11

Le développement d'interfaces graphiques

L'interface graphique d'une application constitue la couche externe de cette dernière, celle avec laquelle l'utilisateur interagit. C'est dire son importance.

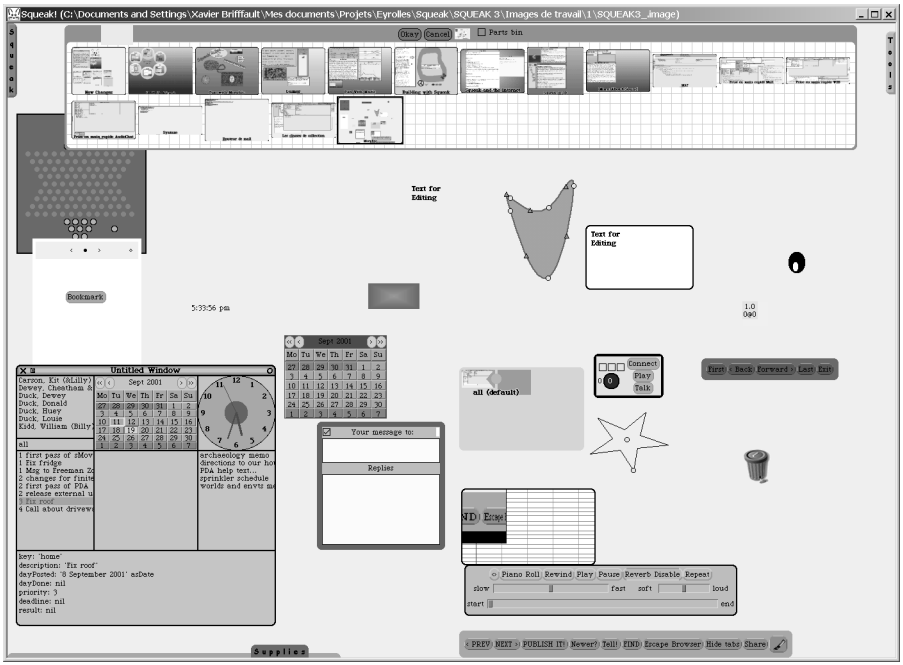
Deux types de modèles sont disponibles en Squeak pour le développement de ces interfaces graphiques : le modèle *MVC* (pour modèle-vue-contrôleur), historiquement le plus ancien, dont les classes sont regroupées dans les applications dont le nom débute par ST80, et le modèle *Morphic*, issu de travaux plus récents sur le langage Self[1], de Sun (John Maloney et Randy Smith), dont les classes sont regroupées dans les catégories dont le nom débute par Morphic. Nous ne présenterons pas ici le modèle MVC, moins utilisé que Morphic en Squeak, et dont de nombreuses présentations sont disponibles par ailleurs[2].

L'interface Morphic

L'un des avantages majeurs que présente Morphic est que tout objet graphique peut être une « fenêtre », c'est-à-dire une entité graphique autonome dotée d'un comportement, à la différence de la plupart des environnements graphiques où les fenêtres rectangulaires traditionnelles sont les composants de plus haut niveau, dans lesquelles doivent s'insérer les autres éléments. Dans l'exemple de la figure 11-1, tous les objets graphiques visibles sont des morphs.

1. http://www.sun.com/research/self/, http://minnow.cc.gatech.edu:8080/squeak/uploads/self-4.0-ui-framework.pdf
2. Le lecteur peut par exemple se reporter à notre précédent ouvrage, *Smalltalk*, paru aux éditions Eyrolles.

Figure 11–1
Quelques morphs



Tous ces morphs, et bien d'autres encore, peuvent être obtenus à partir de l'option <new morph> du menu World.

Qu'est-ce qu'un morph ?

Sur le plan fonctionnel, un morph (terme inspiré du mot grec signifiant *forme –morph*ologie) est un objet qui est à même d'interagir avec l'utilisateur, en entrée et en sortie, et de s'afficher sur un `Display`, l'objet Squeak représentant la zone d'affichage. Techniquement parlant, `Display` est l'unique instance de la classe `DisplayScreen`, catégorie `Graphics-Display Objects`). C'est l'utilisation de ce mécanisme qui rend l'interface graphique de Squeak totalement portable : les primitives graphiques ne font jamais appel aux spécificités de la machine physique, mais à des méthodes définies par le `Display`, celui-ci se chargeant de faire appel à des primitives de la machine virtuelle qui gèrent la liaison avec le matériel.

Plus précisément, un morph est capable :

- d'effectuer certaines actions en réponse à des entrées de l'utilisateur ;
- de gérer le glisser-déposer (*drag and drop*) depuis et vers d'autres morphs ;
- d'effectuer des actions planifiées à des intervalles ou des moments définis ;
- de contrôler le placement et la taille de ses sous-morphs.

Tout morph peut en effet être composé d'autres morphs (les sous-morphs), récursivement, et sans limitation de profondeur dans l'arbre de composition. Il est alors qualifié de morph composite. Un arbre de composition de morphs est considéré comme une unité autonome cohérente ; toute suppression, copie ou déplacement du morph composite entraîne celle de ses composants.

La métaphore traditionnelle des systèmes d'interfaces graphiques repose sur la notion de fenêtre (ils sont d'ailleurs souvent appelés systèmes de fenêtrage, et le nom du système d'exploitation de Microsoft l'illustre bien). Dans cette métaphore, les objets primaires avec lesquels interagit l'utilisateur sont des fenêtres, représentatives d'une application ou de fonctionnalités particulières d'une application.

Squeak, avec l'interface Morphic, généralise donc la notion d'interface graphique en permettant une granularité de représentation bien plus fine : tout objet peut avoir sa contrepartie visuelle interactive, et se présenter de la façon la mieux adaptée à l'utilisateur, et pas seulement sous la forme d'un rectangle doté de menus ou de barres d'outils.

Interfaces statiques

Toutes les méthodes graphiques de Squeak sont écrites en Squeak, ce qui facilite tant leur compréhension que leur modification. Elles sont par ailleurs totalement indépendantes du système d'exploitation et du matériel utilisé.

Morphs simples

Définition

Sur le plan technique, un morph est une instance d'une sous-classe de la classe `Morph`.

Créer un morph

La portion de code suivante crée par exemple un nouveau morph, `TestMorph`.

```
Morph subclass: #TestMorph
      instanceVariableNames: ''
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Exemples'
```

Ce nouveau morph est immédiatement opérationnel, et peut être activé en exécutant `TestMorph new openInWorld`. N'ayant aucune fonctionnalité particulière, il apparaît comme un simple carré vide. Il dispose néanmoins, comme tous les morphs, d'un halo qui permet de le piloter (voir figure 11-2).

Figure 11–2
Un morph et son halo

Comportement des morphs en réponse aux événements clavier/souris

Tous les morphs sont sensibles aux événements clavier et souris.

Plusieurs méthodes (définies de façon abstraite sur la classe `Morph`) permettent de paramétrer le comportement associé à ces événements en les redéfinissant dans les sous-classes. Par exemple, il suffit d'ajouter deux méthodes à `TestMorph` pour qu'il puisse traiter les événements souris, et définir un comportement associé à un événement.

`handlesMouseDown:` détermine si l'événement « clic souris » doit ou pas être pris en compte :

```
TestMorph>>handlesMouseDown: evt
   ^ true
```

`mouseDown:` définit le comportement que doit avoir le `Morph` en cas de clic souris. Pour les besoins de l'exemple, il se déplacera de 10 pixels vers la droite (ligne 2), et affichera son déplacement dans le `Transcript` (ligne 3) :

```
TestMorph>>mouseDown: evt
   self position: self position + (10 @ 0).
   Transcript cr; show: 'J''ai bougé'
```

Apparence d'un morph à l'écran

La méthode `drawOn:` spécifie la manière dont le morph doit s'afficher à l'écran.

