Morphic, les interfaces utilisateurs selon Squeak

H. Fernandes - hilaire@ofset.org
S. Stinckwich - Serge.Stinckwich@info.unicaen.fr

Après notre excursion dans les arcanes de Smalltalk lors de notre article précédent, nous allons cette fois-ci nous rafraîchir un peu avec la découverte du framework Morphic. Morph est l'unité de base de l'interface graphique de Squeak. Il est écrit en Smalltalk et donc entièrement portable d'un système d'exploitation à un autre. La grande particularité de celui-ci est de permettre à l'utilisateur une déconsrtuction/reconstruction à la volée des différents éléments composant une interface graphique morphique.

L'histoire de Morphic

Morphic fût développé dans le cadre du langage *Self*, dans les années 80. Ce dernier s'inspire largement de *Smalltalk*. Morphic fût porté par la suite dans *Squeak*. L'idée originelle du framework Morphic est de permettre à chaque objet d'être directement représentable et manipulable sous la forme d'une forme graphique, le morph. L'utilisateur peut ensuite directement agir à la souris sur les différents morph composants une interface graphique. Par exemple, si vous afficher le menu du monde avec ***Echap***, vous pouvez ensuite détacher un des items du menu pour en faire un bouton : cliquez deux fois de suite avec le bouton droit au dessus de l'item à détacher, puis prenez l'item avec le halo noir (en haut) pour le déposer ailleurs.

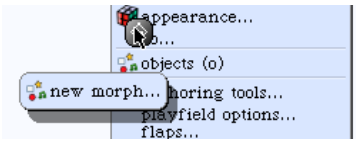

Fig1. - Détacher un morph, ici l'item de menu morphique 'new morph', pour en faire un élément indépendant.

Autre exemple, pour voir la transformation d'un objet en morph, exécutez dans un espace de travail le code suivant :

```
'Morph' asMorph openInWorld.
```

Vous pouvez aussi essayer :

```
 'Morph' asMorph openViewerForArgument
```

Vous obtenez un panneau permettant de contrôler l'élément graphique graphiquement. Bien sûr il est possible de créer un morph qui soit une meilleure représentation graphique. La méthode **asMorph** est implantée dans la classe **Object** et se contente, par défaut, de créer un morph de type chaîne de caractères. Ainsi le code **Color tan asMorph** retourne un morph de type chaîne de caractères avec comme valeur 'Color tan', il serait plus intéressant que soit retourné un morph de type rectangle coloré. Rien de plus facile : ouvrez la classe **Color** (par exemple dans un *workspace* tapez **Color**, sélectionnez le texte et invoquez le navigateur de classe avec **[Alt]+b**). Créez dans la classe **Color** une nouvelle méthode **asMorph** comme suit :

```
Color>>asMorph
   ^ Morph new color: self
```

Maintenant dans un *workspace*, exécutez le code **Color orange asMorph openInWorld**. Vous obtenez comme représentation graphique de l'objet couleur orange, un joli rectangle orange !

Bien sûr les morph sont eux-mêmes des objets, donc nous pouvons les manipuler comme tous les autres objets en Smalltalk : modification de leurs attributs (essentiellement graphiques), création de nouvelles classes

de morph et exploitation de l'héritage, etc. Dans la suite nous allons nous intéresser à certains de ces aspects.

Manipuler des morphs

En tant qu'objet graphique, un morph a une position – son coin supérieur gauche – et une taille – une boîte englobante. La méthode **`position`** retourne un point – par exemple **`10@10`** – spécifiant les coordonnées du coin haut gauche du moprh, l'origine étant le coin haut gauche de l'écran. La méthode **`extent`** retourne également un point, taille du morph ; la première coordonnée est sa largeur, la deuxième sa hauteur.

