reset(getPoint()).
                  setDrawable(true).beforeAll();
           gesture.reset();
           gesture.addPoint(getPoint());
         }
      };
      Transition draw = new Drag(BUTTON1){
         public void action(){
            ink.lineTo(getPoint());
            gesture.addPoint(getPoint());
         }
      };
      Transition end = new Release(BUTTON1){
         public void action(){
            gesture.addPoint(getPoint());
               GestureClass gc =
                        classifier.classify(gesture);
               // envoyer un événement virtuel ayant pour
               // clé le nom du geste reconnu à smGesture
                smGesture.processEvent(
                  new SMVirtualEvent(gc.getName()),
                        getModifier(), getPoint());
               ink.setDrawable(false);
         }
      };
   };
};
```

Cet exemple illustre le dialogue que le développeur peut établir entre différentes machines à états pour décrire des interactions complexes. Nous avons également prototypé une palette semi-transparente [5] pour modifier la couleur des objets graphiques d'une scène (Figure 4). Tout composant Swing, y compris un SMCanvas, peut être ajouté à un SMCanvas grâce à la classe `SMWidget` et manipulé comme tout objet graphique (translation, rotation, etc.). Notre palette est un SMCanvas dont la machine à états envoie des événements à la machine du SMCanvas de la scène principale lorsque l'on clique dessus, implémentant ainsi le "clic à travers". Le déplacement et la rotation de la palette sont commandés par des événements reçus par la machine à états principale. Etablir un dialogue entre machines à états nous parait être une approche particulièrement prometteuse pour traiter des interactions complexes de façon simple. Elle se rapproche des machines hiérarchiques [6] et mérite d'être explorée de façon plus approfondie.

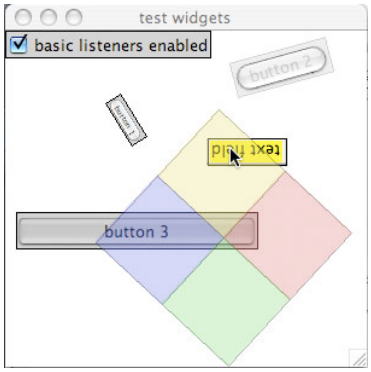


Figure 4 : SMCanvas contenant des widgets Swing, ainsi qu'un autre SMCanvas implémentant une toolglass.

EVALUATION

L'évaluation d'une boîte à outils d'interface est un problème bien connu mais mal résolu. Les concepteurs fournissent généralement des informations sur la taille de la toolkit. Pour SMCanvas, le code source représente 4910 lignes (hors commentaires) et la bibliothèque exécutable 84 Ko. Ceci ne constitue bien évidemment pas une mesure de son efficacité et encore moins de son utilisabilité. Un autre critère fréquemment invoqué est l'utilisation de la boîte à outils par d'autres que leurs concepteurs, mais il se heurte au problème de la poule et de l'œuf : il faut attendre que la boîte à outils soit utilisée pour pouvoir publier, mais il faut publier pour qu'elle soit connue et donc utilisée. Par ailleurs, mesurer le seul nombre de téléchargements est peu scientifique et devrait

être complété par une analyse des usages réels, ce qui est une tâche considérable.

Revenant à notre objectif de conception initial, nous avons décidé d'évaluer si notre boîte à outils répondait au critère d'A. Kay (*simple things should be simple, complex things should be possible*). Nous avons d'abord développé quelques exemples simples, dont certains ont été présentés plus haut, puis nous avons expérimenté SMCanvas avec deux classes d'étudiants de Master. Nous rapportons ci-dessous le résultat de l'une de ces expériences, et proposons une méthode plus générale d'évaluation fondée sur les benchmarks.