Faisons par exemple en sorte qu'il s'affiche comme une ellipse colorée (voir figure 11-3) :

```
TestMorph>>drawOn: aCanvas
   | colors |
   colors := Color wheel: 10.
   colors
      withIndexDo: [:c :i |
            aCanvas
               fillOval: (self bounds insetBy: self width // 25 * i + 1)
               color: c]
```

Figure 11–3
Affichage modifié d'un morph

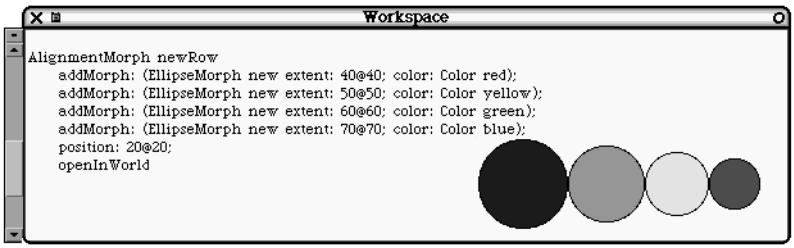
Morphs composites

Imbrication et placement des sous-morphs

Comme nous l'avons mentionné plus haut, tout morph peut avoir des composants, les sous-morphs. Le positionnement relatif de ces composants à l'intérieur du morph conteneur est géré par un morph de positionnement, un `AlignmentMorph`. L'exemple suivant en est une illustration (voir le résultat figure 11-4) :

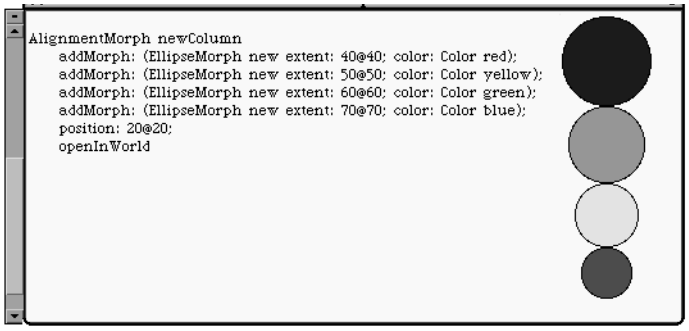
```
AlignmentMorph newRow
    addMorph: (EllipseMorph new extent: 40@40; color: Color red);
    addMorph: (EllipseMorph new extent: 50@50; color: Color yellow);
    addMorph: (EllipseMorph new extent: 60@60; color: Color green);
    addMorph: (EllipseMorph new extent: 70@70; color: Color blue);
    position: 20@20;
    openInWorld
```

Figure 11–4
Positionnement horizontal créé par un AlignmentMorph



Le même code, mais en remplaçant `newRow` par `newColumn`, aura le résultat visible à la figure 11-5.

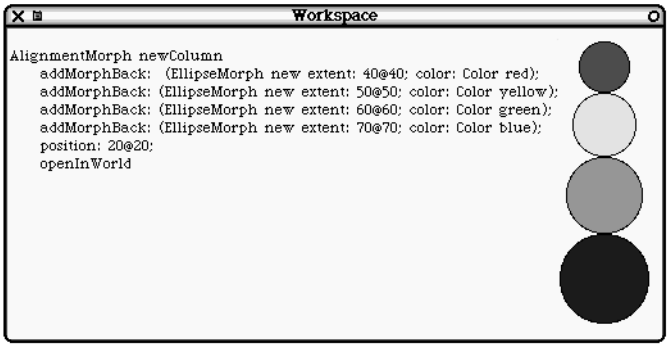
Figure 11–5
Positionnement vertical créé par un AlignmentMorph



En remplaçant `addMorph:` par `addMorphBack:`, on obtient le résultat de la figure 11-6.

Il est important de noter que tout objet Squeak peut être représenté comme un morph. À la classe `String` correspond par exemple la classe `StringMorph`. La portion de code suivante permet d'aligner verticalement des chaînes de caractères obtenues à partir de la liste des noms de catégories de classes de Squeak :

Figure 11–6
Gestion de l'ordre d'ajout des sous-morphs



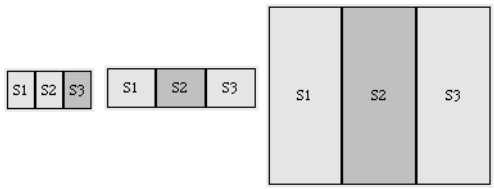

```
StringMorph>>test
   | c |
   c := AlignmentMorph newColumn.
   SystemOrganization categories
      do: [:cat | c addMorph: (StringMorph new contents: cat)].
   ^ c
```

Un aperçu du résultat est donné en figure 11-7.

Figure 11–7
Alignement de StringMorph

Un exemple plus affiné est donné avec la portion de code suivante, qui crée une interface dotée de trois boutons booléens, dont la taille varie en fonction de celle du morph qui les contient (figure 11-8).

Figure 11–8
Trois états différents d'un composite de PluggableButtonMorph




```
1  | s1 s2 s3 b1 b2 b3 row |
2  s1 := Switch new.
3  s2 := Switch new turnOn.
4  s3 := Switch new.
5  s2 onAction: [s3 turnOff].
6  s3 onAction: [s2 turnOff].
7  b1 := (PluggableButtonMorph
8         on: s1
9         getState: #isOn
10        action: #switch) label: 'S1'.
11 b2 := (PluggableButtonMorph
12        on: s2
13        getState: #isOn
14        action: #turnOn)label: 'S2'.
15 b3 := (PluggableButtonMorph
16        on: s3
17        getState: #isOn
18        action: #turnOn) label: 'S3'.
19 b1 hResizing: #spaceFill;
20    vResizing: #spaceFill.
21 b2 hResizing: #spaceFill;
22    vResizing: #spaceFill.
23 b3 hResizing: #spaceFill;
24    vResizing: #spaceFill.
25 row := AlignmentMorph newRow
26    hResizing: #spaceFill;
27    vResizing: #spaceFill;
28    addAllMorphs: (Array with: b1 with: b2 with: b3); extent: 120 @ 35.
29 ^ row
```

Les lignes 2 à 4 permettent la création de trois objets booléens (classe `Switch`). Les lignes 5 à 6 spécifient les actions à effectuer pour chaque switch qui est sélectionné.

Les lignes 7 à 21 créent trois boutons booléens représentant visuellement les `Switch`. Les lignes 22 à 27 déterminent la manière dont la taille (verticale et horizontale) des boutons est déterminée. Trois valeurs sont possibles pour ce redimensionnement : `#rigid` (la taille ne variera pas), `#spaceFill` (le morph s'adaptera à l'espace disponible dans son morph conteneur), `#shrinkWrap` (le morph s'adaptera à la taille de ses composants). Les lignes 28 à 35 créent finalement le morph d'alignement qui sera utilisé pour placer horizontalement les boutons.

Implémentation

La classe `Switch` implémente une bascule booléenne dotée de deux blocs : le bloc à exécuter lorsque la bascule passe en *on* (`onAction`), et le bloc à exécuter lorsque la bascule passe en *off* (`offAction`) :

```
Model subclass: #Switch
      instanceVariableNames: 'on onAction offAction '
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Tools-Menus'
```

Développer un morph de présentation de listes

Nous avons présenté jusqu'alors des applications à forte dominante graphique. Morphic peut évidemment être également utilisée pour présenter des données textuelles.

L'exemple suivant illustre la création d'un morph de gestion de liste, et la création de menus. Pour cela, nous définissons une nouvelle classe, `ExempleDeListe`, qui contiendra une liste et l'élément de la liste sélectionné, et qui sera le modèle de l'interface :

```
Object subclass: #ExempleDeListe
       instanceVariableNames: 'list selectedIndex '
       classVariableNames: ''
       poolDictionaries: ''
       category: 'Morphic-Windows'
```

L'initialisation affecte une liste triée des noms de sous-classes de la classe `Collection` à `list` :

```
ExempleDeListe>>initialize
   list := (Collection withAllSubclasses collect: [:each | each name])
      asSortedCollection: [:a :b | a < b].
   selectedIndex := 1
```

Une méthode `openView` est ensuite définie pour permettre l'ouverture d'un `PluggableListMorph` sur un `ExempleDeListe` :

```
1  ExempleDeListe>>openView
2     | window aListMorph |
3     aListMorph := PluggableListMorph
4           on: self
5           list: #list
6           selected: #listIndex
7           changeSelected: #listIndex:
8           menu: #listMenu:.
9     aListMorph color: Color white.
10    window := SystemWindow labelled: 'Exemple de liste'.
11    window color: Color blue;
12          addMorph: aListMorph frame: (0 @ 0 corner: 1 @ 1).
13    ^ window openInWorldExtent: 380 @ 220
```

La méthode `on:list:selected:changeSelected:menu:` (lignes 4 à 8) permet de préciser au morph d'affichage de la liste les informations dont il a besoin pour gérer son affichage :

- le modèle (`on:`) ;
- la liste (`list:`) ;
- l'élément sélectionné (`selected:`) ;
- la méthode à déclencher lorsqu'un item de la liste est sélectionné (`changeSelected:`) ;
- et la méthode renvoyant le menu (`menu:`).