Dans un *worskpace* commençons par créer deux morph pour jouer avec. Écrivez et exécutez avec **[Alt]+e** le code suivant :

```
joe := Morph new color: Color blue.
joe openInWorld.
bill := Morph new color: Color red .
bill openInWorld.
```

Puis faites un *Print-it* avec **[Alt]+p** de : **`joe position.`**

Pour déplacer joe, exécutez plusieurs fois le code suivant en maintenant la combinaison **[Alt]+e** appuyée :

```
joe position: (joe position + (10@3)).
```

On a même un effet de déplacement !

On peut faire le même type de manipulation avec la taille : un *Print-it* de **`joe extent`** nous donne sa taille. Pour faire grossir joe, exécutez plusieurs fois : **`joe extent: (joe extent * 1.1)`**

Pour changer la couleur de nos morphs, leur envoyer le message **`color:`** avec comme argument une couleur. Par exemple **`joe color: Color orange`** ou en transparence **`joe color: (Color orange alpha: 0.5).`**

Pour programmer bill afin qu'il suive joe :

```
bill position: (joe position + (100@0)).
```

Exécutez ce code à chaque fois que joe est déplacé à la souris, bill se positionnera à 100 pixel à droite de joe.

Emboîter des morphs

Un morph peut être ajouté dans un autre morph avec le message **`addMorph:`** envoyé au conteneur – le morph qui va contenir un autre morph.

Exécutez le code suivant :

```
star := StarMorph new color: Color yellow.
joe addMorph: star.
star position: joe position.
```

La dernière ligne positionne l'étoile aux mêmes coordonnées que joe. Vous aurez sans doute remarqué que les coordonnées d'un morph contenu dans un autre restent absolues, elles ne sont pas relatives au morph le contenant (ceci est une limite de Morphic). Pour le positionnement d'un morph, il existe beaucoup de méthodes disponibles, il faut parcourir la classe **`Morph`** pour s'en rendre compte. Pour positionner l'étoile au milieu de joe, exécutez le code : **`star center: joe center`**

Fig. 2 – L'étoile est contenue dans joe, le morph bleu transparent

Maintenant, si vous attrapez à la souris l'étoile vous attrapez en fait joe, et les deux morph sont déplacés en même temps, l'étoile est en quelque sorte ancrée dans joe. Il est bien sûr possible d'ajouter d'autres morphs dans joe.

Pour enlever un sous-morph : **`joe removeMorph: star`**, ou bien encore **`star delete`**

Créer et dessiner nos propres morph

La composition de morph permet d'obtenir des représentations intéressantes, mais parfois vous avez vraiment besoin de dessiner un morph qui soit totalement différent. Pour ce faire il suffit de définir une classe héritière de **`Morph`** – ou toute autre classe fille – et d'y surcharger (*override*) la méthode **`drawOn:`**

Cette méthode a un paramètre qui est une instance de **`Canvas`** pour dessiner dessus. Essayons de dessiner un morph représentant une croix. Depuis le navigateur de classe définissez une nouvelle classe **`CrossMorph`** héritière de la classe **`Morph`** :

```
Morph subclass: #CrossMorph
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'LesMorph'
```

Nous définissons ensuite la méthode **`drawOn:`** comme suit :

```
drawOn: aCanvas
   | aBounds crossHeight crossWidth |
   crossHeight := (self height / 3) asFloat.
   crossWidth := (self width / 3) asFloat.
   aBounds := self bounds top: self top + crossHeight.
   aBounds := aBounds bottom: self bottom - crossHeight.
   aCanvas fillRectangle: aBounds color: self color.
   aBounds := self bounds left: self left + crossWidth.
   aBounds := aBounds right: self right - crossWidth.
   aCanvas fillRectangle: aBounds color: self color
```

La méthode **`bound`** retourne la boîte englobante d'un morph, c'est une instance de **`Rectangle`** définie par deux points : l'origine en haut à gauche et le coin en bas à gauche du rectangle.

Pour tester notre nouveau morph exécutez **`CrossMorph new openInWorld`**.