Evaluation dans l'enseignement

Notre expérience d'enseignement de l'IHM est qu'il est difficile de faire expérimenter par les étudiants les nouvelles techniques d'interaction car l'effort de programmation avec les boîtes à outils existantes est trop élevé. Nous utilisons depuis longtemps le langage Tcl et sa boîte à outils Tk [21], notamment le widget *canvas* dont nous nous sommes inspirés pour le modèle graphique de SMCanvas. Les étudiants sont cependant rebutés par la nécessité d'apprendre un nouveau langage dont ils auront peu l'usage par ailleurs. Surtout, ils ne voient pas comment exploiter les connaissances acquises dans ce cadre avec les outils qu'ils auront à utiliser dans leur vie professionnelle, et en particulier Java.

Nous avons utilisé SMCanvas dans le module "Fondements de l'Interaction Homme-Machine" du Master de recherche en informatique de l'Université Paris-Sud entre octobre et décembre 2005. Les étudiants devaient implémenter une technique d'interaction décrite dans un article de recherche et appliquée à l'édition de formes graphiques simples. 14 binômes ont traité 8 sujets (Table 1). Une présentation du widget SMCanvas fut faite en cours (1/2 h), et la documentation et un site Wiki donnant quelques exemples de code mis à la disposition des étudiants. Tous les groupes sauf un ont rendu un prototype fonctionnel illustrant la technique d'interaction choisie et un seul groupe a fait appel à nous pour mener à bien son projet. La Figure 5 montre trois exemples de projets : *alignment stick*, *local tools* et *magnetic guidelines*.

Marking Menu [13]	2 groupes
Flow Menu [10]	1 groupe
Entrée gestuelle [14]	2 groupes
Side Views [28]	2 groupes
Local tools [3]	2 groupes
Alignment stick [24]	2 groupes
Pushpins, rulers and pens [26]	2 groupes
Magnetic guidelines [2]	1 groupe

Table 1 : Techniques d'interaction de l'expérimentation.

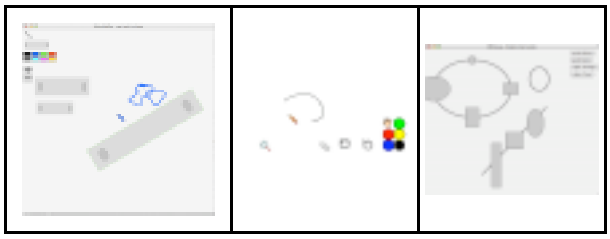


Figure 5 : Exemples de projets réalisés par les étudiants.

L'examen du code source montre que les étudiants ont, dans leur majorité, exploité sans difficulté les capacités de SMCanvas. Tous les groupes ont utilisé une seule machine à états, sauf un qui a utilisé une machine pour les règles et une pour les stylos pour le projet *Pushpins, rulers and pens*. Les machines comportent environ 300 lignes de code (lignes vides et commentaires inclus), à l'exception d'un groupe dont la machine fait 794 lignes. Les machines ont de 2 à 9 états et de 8 à 32 transitions.

Tous les groupes ont utilisé la syntaxe enchaînée de SMCanvas. Seul un groupe (*SideViews*) a étendu `SMShape` pour gérer un historique des transformations. Tous les groupes ont utilisé les tags à la fois pour grouper les objets (outils d'une palette par exemple) et pour l'interaction (pour distinguer les outils des objets édités par exemple). Seuls les tags extensionnels étaient disponibles lors de l'évaluation. Un groupe a noté la difficulté de manipuler une forme et ses descendants, et aurait bénéficié, comme d'autres, des tags intentionnels. Certains groupes n'ont pas bien compris l'usage des transitions `*OnTag` et ont utilisé à la place des transitions `*OnShape` avec une garde qui vérifie si la forme est taggée.

Cette évaluation montre que SMCanvas permet de mettre en œuvre des techniques d'interaction avancées et donc qu'au moins certaines choses "complexes" sont possibles. Elle montre aussi que des étudiants connaissant le langage Java ont pu s'approprier la boîte à outils sans difficulté. Une autre expérience a démarré avec des étudiants de Master 1 qui suivent un cours d'introduction à l'IHM. Nous renouvellerons ces expériences l'an prochain. SMCanvas est à disposition de la communauté à l'adresse `http://wiki.lri.fr/SMCanvas`.