Une fenêtre qui contient le `PluggableListMorph` est alors créée, puis ouverte (lignes 10 à 14).

La création du menu associé à la visualisation de la liste est assurée par la méthode `listMenu:` :

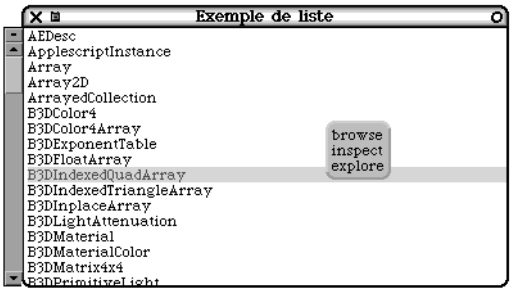
```
ExempleDeListe>>listMenu: aMenu
   | targetClass differentMenu className |
   className := list
                at: selectedIndex
                ifAbsent: [^ aMenu
                           add: 'rien n''est sélectionné'
                           target: self
                           selector: #beep].
   targetClass := Smalltalk
                at: className
                ifAbsent: [^ aMenu
                           add: 'Cette classe n''existe plus'
                           target: self
                           selector: #beep].
   differentMenu := DumberMenuMorph new.
   differentMenu
      add: 'browse'
      target: targetClass
      selector: #browse;

      add: 'inspect'
      target: targetClass
      selector: #inspect;

      add: 'explore'
      target: targetClass
      selector: #explore.
   ^ differentMenu
```

La méthode `add:target:selector:` permet de définir chaque entrée du menu : le premier paramètre (`add:`) est la chaîne à afficher dans le menu, le troisième (`selector:`) est le nom du message qui sera envoyé au deuxième paramètre (`target:`) lorsque l'option sera sélectionnée. Le résultat que l'on obtient finalement est illustré par la figure 11-9.

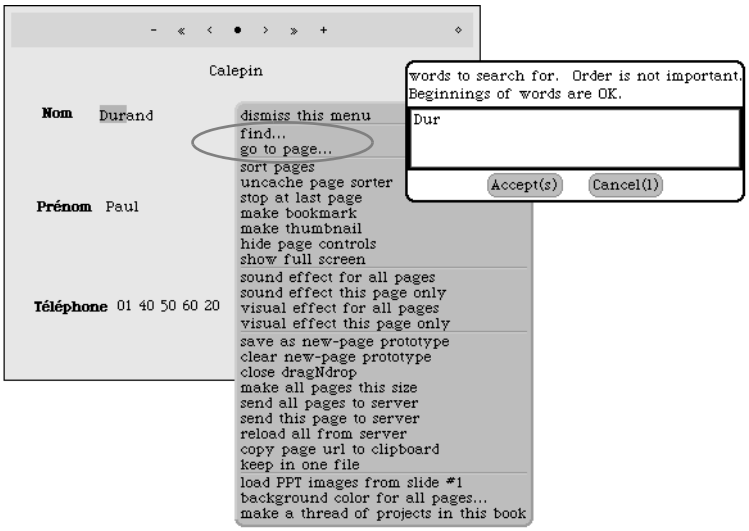
Figure 11–9
Exemple de morph sur une liste



Construction interactive d'un calepin

Dans la lignée des outils de visualisation de données textuelles, nous nous proposons maintenant de créer un calepin électronique ; il sera utilisé pour stocker, afficher et rechercher des informations personnelles sur des contacts (nom, adresse, téléphone…). Un exemple du type de calepin que nous allons définir est donné ci-après (voir figure 11-10).

Figure 11–10
Exemple de calepin



Nous profiterons pour cela des possibilités de création interactive d'interfaces de Squeak, qui ne nécessitent aucune programmation, en utilisant un `BookMorph`, objet spécialisé dans la gestion de pages structurées. La première opération consiste donc à créer un nouveau bookmorph. Il suffit pour cela de glisser-déposer l'icône correspondante de la barre d'outils <Supplies> (voir figure 11-11, flèche 1).

Pour définir les champs de chaque fiche, il suffit de faire glisser plusieurs `TextMorph` sur le `BookMorph` (voir figure 11-11, flèche 2). Le résultat apparaît à la figure 11-12.

Figure 11–11
Création d'un nouveau BookMorph

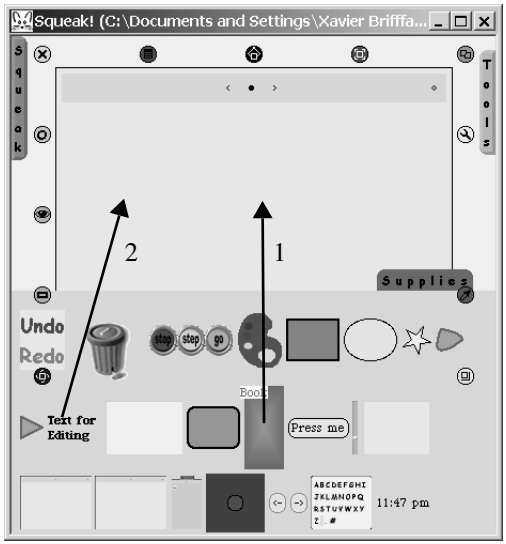
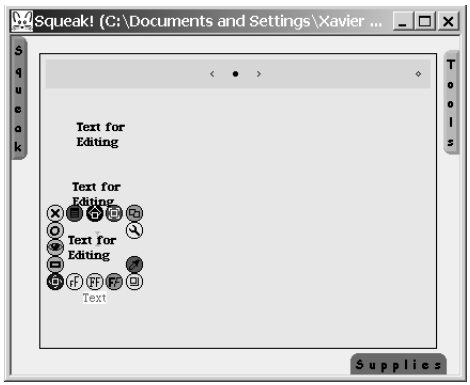


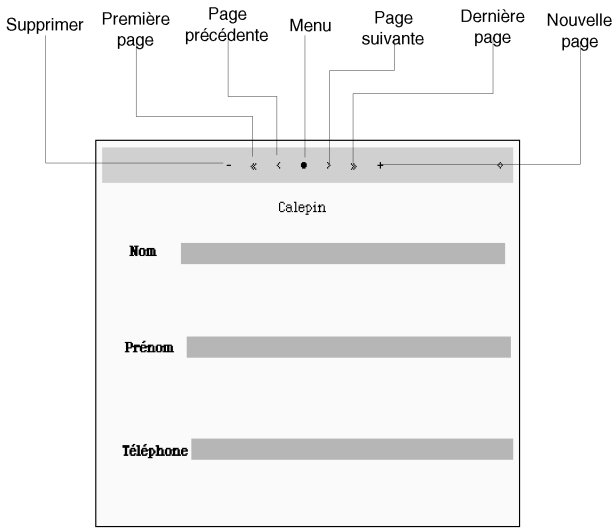
Figure 11–12
Création des champs des fiches du calepin



Chaque `TextMorph` est alors renommé, pour prendre le nom du champ qu'il représente, et des `TextMorph` vides sont ajoutés ; ils contiendront les valeurs correspondant aux champs (voir figure 11-10). Il convient alors de sauvegarder ce canevas général, qui constituera le prototype de toute nouvelle page ajoutée au calepin. On utilise pour ce faire l'option <save as new page prototype> du menu obtenu, en cliquant sur le point central de la barre de boutons (voir en figure 11-10 le menu).

Le calepin est dès lors prêt à fonctionner, en utilisant les boutons de la barre d'outils supérieure (figure 11-13). Le tout a été réalisé en moins d'un quart d'heure, sans aucune programmation.