Fig. 3 – Notre morph en forme de croix. Il peut être dimensionné à loisir.

Notez cependant que la zone sensible – où il faut cliquer à la souris pour attraper le morph – est toujours un rectangle englobant. Nous voudrions que cette zone soit réduite à la seule surface de la croix. Pour ce faire il suffit de redéfinir la méthode **`containsPoint:`**

```
containsPoint: aPoint
   | aBounds1 aBounds2 crossHeight crossWidth |
   crossHeight := (self height / 3) asFloat.
   crossWidth := (self width / 3) asFloat.
   aBounds1 := self bounds top: self top + crossHeight.
   aBounds1 := aBounds1 bottom: self bottom - crossHeight.
   aBounds2 := self bounds left: self left + crossWidth.
   aBounds2 := aBounds2 right: self right - crossWidth.
   ^ (aBounds1 containsPoint: aPoint)
      | (aBounds2 containsPoint: aPoint)
```

Nous nous appuyons en fait largement sur la méthode **`containsPoint:`** de la classe **`Rectangle`**. Entre les méthodes **`drawOn:`** et **`containtsPont:`** il y a clairement duplication de code, cela indique un besoin de factorisation de code dans une tiers méthode, nous vous laissons cela comme d'exercice.

Ainsi pour créer de nouveaux morph, il y a donc principalement deux méthodes :

1. Par composition et emboîtement de morph les uns dans les autres, c'est ce que nous avons vu au début de cet article ;
2. Par utilisation de la méthode **`drawOn:`** pour dessiner de nouvelles formes de morph.

Il est possible de combiner ces méthodes.

Ci dessous nous vous présentons quelques morphs conçus pour un projet éducatif. Vous trouverez les sources de ces morph en référence en fin d'article. Vous devez télécharger les fichiers **`.st`** et les installer dans votre

image de Squeak : ouvrir le menu du monde, choisir ***open...*** puis ***file list*** puis faire un ***filein*** du fichier source correspondant. Ces morph sont rangés dans la catégorie de classe **`LesMorph`**, vous êtes fortement invités à naviguer dans leur code source et aussi à tester des modifications sur ceux-ci.

Une étiquette auto-collante

Le **`LabelstickerMorph`** est la métaphore d'une étiquette auto-collante comprenant un liseré de couleur unie et trois lignes de texte.

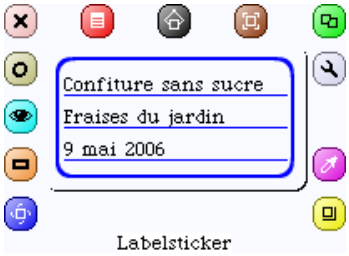

Fig. 4 – Notre morph étiquette auto-collante

Pour manipuler l'étiquette nous disposons des méthodes suivantes :

```
label := LabelstickerMorph new openInWorld.
label text1: 'Confiture sans sucre';
   text2: 'Fraises du jardin';
   text3: '9 mai 2006'.
label lineColor: Color red
```

Une pyramide de nombres

Le morph précédant est conçu en redéfinissant la méthode **`drawOn:`**. Le morph **`PyramidMorph`** est lui construit par composition de morph, ici des **`TextMorph,`** constituant les différentes briques de la pyramide, sont ajoutés dans le morph de base.

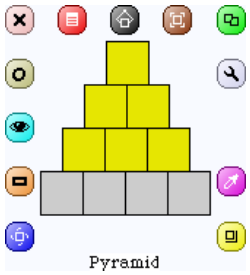

Fig. 5 – Un morph pyramide de nombres

Pour la manipuler nous disposons des méthodes :

```
pyramid := (PyramidMorph base: 4) openInWorld
pyramid block: 8 value: 2
```

Interactions souris et clavier

Événements souris

Outre l'aspect graphique des morphs, nous avons également besoin d'interagir avec ceux-ci par l'intermédiaire de la souris ou du clavier. Afin de gérer les événements souris, un morph doit répondre **`true`** au message **`handlesMouseDown:`** En effet, lorsqu'un bouton souris est pressé, le système morphic interroge chaque morph en-dessous de la souris en utilisant le message **`handlesMouseDown:`**, lorsqu'un morph répond **`true`** à ce message, alors morphic lui envoie immédiatement le message **`mouseDown:`**, et le message **`mouseUp:`** lorsque le bouton souris est lâché. Si tous les morph répondent **`false`**, alors morphic démarre une opération de saisissement, ce qui est le comportement par défaut pour attraper les morph.