Vers des benchmarks de boîtes à outils

L'expérimentation menée ci-dessus nous amène à proposer le développement de *benchmarks* pour évaluer les boîtes à outils. Cette technique est commune en informatique et dans d'autres domaines de l'IHM, comme la visualisation d'information [23]. Bien sûr, elle n'est pas exclusive d'autres approches d'évaluation, notamment les études d'utilisabilité auprès des développeurs. Elle présente l'avantage de pouvoir être mise en œuvre de façon relativement simple pour les boîtes à outils d'interface.

Le principe est de définir un jeu de tests constitué d'interfaces et de techniques d'interaction couvrant le spectre

des interfaces graphiques, des plus simples aux plus complexes. Les techniques de la table 1 nous semblent un bon sous-ensemble d'un tel benchmark, qui devra bien sûr évoluer pour prendre en compte les avancées du domaine. Chaque boîte à outils se mesure à un tel benchmark en indiquant si elle est ou non capable d'implémenter l'interface ou la technique, et, si oui, en donnant des mesures objectives (nombre de lignes de codes, taille à l'exécution, temps de programmation, etc.) Les résultats seraient collectés dans un site Web et mis à la disposition de la communauté.

CONCLUSION ET PERSPECTIVES

Nous avons présenté SMCanvas, une extension de la boîte à outils Java Swing fondée sur un modèle graphique à base de structure d'affichage et un modèle d'interaction à base de machine à états. SMCanvas met en œuvre les principes de polymorphisme et de réification et exploite les caractéristiques du langage Java pour développer des interactions post-WIMP de façon simple et concise. SMCanvas a été évaluée avec des étudiants de Master sur un ensemble de techniques d'interaction avancées, confirmant sa facilité de prise en main et sa puissance d'expression. Nous proposons d'étendre cette approche d'évaluation en développant des benchmarks pour l'évaluation des boîtes à outils.

Les pistes que nous explorons pour les futures versions de SMCanvas sont les suivantes : placement des objets par contraintes, animation, utilisation de machines à états hiérarchiques [6], interfaçage avec ICON [7] pour traiter des entrées non conventionnelles. Nous voulons également augmenter son pouvoir d'expression pour implémenter des techniques qui ne sont pas confinées à l'intérieur d'un widget, comme le *drag-and-drop* [11]. Enfin, nous envisageons d'appliquer une approche similaire dans d'autres contextes, comme DHTML/Javascript pour le développement d'applications Web.

BIBLIOGRAPHIE

1. Beaudouin-Lafon, M. and Mackay, W. E. Reification, polymorphism and reuse: three principles for designing visual interfaces. In *Proc. Conference on Advanced Visual interfaces*. AVI '00. ACM Press, 2000, pp 102-109.
2. M. Beaudouin-Lafon. Designing interaction, not interfaces. In *Proc. Conference on Advanced Visual Interfaces*. AVI '04. ACM Press, 2004, pp 15–22.
3. Bederson, B. B. et al. Local tools: an alternative to tool palettes. In *Proc. ACM Symposium on User interface Software and Technology*. UIST '96. ACM Press, 1996, pp 169-170.
4. Bérard, F. GmlCanvas. http://iihm.imag.fr/projects/gml
5. Bier, E. A., Stone, M. C., Pier, K., Buxton, W., and DeRose, T. D. Toolglass and magic lenses: the see-through interface. In *Proc. ACM* SIGGRAPH '93. ACM Press, 1993, 73-80.
6. Blanch, R. Programmer I'interaction avec des machines à états hiérarchiques. In *Proc. French-Speaking Conference on Human-Computer interaction*. IHM '02. ACM Press, 2002, pp 129-136.