Figure 11–13
Définition des champs des fiches du calepin



Interfaces dynamiques

Création et visualisation d'un film

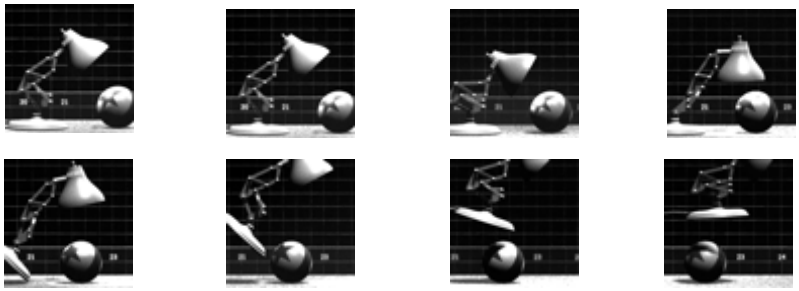
Squeak propose un ensemble de classes qui permettent de créer et de visualiser des films (dans un format interne propre à Squeak), regroupées dans la catégorie `Morphic-Demo`. Ce type de film Squeak contient en fait une série de `Form`, en format binaire non compressé (des lecteurs plus « sérieux » pour les formats MPEG et QuickTime existent également dans la version 3.1 de Squeak).

Le morph `MoviePlayerMorph` permet la visualisation des films. Il s'agit d'une sous-classe directe de `BookMorph`, dont voici la hiérarchie :

```
ProtoObject #()
   Object #()
      Morph #('bounds' 'owner' 'submorphs' 'fullBounds' 'color' 'extension')
      BorderedMorph #('borderWidth' 'borderColor')
         RectangleMorph #()
         AlignmentMorph #()
            BooklikeMorph #('pageSize' 'newPagePrototype')
            BookMorph #('pages' 'currentPage')
               MoviePlayerMorph #('movieFileName' 'movieFile' 'frameSize'
                     'frameDepth' 'frameNumber' 'frameCount' 'playDirection'
                     'msSinceStart' 'msAtStart' 'msAtLastSync'
                     'frameAtLastSync' 'msPerFrame' 'frameBufferIfScaled'
                     'soundTrackFileName' 'scorePlayer' 'soundTrackForm'
                     'soundTrackMorph' 'pianoRoll' 'cueMorph')
```

La création d'un film utilise des images fixes en format bmp (voir ci-après figure 11-14), stockées sur disque. …

Figure 11–14
Exemples d'images fixes utilisées pour construire un film



À partir de ces images, le code suivant permet de construire un fichier au format .movie, qui peut être utilisé par le `MoviePlayerMorph` :

```
| ps zps f32 out ff |
out := FileStream newFileNamed: 'luxo2.movie'.
out binary.
ff := Form extent: 64@64 depth: 32.
#(22 64 64 32 12 100000) , (7 to: 32) do: [:i | out nextInt32Put: i].
3 to: 36 by: 3 do:
   [:i | ps:=i printString. zps:=ps padded: #left to: 3 with: $0.
         f32 := Form fromFileNamed: 'luxo' , zps , '.bmp'.
         f32 displayOn: ff at: 0@0. "Convert down to 16 bits"
         ff display; writeOnMovie: out]. out close.
```

Les données numériques insérées en tête renseignent le `MoviePlayerMorph` sur le contenu du fichier (64 est la taille en x et y des images, 32 est la profondeur de chaque image, 12 le nombre d'images…). Le fichier .movie obtenu à partir du code précédent peut alors être relu par le `MoviePlayerMorph` au moyen de l'option <open as movie> du menu associé au `File List Browser`. `MoviePlayerMorph` dispose (voir figure 11-15) de boutons de contrôle pour la lecture du film. Une technique intéressante est utilisée pour construire l'ensemble des composants de cette interface : un langage de description littérale (une spécification) est utilisé par un mécanisme de construction pour créer l'interface. La spécification du `MoviePlayerMorph` se trouve dans la méthode `fullControlSpecs` :

```
MoviePlayerMorph>>fullControlSpecs
   ^ #(#('•' #invokeBookMenu 'Invoke menu')
       #('<--' #firstPage 'Go to first page')
       #('<<' #playReverse 'Play backward')
       #('<-' #previousPage 'Back one frame')
       #('| |' #stopPlay 'Stop playback')
       #('->' #nextPage 'Forward one frame')
       #('>>' #playForward 'Play forward')
       #('-->' #lastPage 'Go to final page')
       #('<->' #scanBySlider 'Scan by slider' 'menu') )
```

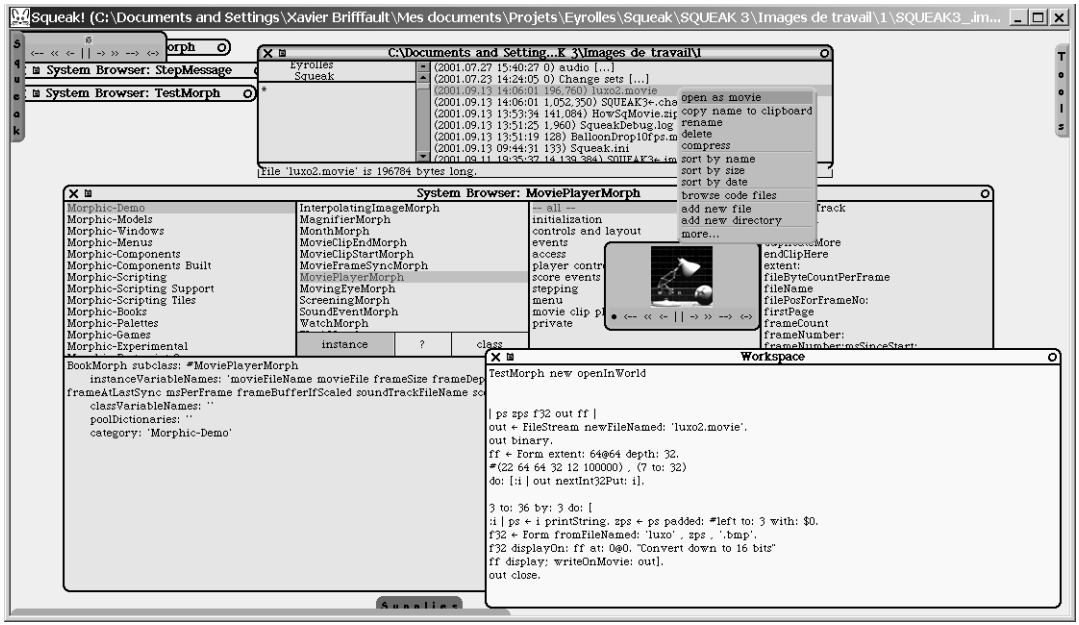


Figure 11–15

Le MoviePlayerMorph, visualisateur, browser, et un fichier .movie

Cette méthode renvoie un tableau de triplets, dont le premier élément contient la valeur qui est affichée comme label du sous-morph (•,<--,<<...), le deuxième contient la méthode à invoquer lorsque le sous-morph est activé, et le troisième le message d'information associé à la bulle d'aide.

Bulles d'aide

Des bulles d'aides peuvent être associées à tout morph. La méthode `setBallonText:` permet de préciser le texte qui doit être présenté. Par exemple :

```
World currentHand attachMorph: (RectangleMorph new setBalloonText:
   'Je suis une bulle d''aide spécifique à ce morph').
```

Cette spécification est ensuite utilisée par la méthode `buildFloatingPageControls` pour construire effectivement les sous-morphs :

```
BookMorph>>buildFloatingPageControls
   | pageControls |
   pageControls := self makePageControlsFrom: self fullControlSpecs.
   pageControls borderWidth: 0; layoutInset: 4.
   pageControls setProperty: #pageControl toValue: true.
```

```
    pageControls setNameTo: 'Page Controls'.
    pageControls color: Color yellow.
    ^ FloatingBookControlsMorph new addMorph: pageControls
```

Cette technique est intéressante, car elle permet de spécifier très rapidement des interfaces à plusieurs composants, sans qu'il soit nécessaire de développer directement le code Squeak correspondant.