Les méthodes **`mouseDown:`** et **`mouseUp:`** ont comme argument un instance de **`MorphicEvent`**. Parcourez la classe **`MorphicEvent`** pour en savoir plus sur son protocole.

Nous allons étendre notre morph **`CrossMorph`** afin qu'il puisse gérer les événements souris, d'abord celui-ci doit accepter les événements souris, ajoutez dans sa classe la méthode :

```
CrossMorph>>handlesMouseDown: evt
       ^true
```

Lorsque le bouton gauche – bouton rouge en Smalltalk – est cliqué, nous proposons de changer la couleur de la croix en rouge. Lorsque le bouton du milieu – bouton jaune en Smalltlak – nous changeons la couleur en bleu :

```
CrossMorph>>mouseDown: evt
   evt redButtonPressed
      ifTrue: [self color: Color red].
   evt yellowButtonPressed
      ifTrue: [self color: Color blue].
   self changed
```

Une fois que les événements souris sont gérés par le morph, il n'est plus possible d'attraper et déplacer le morph directement à la souris. Pour ce faire il faut utiliser les halos : clic droit sur le morph pour faire apparaître les halos et prendre la poignée noir ou marron en haut du morph.

L'argument **evt** de **mouseDown:** est une instance de **MouseEvent**, sous-classe de **MorphicEvent**. C'est dans cette première que les méthodes **redButtonPressed** et **yellowButtonPressed** sont définies. Parcourez cette classe pour en savoir plus sur les méthodes disponibles pour gérer les événements souris.

Événements clavier

Attraper les événements clavier se fait en plusieurs temps :

1. Donner le focus clavier à notre morph, pour notre exemple nous pouvons considérer que nous donnons le focus clavier à notre morph lorsque la souris est au dessus de celui-ci.
2. Gérer l'événement clavier lui-même, cela se fait avec la méthode **handleKeystroke:**. Ce message est envoyé à notre morph lorsqu'une touche du clavier est pressée **et** que notre morph à le focus clavier.
3. Libérer le focus clavier lorsque la souris n'est plus au-dessus de notre morph.

Nous allons étendre notre morph **CrossMorph** pour qu'il réponde à certaines touches clavier.

Afin que notre morph soit informé de la présence de la souris au-dessus de sa surface, celui-ci doit répondre **true** au message **handlesMouseOver:**

```
CrossMorph>>handlesMouseOver: evt
   ^true
```

Ce message est le pendant de **handlesMouseDown:** pour la position de la souris. Lorsque la souris entre dans la surface du morph, le message **mouseEnter:** est envoyé à celui-ci, lorsqu'elle sort de la surface le message **mouseLeave:** lui est envoyé. Nous définissons donc ces deux méthodes afin que notre morph attrape et libère le focus clavier :

```
CrossMorph>>mouseEnter: evt
   evt hand newKeyboardFocus: self


CrossMorph>>mouseLeave: evt
   evt hand newKeyboardFocus: nil
```

Maintenant que notre morph attrape et libère bien le focus clavier en fonction de la position de la souris au dessus de sa surface, nous pouvons maintenant traiter la réception des événements clavier eux-mêmes :

```
CrossMorph>>handleKeystroke: evt
   | keyValue |
   keyValue := evt keyValue.
   keyValue = 30
      ifTrue: [self position: self position - (0 @ 1)].
   keyValue = 31
      ifTrue: [self position: self position + (0 @ 1)].
   keyValue = 29
      ifTrue: [self position: self position + (1 @ 0)].
   keyValue = 28
      ifTrue: [self position: self position - (1 @ 0)]
```