7. Dragicevic, P. and Fekete, J. Support for input adaptability in the ICON toolkit. In *Proc. international Conference on Multimodal Interfaces*. ICMI '04. ACM Press, 2004, pp 212-219.
8. Eng, E. Qt GUI Toolkit: Porting graphics to multiple platforms using a GUI toolkit. *Linux J.*, issue 31es, article 2, 1996.
9. Green, M.. A survey of three dialogue models. *ACM Trans. Graph.*, 5(3):244–275, 1986.
10. Guimbretière, F. and Winograd, T. FlowMenu: combining command, text, and data entry. In *Proc. ACM Symposium on User interface Software and Technology*. UIST '00. ACM Press, 2000, pp 213-216.
11. Hascoët, M., Collomb, M. et Blanch, R. Évolution du drag-and-drop : du modèle d'interaction classique aux surfaces multi-supports. In *Revue I3 (Information, Interaction, Intelligence)*, 4(2), pp 9-38, 2004.
12. Hudson, S. E., Mankoff, J., and Smith, I. Extensible input handling in the subArctic toolkit. In *Proc. ACM Conference on Human Factors in Computing Systems*. CHI '05. ACM Press, 2005, pp 381-390.
13. Kurtenbach, G. and Buxton, W. The limits of expert performance using hierarchic marking menus. In *Proc. ACM Conference on Human Factors in Computing Systems*. CHI '93. ACM Press, 1993, pp 482-487.
14. Landay, J. A. SILK: sketching interfaces like krazy. In *Conference Companion, Human Factors in Computing Systems*. CHI '96. ACM Press, 1996, pp 398-399.
15. Linton, M.A., Vlissides, J.M., and Calder, P.R. Composing User Interfaces with InterViews. *lEEE Computer*. Febuary, 1989, pp 8-22.
16. Lecolinet, E. A Brick Construction Game Model for Creating Graphical User Interfaces: The Ubit Toolkit. *Proc. IFIP Conference on Human-Computer Interaction*. INTERACT'99. IOS Press, 1999, pp 510-518.
17. Logan, S. *Gtk+ Programming in C.* Prentice Hall, 2001.
18. McCormack, J. and Asente, P. An overview of the X toolkit. In *Proc. ACM Symposium on User Interface Software and Technology*. UIST '88. ACM Press, 1988, pp 46-55.

19. Myers B.A. et al. The Amulet environment: new models for effective user interface software development. *IEEE Trans. Soft. Eng.*, 1997, 23(6):347-365.

20. Myers, B. A. et al. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *IEEE Computer*, 1990, 23(11):71-85.

21. Ousterhout, J. K. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

22. Pausch, R., Conway, M., and Deline, R. Lessons learned from SUIT, the simple user interface toolkit. *ACM Trans. Inf. Syst.* 10(4):320-344, 1992.

23. Plaisant, C. The challenge of information visualization evaluation. In *Proc. Conference on Advanced Visual interfaces*. AVI '04. ACM Press, 2004, pp 109-116.

24. Raisamo, R. and Räihä, K. A new direct manipulation technique for aligning objects in drawing programs. In *Proc. ACM Symposium on User interface Software and Technology*. UIST '96. ACM Press, 1996, pp 157-164.

25. Rubine, D. Specifying gestures by example. In *Proc. ACM Conference on Computer Graphics and Interactive Techniques* SIGGRAPH '91. ACM Press, 1991, pp 329-337.

26. St. Amant, R. and Horton, T. E. Characterizing tool use in an interactive drawing environment. *Proc. international Symposium on Smart Graphics*. SMARTGRAPH '02, vol. 24. ACM Press, 2002, pp 86-93.

27. Strass, P. IRIS Inventor, a 3D Graphics Toolkit. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA '93, ACM Press, 1993, pp 192-200.

28. Terry, M. and Mynatt, E. D. Side views: persistent, on-demand previews for open-ended tasks. In *Proc. ACM Symposium on User interface Software and Technology*. UIST '02. ACM Press, 2002, pp 71-80.