Animation d'un morph

Le simple morph que nous avions commencé à définir avec la classe `TestMorph` peut être animé très facilement. Ajoutons-lui pour cela une variable d'instance, `trajectoire`, qui contiendra une liste de points qui définissent les positions successives du morph dans son déplacement :

```
Morph subclass: #TestMorph
      instanceVariableNames: 'trajectoire '
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Exemples'
```

La méthode d'initialisation associée est :

```
TestMorph>>initialize
   super initialize.
   trajectoire := OrderedCollection new
```

Définissons maintenant une méthode de gestion du déplacement, `demarreAnimation` :

```
TestMorph>>demarreAnimation
   trajectoire := OrderedCollection new.
   0
      to: 30
      do: [:i | trajectoire add: self position + (0 @ (10 * i))].
   trajectoire := trajectoire , trajectoire reversed.
   self startStepping
```

La méthode `startStepping` (dernière ligne ci-avant) est définie par la classe `Morph` ; elle provoque le déclenchement itératif de la méthode `step`, à redéfinir dans les sous-classes, qui définit l'activité à effectuer pour chaque étape du déplacement. `startStepping` fait appel à `startStepping:at:arguments:stepTime:` :

```
Morph>>startStepping
   self
      startStepping: #stepAt:
      at: Time millisecondClockValue
      arguments: nil
      stepTime: nil
```

```
Morph>>startStepping: aSelector at: scheduledTime arguments: args
stepTime: stepTime
   | w |
   w := self world.
   w ifNotNil: [w
                  startStepping: self
                  at: scheduledTime
                  selector: aSelector
                  arguments: args
                  stepTime: stepTime.
                  self changed]
```

`startStepping:at:arguments:stepTime:` fait ensuite appel à l'espace au « monde morphique » (une instance de `PasteUpMorph`) dans lequel évolue le morph :

```
Morph>>startStepping: aMorph at: scheduledTime selector: aSelector
arguments: args stepTime: stepTime
   worldState
      startStepping: aMorph
      at: scheduledTime
      selector: aSelector
      arguments: args
      stepTime: stepTime
```

`worldState`, une instance de `WorldState`, reçoit alors le message `startStepping:at:selector: arguments:stepTime:` :

```
Morph>>startStepping: aMorph at: scheduledTime selector: aSelector
arguments: args stepTime: stepTime
   self stopStepping: aMorph selector: aSelector.
   self adjustWakeupTimesIfNecessary.
   stepList
      add: (StepMessage
               scheduledAt: scheduledTime
               stepTime: stepTime
               receiver: aMorph
               selector: aSelector
               arguments: args)
```

Le message d'animation réifié (`StepMessage`) est ensuite ajouté à la `stepList` (liste des morphs à animer) du monde morphique, qui se charge de provoquer à intervalles réguliers, lesquels correspondent à la valeur renvoyée par la méthode `stepTime`, les actions d'animation pour chaque morph de la liste, à savoir le déclenchement de la méthode `step`.

Voici, en guise de synthèse, la définition complète du code d'animation de la classe `TestMorph` :

```
TestMorph>>demarreAnimation
   trajectoire:=OrderedCollection new.
   0
```

```
      to: 30
      do: [:i | trajectoire add: self position + (0 @ (10 * i))].
   trajectoire := trajectoire , trajectoire reversed.
   self startStepping

TestMorph>>drawOn: aCanvas
   | colors |
   colors := Color wheel: 10.
   colors withIndexDo: [:c :i |
            aCanvas
               fillOval: (self bounds insetBy: self width // 25 * i + 1)
               color: c]

TestMorph>>handlesMouseDown: evt
   ^ true

TestMorph>>initialize
   super initialize.
   trajectoire := OrderedCollection new

TestMorph>>mouseDown: evt
   self demarreAnimation

TestMorph>>step
   trajectoire size > 0
         ifTrue: [self position: trajectoire removeFirst]

TestMorph>>stepTime
   ^ 50
```

La méthode `step` assure le déplacement du `TestMorph` en modifiant sa position à partir de chacune des coordonnées contenues dans la liste `trajectoire`. Notons que, dans cette implémentation, l'arrêt de l'animation ne signifie pas que le morph soit supprimé de la `stepList`. Il continue à recevoir le message `step`, comme peut le montrer l'insertion d'un point d'arrêt dans cette méthode. Pour qu'un morph soit supprimé de la `stepList`, il faut lui envoyer le message `stopStepping`.

Les interfaces de construction d'interfaces

La programmation directe ou l'usage de spécifications littérales sont intéressants, voire indispensables, dans certaines situations. Dans d'autres cas, il est possible d'utiliser des outils de construction graphique d'interfaces qui assistent le concepteur dans sa démarche de création. Les interfaces Etoys en sont un exemple. Une interface EToys (également connue sous le nom de Viewer, que nous avons rapidement présenté dans le chapitre 2) peut être obtenue pour un morph donné à partir de l'icône bleue représentant un œil, au milieu à gauche, du halo du morph (voir figure 11-16). Les différentes sections du Viewer présentent divers types d'informations sur le morph. Le type d'information montré ci-après peut être sélectionné à partir du menu category.

Figure 11–16
Un morph et son interface Etoys (Viewer)

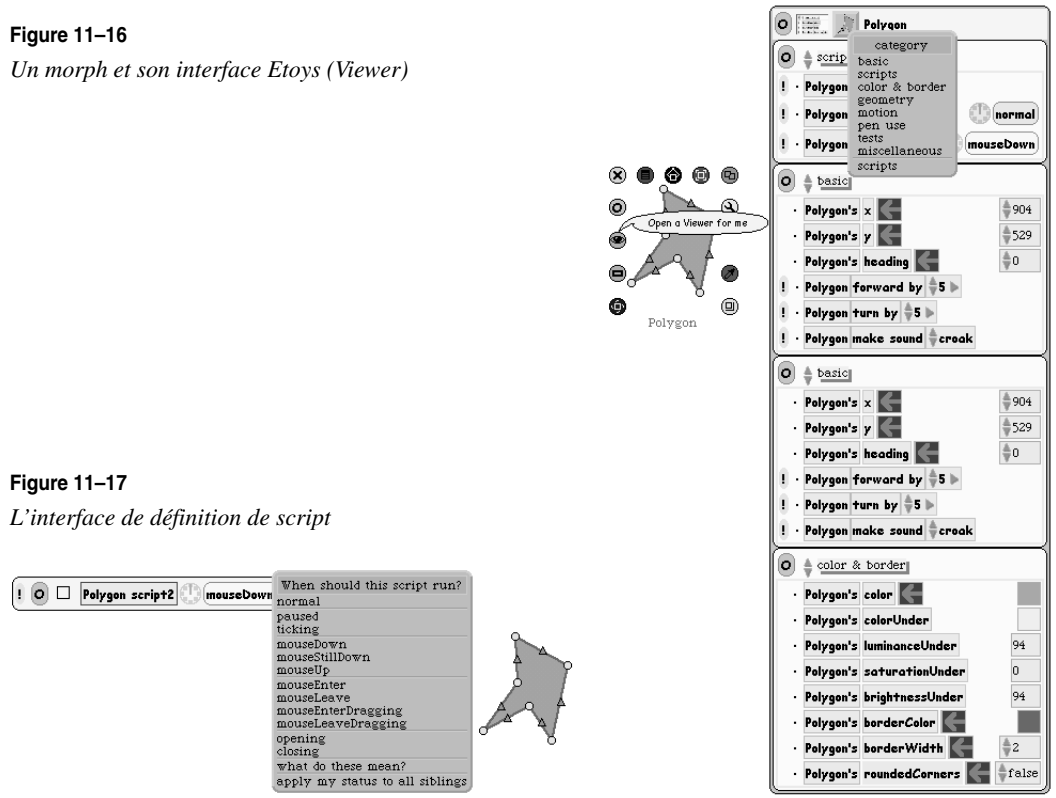


Figure 11–17
L'interface de définition de script

L'icône qui représente le morph dans le Viewer lui-même (en haut) permet d'accéder à une interface d'ajout de variables d'instances, de définition de scripts et d'ouverture d'outils de programmation. En sélectionnant l'option <add a new script>, l'interface de définition de script apparaît (voir figure 11-17). Un script est un petit programme qui définit un comportement du morph.