Nous avons ici programmé la méthode afin de permettre le déplacement de la croix à l'aide des touches fléchées du clavier. Notez bien que lorsque la souris n'est plus au-dessus du morph ce message n'est plus envoyé et donc le morph ne répond plus aux commandes clavier. Pour connaître les valeurs des touches clavier, il suffit d'ouvrir un *transcript* et d'ajouter la ligne de code **Transcript show: evt keyValue.** dans la méthode ci-dessus.

L'argument **evt** reçu par **handleKeystroke:** est une instance de la classe **KeyboardEvent**, sous classe de

MoprhicEvent. Parcourrez cette classe pour connaître les méthodes disponibles dans la gestion des événements clavier.

Animations avec des morph

Le système morphic intègre un système simple d'animation. Il se compose de deux éléments principaux : une méthode **step** appelée à intervalle régulier et **stepTime** une méthode qui précise la périodicité – en millième de seconde – d'appel de **step**.

Pour que notre croix rouge soit clignotante nous pouvons définir ces méthodes comme suit :

```
CrossMorph>>stepTime
   ^ 100
```

et

```
CrossMorph>>step
   (self color diff: Color black) < 0.1
      ifTrue: [self color: Color red]
      ifFalse: [self color: self color darker]
```

Interacteurs

Pour demander à l'utilisateur des informations la classe **FillInTheBlank** proposent quelques boîtes de dialogues prédéfinies. Le code suivant retourne la chaîne de caractères saisie par l'utilisateur :

```
FillInTheBlank request: 'Ton nom ?' initialAnswer: 'sans nom'
```

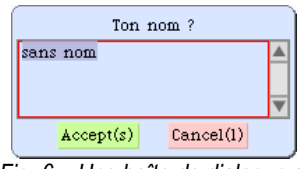

Fig. 6 – Une boîte de dialogue simple

Pour afficher un menu pop-up, la classe **PopupMenu** est disponible :

```
menu := PopUpMenu
   labelArray: #('rond' 'oval' 'carré' 'rectangle' 'triangle')
   lines: #(2 4).
menu startUpWithCaption: 'Choisir une forme'
```

Le tirer-déposer

Ce domaine n'est pas en reste dans morphic. Nous allons présenter un petit exemple simple avec deux morphs. Un morph receveur et un morph déposé. Le premier n'accepte de recevoir un morph qui si celui-ci répond à une condition donnée – ici le morph déposé doit être bleu – ensuite le morph déposé répond à un comportement donné s'il n'est pas accepté – ici il reste accroché au curseur souris.

Définissons d'abord le morph receveur :

```
Morph subclass: #ReceiverMorph
       instanceVariableNames: ''
       classVariableNames: ''
       poolDictionaries: ''
       category: 'LesMorph'
```

La méthode d'initialisation est classique :

```
ReceiverMorph>>initialize
       super initialize.
       color := Color red.
       bounds := 0 @ 0 extent: 200 @ 200
```

La méthode suivante répond un booléen, **aMorph** est le morph lâché sur le receveur. Une réponse **true** signifie que le morph est accepté.

```
ReceiverMorph>>wantsDroppedMorph: aMorph event: anEvent
       ^ aMorph color = Color blue
```

Le pendant pour indiquer si le morph est rejeté :

```
ReceiverMorph>>repelsMorph: aMorph event: ev
```

```
	^ (self wantsDroppedMorph: aMorph event: ev) not
```

C'est tout pour le receveur, nous pouvons déjà créer une instance depuis un *workspace* : **`ReceiverMorph new openInWorld`**. Si vous créez une instance de rectangle – non bleu – **`RectangleMorph new openInWorld`**, vous pouvez essayer de le je tirer-déposer dans le receveur. Il sera rejeté et renvoyé à sa position initiale. Pour changer ce comportement nous créons un morph spécifique, le **`DroppedMorph`** :

```
Morph subclass: #DroppedMorph
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'LesMorph'

DroppedMorph>>initialize
	super initialize.
	color := Color blue.
	self position: 250@100
```

Et maintenant, nous précisons ce que doit faire le morph en cas de rejet du receveur, à savoir rester accrocher au curseur souris :

```
DroppedMorph>>rejectDropMorphEvent: evt
	| h |
	h := evt hand.
	WorldState
		addDeferredUIMessage: [h grabMorph: self].
	evt wasHandled: true
```

`evt hand` retourne une instance de **`HandMorph`** qui est une représentation du curseur souris et de ce qu'il embarque avec lui. Nous indiquons au Monde que le curseur souris doit embarquer avec lui le morph – **`self`** – qui a été rejeté.

Créons deux instances de ce morph : **`DroppedMorph new color: Color blue`** et **`DroppedMoprh new color: Color green`**. Tirez-déposez ces deux morphs sur le receveur et voyez.

Un exemple concret de A à Z

Nous vous proposons de concevoir un morph représentant un dé à jouer. Il fera défiler les valeurs de ces différentes faces, un clic sur celui-ci arrêtera le défilement, un nouveau clic le relancera.

Fig. 7 – Le dé programmé avec Morph.

Nous définissons le dé comme un sous morph de **`BorderedMorph`** et non pas **`Morph`**, nous utiliserons en effet certains attributs de bordure de ce premier. Dans la catégorie de classe **`LesMorph`** notre **`DiceMorph`** est défini comme suit :

```
BorderedMorph subclass: #DiceMorph
   instanceVariableNames: 'face diceValue isStopped'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'LesMorph'
```

Les variables d'instance **`face`**, **`diceValue`** et **`isStopped`** définissent respectivement le nombre de faces du dé, la face affichée et si le dé est en défilement. Pour créer une instance d'un dé, nous définissons la méthode de classe **`face:`** (à définir dans la partie ***class*** du navigateur de classe) :

```
DiceMorph class>>face: aNumber
	^ self new face: aNumber
```

La méthode **`initialize`** – appelée par **`new`** – est définie dans la partie instance du navigateur :

```
DiceMorph>>initialize
   super initialize.
   self extent: 50 @ 50.
   self useGradientFill; borderWidth: 2; useRoundedCorners.
   self setBorderStyle: #complexRaised.
```

```
   self fillStyle direction: self extent.
   self color: Color green.
   diceValue := 1.
   face := 6.
   isStopped := false
```

Nous utilisons quelques méthodes de **BorderedMorph** pour donner un aspect sympathique à notre dé : bordure épaisse avec un effet de relief, coins arrondis, dégradé de couleur sur la face du dé.

Ensuite la méthode d'instance **face:** est définie comme suit :

```
DiceMorph>>face: anObject
   "Set the number of face"
   (anObject isInteger
         and: [anObject > 0]
         and: [anObject <= 9])
      ifTrue: [face := anObject]
```

Notre dé peut comporter jusqu'à 9 faces !

Comprenez bien l'ordre dans lequel les méthodes sont appelées lors de la création d'un dé avec l'expression (**DiceMorph face: 9**) :

1. dans la méthode de classe **face:** la méthode **new** est appelée ;
2. la méthode **new** appelle la méthode d'instance **initialize**, dans celle-ci **face** est initialisée à **6** ;
3. de retour dans la méthode de *classe* **face:** la méthode *d'instance* **face:** est cette fois-ci appelée, **face** est réinitialisée à **9**.