L'interface de définition permet de construire d'une façon interactive un tel script. Il suffit pour cela de glisser-déposer dans le script (après modification dans le Viewer) les éléments du morph dont les modifications feront partie du script. Sélectionnez pour cela la flèche orientée vers la gauche qui se trouve à côté de l'item sélectionné (voir figure 11-18), et faites-la glisser sur l'interface de construction du script. Cette définition interactive des actions du script génère en fait une méthode Squeak, qui sera sauvegardée sur une nouvelle classe de `Player`, un objet spécialisé dans l'exécution de scripts (figure 11-19). La forme textuelle du script peut évidemment être modifiée pour ajouter toute instruction Squeak de pilotage du morph.

Implémentation

La gestion des scripts est assurée par les classes des catégories `Morphic-Scripting`, `Morphic-Scripting Support` et `Morphic-Scripting Tiles`. `Player` en est la classe principale.

Figure 11–18
Construction interactive d'un script

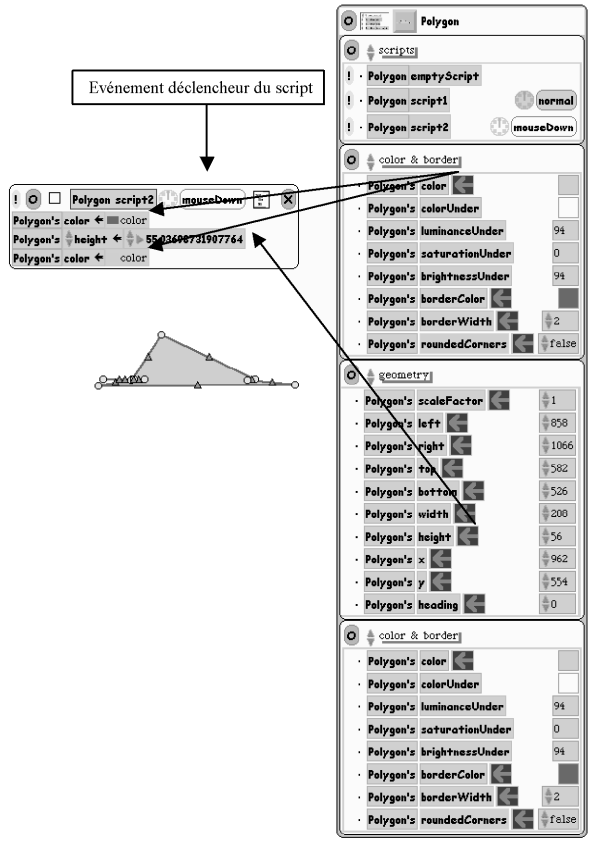
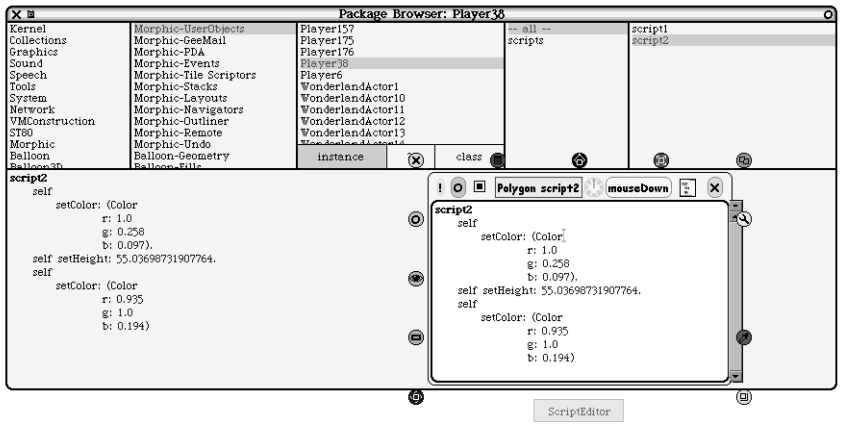


Figure 11–19
La forme textuelle d'un script et la méthode correspondante



Coordination des animations de morphs

Nous n'avons abordé jusqu'à présent que le cas de l'animation d'un unique morph. Les animations peuvent également être coordonnées, comme le montre la figure 11-20.

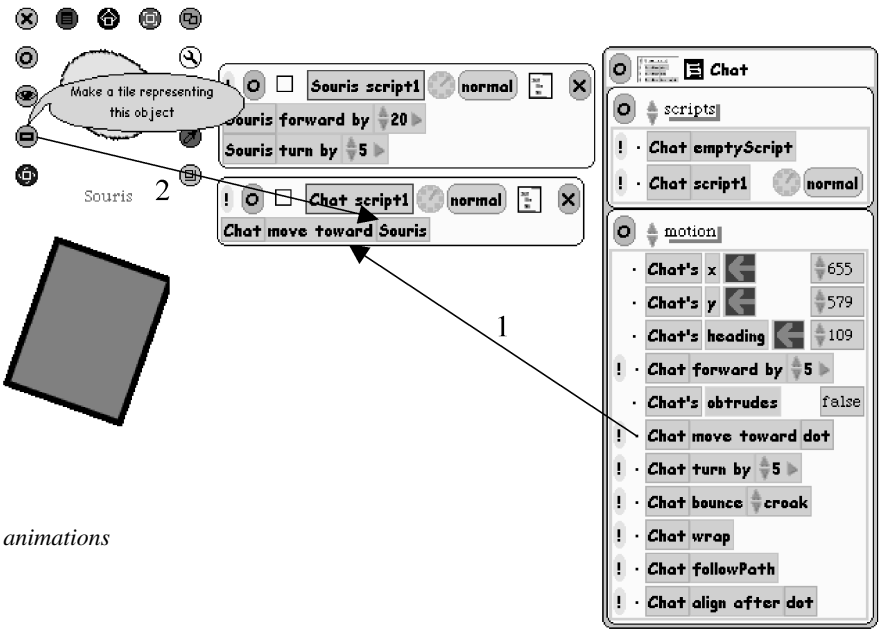


Figure 11–20

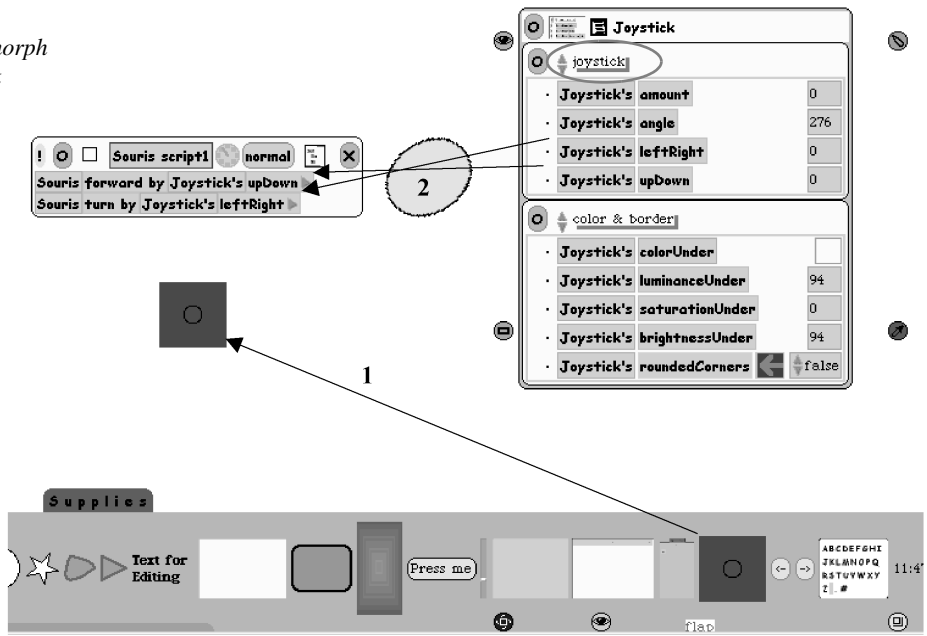
Coordination des animations de deux morphs

Dans cet exemple, deux morphs sont utilisés. L'un, nommé `souris`, est représenté par une ellipse, à laquelle est associé un script de déplacement (première fenêtre de script). L'autre, nommé `chat`, est représenté par un carré, doté d'un script qui lui permet de poursuivre le premier morph (deuxième fenêtre de script). L'action `move toward`, créée par le glisser-déposer de l'action `move toward` (flèche 1), indique au morph qu'il doit en suivre un autre, et est dotée de deux paramètres. Le premier, `chat`, est celui du morph porteur du script. Le deuxième, `souris`, a été créé par glisser-déposer du paramètre `tile` (un « carreau » représentant le morph) du morph `souris` (flèche 2). Tout déplacement du morph `souris` sera ainsi suivi par un déplacement correspondant du morph `chat`, qui essaiera de le rejoindre.