Avant de passer à la définition de la méthode **drawOn:**, nous avons besoin de quelques méthodes supports pour le positionnement des ronds selon la face affichée :

```
DiceMorph >>face1
   ^{0.5@0.5}
DiceMorph>>face2
   ^{0.25@0.25 . 0.75@0.75}
DiceMorph>>face3
   ^{0.25@0.25 . 0.75@0.75 . 0.5@0.5}
DiceMorph>>face4
   ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75}
DiceMorph>>face5
   ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.5@0.5}
DiceMorph>>face6
   ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5}
DiceMorph>>face7
   ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 . 0.5@0.5}
DiceMorph >>face8
   ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 . 0.5@0.5 .
0.5@0.25}
DiceMorph >>face9
   ^{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 . 0.75@0.5 . 0.5@0.5 .
0.5@0.25 . 0.5@0.75}
```

Ces méthodes définissent des collections de coordonnées des ronds de face dans un carré de dimension 1 par 1. Il nous suffira de faire une mise à l'échelle pour placer nos points.

Notre méthode **drawOn:** fait alors deux choses : dessiner le fond du dé par l'appel **super** et le dessin des ronds eux-mêmes :

```
DiceMorph>>drawOn: aCanvas
   super drawOn: aCanvas.
   (self perform: ('face' , diceValue asString) asSymbol)
      do: [:aPoint | self drawDotOn: aCanvas at: aPoint]
```

La deuxième partie de cette méthode utilise une capacité réflexive de Smalltalk.

Pour déterminer la collection de coordonnées des ronds de la face – valeur retournée par les méthodes **faceX** – nous utilisons la méthode **perform:** qui permet d'envoyer au receveur un message construit sachant une chaîne de caractères – ici **('face', diceValue asString) asSymbol**. Cette utilisation de la méthode **perform:** est assez courante, vous la rencontrerez souvent.

Nous énumérons la collection et pour chacune des coordonnées – instances de **Point** – nous envoyons le message **drawDotOn:at:**.

```
DiceMorph>>drawDotOn: aCanvas at: aPoint
   aCanvas
```

```
      fillOval: (Rectangle
         center: self position + (self extent * aPoint)
         extent: self extent / 6)
      color: Color black
```

Comme les coordonnées sont normalisées dans [0 ; 1], nous mettons à l'échelle des dimensions de notre dé : **`(self extent * aPoint)`**.

Déjà nous pouvons créer une instance de dé depuis un *Workspace* :
`monDe := (DiceMorph face: 6) openInWorld`. Pour changer la valeur affichée par le dé, nous créons un accesseur :

```
DiceMorph>>diceValue: anObject
   (anObject isInteger
         and: [anObject > 0]
         and: [anObject <= face])
      ifTrue:
         [diceValue := anObject.
         self changed]
```

que nous utilisons comme **`monDe diceValue: 4`**.

Maintenant pour programmer le défilement automatique des faces du dé, nous utilisons le système d'animation :

```
DiceMorph>>stepTime
   ^ 100
DiceMorph>>step
   isStopped ifFalse: [self diceValue: (1 to: face) atRandom]
```

Maintenant les faces de notre dé défilent !

Pour programmer l'arrêt/démarrage du défilement par un clic, il nous faut réutiliser ce que nous avons vu sur les événement souris. D'abord on active la réception des événements de type clic de souris sur le morph :

```
DiceMorph>>handlesMouseDown: anEvent
   ^ true
```

Puis nous gérons le clic souris par lui-même :

```
DiceMorph>>mouseDown: evt
   evt redButtonPressed
      ifTrue: [isStopped := isStopped not]
```

Voilà c'est tout pour l'essentiel ! Une grande partie de l'élaboration de notre objet a pu se faire tout en ayant une instance de celui-ci en cours de fonctionnement, c'est relativement confortable pour la mise au point

L'exemple complet à télécharger comprend quelques méthodes supplémentaires mais l'essentiel à été exposé ici. Créer des morph interactifs n'est pas compliqué, comme d'habitude nous capitalisons au maximum sur l'existant !

Encore quelques mots sur le canevas

La méthode **`drawOn:`** a comme unique argument un canevas, instance de **`Canvas`**. Un canevas est l'espace sur lequel est dessiné le morph, en utilisant les méthodes graphiques du canevas, vous avez toute la liberté de lui donner l'apparence que vous souhaitez.

En parcourant la hiérarchie de la classe **`Canvas`**, vous constaterez qu'il existe plusieurs variantes. Celle utilisée par défaut est **`FormCanvas`**. Vous trouverez les méthodes graphiques entre ces deux classes. Ces méthodes permettent de dessiner point, ligne, polygone, rectangle, ellipse, texte, image (avec possibilité de rotation et changement d'échelle).

Il est possible d'utiliser d'autres canevas, pour avoir des morphs en alpha transparence, des méthodes graphiques supplémentaires, de l'*antialising*, etc. Pour cela vous devez utiliser des canevas comme **`AlphaBlendingCanvas`** ou encore **`BalloonCanvas`**. Mais comment faire pour les utiliser puisque la méthode **`drawOn:`** reçoit une instance de **`FormCanvas`** ?

En fait il est possible de transformer un type de canevas en un autre. Reprenons notre exemple du dé à jouer, pour utiliser un canevas avec une alpha transparence de 0.5, il nous suffit de récrire la méthode **`drawOn:`** comme suit :

```
DiceMorph>>drawOn: aCanvas
   | theCanvas |
   theCanvas := aCanvas asAlphaBlendingCanvas: 0.5.
```

```
    super drawOn: theCanvas.
    (self perform: ('face' , diceValue asString) asSymbol)
       do: [:aPoint | self drawDotOn: theCanvas at: aPoint]
```

C'est tout ! Par curiosité, visitez la méthode **`asAlphaBlendingCanvas:`**.

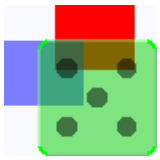

Fig. 8 – Notre dé rendu en alpha-transparence.

Pour utiliser l'*antialiasing*, il faudra utiliser le canevas **`BalloonCanvas`** et transformer la méthode d'affichage des ronds de notre dé :

```
DiceMorph>>drawOn: aCanvas
    | theCanvas |
    theCanvas := aCanvas asBalloonCanvas aaLevel: 3.
    super drawOn: aCanvas.
    (self perform: ('face' , diceValue asString) asSymbol)
       do: [:aPoint | self drawDotOn: theCanvas at: aPoint]
```

et l'utilisation de la méthode avec *antialiasing* :

```
DiceMorph>>drawDotOn: aCanvas at: aPoint
    aCanvas
       drawOval: (Rectangle
          center: self position + (self extent * aPoint)
          extent: self extent / 6)
       color: Color black
       borderWidth: 0
       borderColor: Color transparent
```

Le **`BalloonCanvas`** est dans un état qui n'est pas finalisé, il est donc possible que certaines méthodes graphiques aient un comportement inattendu. L'exemple de l'étiquette présenté précédemment utilise **`BalloonCanvas`**, cela peut donner quelques idées supplémentaires sur son utilisation.

Conclusions

A écrire... Parler de Tweak?, Rome,

Liens

- Les codes sources des exemples à télécharger :http://squeak.ofset.org/LM90
- Le site officiel : http://www.squeak.org/
- Le Wiki de la communauté : http://minnow.cc.gatech.edu/
- Le Wiki de la communauté française : http://community.ofset.org/wiki/Squeak
- Le groupe des utilisateurs européens de Smalltalk (European Smalltalk User Group). L'adhésion est gratuite : http://www.esug.org/
- Des livres gratuits en ligne sur Smalltalk et Squeak : http://www.iam.unibe.ch/~ducasse/FreeBooks.html
- Un livre sur Squeak en français: http://www.iam.unibe.ch/~ducasse/Books.html : Squeak, X. Briffault et S. Ducasse, Eyrolles, 2002.