Pilotage d'un morph avec un joystick

Les morphs peuvent être animés, nous l'avons vu, par des scripts qui leur sont associés, ou au travers d'une coordination avec le comportement d'autres morphs. Ils peuvent aussi être pilotés directement par l'utilisateur, par exemple par l'intermédiaire d'un joystick, réel ou simulé. Nous considérerons ici le cas d'un joystick simulé, que nous utiliserons pour piloter la souris présentée dans l'exemple précédent. Le morph joystick est accessible à partir de la barre d'outils inférieure de Squeak, `Supplies` (figure 11-21).

Figure 11–21
Pilotage d'un morph avec un joystick



Pour créer un nouveau joystick, il suffit de faire glisser son icône sur l'environnement de travail (flèche 1). L'ouverture d'un Viewer sur ce joystick (figure 11-21, à droite) permet d'accéder aux paramètres fondamentaux du joystick (angle, quantité, gauche/droite, haut/bas).

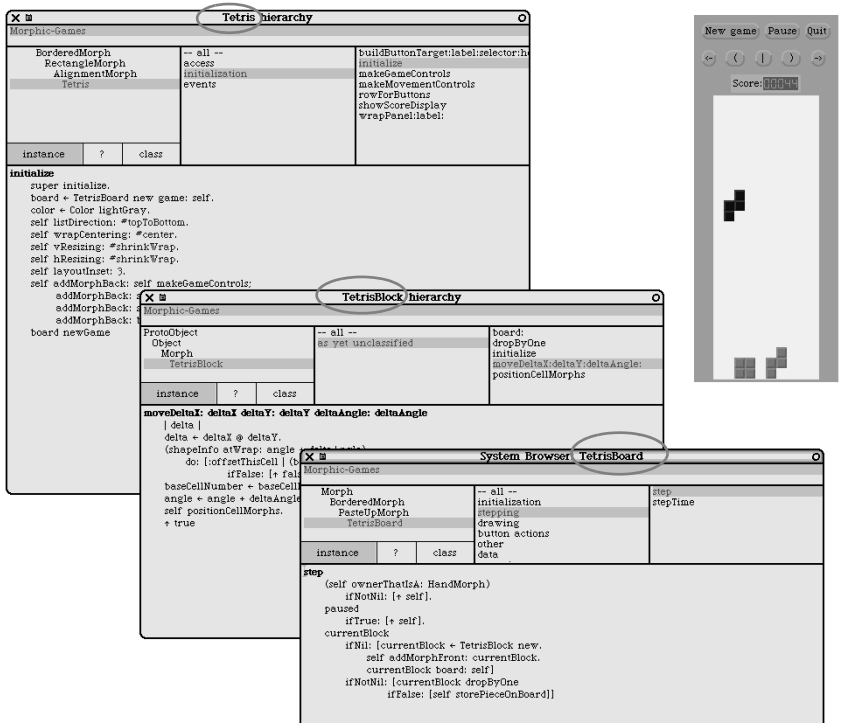
Il suffit alors de remplacer les paramètres numériques associés aux actions du script (`Souris forward by 20`, `Souris turn by 5`) par les paramètres correspondants du joystick qui les déterminent (flèches 2). En sélectionnant l'option « fonctionnement permanent » du script (option <ticking>) du morph souris qui active son déplacement, il est alors possible de piloter sa trajectoire en agissant sur le joystick.

Analyse d'un exemple complet : le jeu Tetris

Afin d'illustrer sur une application complète le fonctionnement de Morphic, nous abordons dans cette section l'analyse de l'implémentation en Squeak/Morphic d'un jeu bien connu, Tetris (catégorie `Morphic-Games`).

Ce jeu, dont un exemple est donné ci-après, a pour objet de positionner des blocs (constitués de cellules), qui tombent du haut de la fenêtre, de façon à constituer des lignes de cellules complètes. Lorsqu'une ligne complète de cellules est constituée, elle disparaît. Le jeu est terminé lorsque les blocs, empilés les uns sur les autres sans qu'aucune ligne complète de cellules n'ait pu être constituée, ne laissent plus de place disponible pour un nouveau bloc. Les blocs peuvent être déplacés latéralement à l'aide des flèches <droite>/<gauche>, tournés sur eux-mêmes d'un quart de tour dans le sens horaire ou anti-horaire avec les flèches <haut>

Figure 11–22
L'interface de Tetris et les trois classes qui l'implémentent



et <bas>, et leur chute peut être accélérée à l'aide de la touche espace (les boutons placés en haut de l'interface remplissent les mêmes fonctions).

Tetris est implémenté en Squeak par trois classes (voir les explorateurs hiérarchiques sur ces classes, figure 11-22) : `Tetris`, la classe principale, `TetrisBoard`, la zone d'affichage des blocs, et `TetrisBlock`, les blocs eux-mêmes.

L'initialisation du jeu et la création de l'interface

Le lancement de l'interface Tetris (par exemple, par `Tetris new openInWorld`, ou par l'option de création d'un morph dans le menu World), provoque l'exécution de la méthode d'initialisation :

```
Tetris>>initialize
   super initialize.
   board := TetrisBoard new game: self.
   color := Color lightGray.
   self listDirection: #topToBottom.
   self wrapCentering: #center.
   self vResizing: #shrinkWrap.
   self hResizing: #shrinkWrap.
```

```
    self layoutInset: 3.
    self addMorphBack: self makeGameControls;
         addMorphBack: self makeMovementControls;
         addMorphBack: self showScoreDisplay;
         addMorphBack: board.
    board newGame
```

Cette méthode crée les différents composants de l'interface et permet de démarrer le jeu. Après la création du « board » (la zone d'affichage des blocs, ligne 3), et la spécification des paramètres de redimensionnement (lignes 5-9), les sous-morphs correspondant respectivement aux boutons de contrôle du jeu, aux boutons de contrôle des déplacements des blocs, à l'affichage du score et à la zone d'affichage des blocs sont ajoutés au morph `Tetris` (lignes 10-13). On démarre ensuite le jeu en envoyant le message `newGame` au `board` créé.

Les méthodes `makeGameControls`, `makeMovementControls` et `showScoreDisplay` ont pour fonction d'ajouter les sous-morphs de contrôle du jeu, de contrôle des pièces et d'affichage du *board* :

```
Tetris>>makeGameControls
   ^ self rowForButtons
      addMorph: (self
                  buildButtonTarget: self
                  label: 'Quit'
                  selector: #delete
                  help: 'quit');

      addMorph: (self
                  buildButtonTarget: board
                  label: 'Pause'
                  selector: #pause
                  help: 'pause');

      addMorph: (self
                  buildButtonTarget: board
                  label: 'New game'
                  selector: #newGame
                  help: 'new game')
```

La méthode `buildButtonTarget:label:selector:help:` crée un bouton d'action, en précisant l'objet qui supporte l'action du bouton (`target`), le label à afficher (`label`), le sélecteur de la méthode à déclencher (`selector`), et le texte associé à la bulle d'aide (`help`). Elle est définie de la façon suivante :

```
Tetris>>buildButtonTarget: aTarget label: aLabel selector: aSelector help:
aString
   ^ self rowForButtons
      addMorph: (SimpleButtonMorph new
                  target: aTarget;
                  label: aLabel;
```

```
                    actionSelector: aSelector;
                    borderColor: #raised;
                    borderWidth: 2;
                    color: color)
```

Les deux autres méthodes, `makeMovementControls` et `showScoreDisplay`, sont définies de la façon suivante :

```
Tetris>>makeMovementControls
   ^ self rowForButtons
      addMorph: (self
                    buildButtonTarget: board
                    label: '->'
                    selector: #moveRight
                    help: 'move to the right');

      addMorph: (self
                    buildButtonTarget: board
                    label: ' ) '
                    selector: #rotateClockWise
                    help: 'rotate clockwise');

      addMorph: (self
                    buildButtonTarget: board
                    label: ' | '
                    selector: #dropAllTheWay
                    help: 'drop');

      addMorph: (self
                    buildButtonTarget: board
                    label: ' ( '
                    selector: #rotateAntiClockWise
                    help: 'rotate anticlockwise');

      addMorph: (self
                    buildButtonTarget: board
                    label: '<-'
                    selector: #moveLeft
                    help: 'move to the left')

Tetris>>showScoreDisplay
   ^ self rowForButtons hResizing: #shrinkWrap;
         addMorph: (self wrapPanel:
                        ((scoreDisplay := LedMorph new) digits: 5;
                        extent: 4 * 10 @ 15) label: 'Score:')
```

Quant à `rowForButtons`, elle utilise un `AlignmentMorph` pour placer les boutons horizontalement :

```
Tetris>>rowForButtons
   ^ AlignmentMorph newRow color: color;
         borderWidth: 0;
         layoutInset: 3;
         vResizing: #shrinkWrap;
         wrapCentering: #center
```

Le lancement du jeu et la gestion de l'affichage

La méthode `newGame` de la classe `TetrisBoard` est très simple ; elle ne fait qu'initialiser les principales variables et positionner les booléens qui permettent au cycle d'animation de fonctionner.

```
TetrisBoard class>>newGame
   self removeAllMorphs.
   gameOver := paused := false.
   delay := 500.
   currentBlock := nil.
   self score: 0
```

En effet, en tant que `morph` présent à l'écran, l'interface `Tetris` bénéficie automatiquement du mécanisme de gestion de l'activation des animations qui a été présenté (la `stepList`). La méthode `step`, définie sur `TetrisBoard`, sera donc activée (se reporter au test sur la variable `paused`, ligne 3). Elle est définie de la façon suivante :

```
1  TetrisBoard>>step
2     (self ownerThatIsA: HandMorph)ifNotNil: [^ self].
3     paused ifTrue: [^ self].
4     currentBlock
5        ifNil: [currentBlock := TetrisBlock new.
6                self addMorphFront: currentBlock.
7                currentBlock board: self]
8        ifNotNil: [currentBlock dropByOne
9                ifFalse: [self storePieceOnBoard]]
```

Deux cas sont envisagés dans cette méthode : s'il n'y a pas de bloc en train de descendre, il faut alors en créer un (lignes 5-7) ; s'il y en a un, il faut le faire avancer d'une unité (lignes 8-9). Dans le cas où le bloc doit être créé, la sélection aléatoire de la forme de ce bloc est assurée par la méthode `initialize` de `TetrisBlock`, qui fait appel à une sélection aléatoire sur une collection de formes renvoyées par la méthode de classe `shapeChoices` :

```
TetrisBlock>>initialize
   super initialize.
   bounds := (2 @ 2) negated extent: 1 @ 1.
   shapeInfo := self class shapeChoices atRandom.
   baseCellNumber := 4 atRandom + 2 @ 1.
   angle := 4 atRandom.
   color := Tetris colors atRandom
```

Le cas où le bloc existe, et doit être déplacé, est géré par la méthode `dropByOne`, qui est en fait un appel à la méthode `moveDeltaX:deltaY:deltaAngle:` :

```
TetrisBlock>>dropByOne
   ^ self
      moveDeltaX: 0
      deltaY: 1
      deltaAngle: 0

TetrisBlock>>moveDeltaX: deltaX deltaY: deltaY deltaAngle: deltaAngle
   | delta |
   delta := deltaX @ deltaY.
   (shapeInfo atWrap: angle + deltaAngle)
      do: [:offsetThisCell |
            (board emptyAt: baseCellNumber + offsetThisCell + delta)
               ifFalse: [^ false]].
   baseCellNumber :=baseCellNumber + delta.
   angle := angle + deltaAngle - 1 \\ 4 + 1.
   self positionCellMorphs.
   ^ true
```

Le positionnement effectif des sous-morphs qui représentent les blocs est assuré par une réaffectation de leur variable `position`, dans la méthode `positionCellMorphs` (ligne 4) :

```
1  TetrisBlock>>positionCellMorphs
2     (shapeInfo atWrap: angle)
3        withIndexDo: [:each :index | (submorphs at: index)
4              position: (board originForCell: baseCellNumber + each)].
5     fullBounds := nil.
6     self changed
```

Dans le cas où l'utilisateur aurait actionné l'un des boutons qui commandent le déplacement du bloc en chute, la prise en considération de cette commande aurait été assurée par l'exécution d'une des méthodes correspondantes de `TetrisBoard`. Par exemple, pour le déplacement vers la gauche et la rotation anti-horaire :

```
TetrisBoard>>moveLeft
   self running
      ifFalse: [^ self].
   currentBlock
      moveDeltaX: -1
      deltaY: 0
      deltaAngle: 0

TetrisBoard>>rotateAntiClockWise
   self running
      ifFalse: [^ self].
   currentBlock
      moveDeltaX: 0
      deltaY: 0
      deltaAngle: -1
```

Le dernier problème à gérer est l'effacement des lignes complètes et la comptabilisation du score. Cela s'effectue au moyen de la méthode `storePieceOnBoard`, appelée à la fin de la méthode `step` :

```
TetrisBoard>>storePieceOnBoard
   currentBlock submorphs
      do: [:each |
            self addMorph: each.
            each top - self top // self cellSize y < 3
               ifTrue: [paused := gameOver := true]].
   currentBlock delete.
   currentBlock := nil.
   self checkForFullRows.
   self score: score + 10.
   delay := delay - 2 max: 80
```

La ligne 10 assure l'incrémentation du score, tandis que la ligne 11 diminue le délai entre chaque mouvement de bloc pour augmenter la difficulté du jeu. Avec la méthode `checkForFullRows` (présentée ci-après), on réalise la suppression des lignes complètes et le décalage vers le bas des blocs qui en résulte, et on gère le bonus pour chaque ligne détruite :

```
TetrisBoard>>checkForFullRows
   | targetY morphsInRow bonus |
   self numRows
      to: 2
      by: -1
      do: [:row |
         targetY :=(self originForCell: 1 @ row) y.
         [morphsInRow :=self
         submorphsSatisfying: [:each | each top = targetY].
         morphsInRow size = self numColumns]
            whileTrue: [bonus :=(morphsInRow
         collect: [:each | each color]) asSet size = 1
               ifTrue: [1000]
               ifFalse: [100].
      self score: score + bonus.
      submorphs copy
         do: [:each |
               each top = targetY
                  ifTrue: [each delete].
               each top < targetY
                  ifTrue: [each position: each position +
                           (0 @ self cellSize y)]]]]
```

Quelques autres méthodes annexes sont également utilisées, que le lecteur aura tout le loisir de découvrir, utilisant l'application Tetris et les autres jeux disponibles.

En résumé

Deux technologies d'interfaces sont disponibles en Squeak : la technologie MVC (modèle-vue-contrôleur), la plus ancienne et la moins utilisée en Squeak, que nous avons préféré laisser de côté, et la technologie Morphic, que nous avons présentée en détail.

Composants graphiques mobiles de base de Morphic, les morphs sont entièrement développés en Squeak, ce qui a pour effet de rendre l'interface graphique complètement portable. Nous avons présenté les principes de base de création, d'affichage et d'animation des morphs, ainsi que leurs possibilités d'agrégation, qui permettent de construire des interfaces composites, de façon programmatique ou à l'aide d'outils d'aide au développement (les Etoys).

L'analyse complète d'un jeu graphique interactif (Tetris) a permis de voir comment les différents éléments techniques pouvaient être articulés ensemble pour produire une application complète. La construction interactive d'une application de gestion de calepin, réalisée en quelques minutes, nous a permis de démontrer la productivité qui pouvait être atteinte en combinant tous les outils de Squeak.