The Art of Morph - (DRAFT)

Craft your own Morph

> The screen is a window through which one sees a virtual world. The challenge is to make that world look real, act real, sound real, feel real.
>
> —*Ivan Sutherland*

Hilaire Fernandes

"The Art of Morph - (DRAFT)" booklet is for Cuis-Smalltalk (7.0 or later), a free and modern implementation of the Smalltalk language and environment.

Thanks to Mark Volkmann to share his examples.
Thanks to ... for the reviews of the booklet, suggestions and borrowed texts. Your help is very valuable.

Compilation : 1 August 2025

Documentation source: `https://github.com/DrCuis/TheArtOfMorph`

Table of Contents

1 Introduction

> The computer is simply an instrument whose music is ideas.
> —*Alan Kay*

Cuis-Smalltalk offers the possibility to easily design your own Morphs – widgets you can interact with and later integrate in your GUI application. There are three ways to design a custom Morph: an aggregate of existing Morphs, a design from scratch or a combination of the two former approaches.

Building a new Morph with an aggregate of existing Morphs is mainly about laying out together Morphs and let the aggregated Morphs manage the low level drawings and input event operations. When there is a need for a custom Morph, this is the path to investigate first; if there is no way to do so, then consider designing from scratch a Morph.

Designing from scratch a Morph requires to deal with its appearance and the handling of the input events; for the former, Cuis offers a vector graphics anti-aliased canvas, the latter is done with a mechanism to filter and to handle mouse and keyboard events occurring in the scope of the custom Morph.

Let's start right away with a design from scratch.

1.1 A first glimpse

It's easy to create custom morphs. Just create a subclass of an existing morph class. Then implement the drawOn: method or add and layout sub morphs.

Let's make an example that draws an ellipse. Making it a subclass of BoxMorph gives it an extent instance variable which specifies its width and height.

```
BoxMorph subclass: #EllipseDemo
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ArtOfMorph'
```

We adjust its default extent[1]:

```
defaultExtent
  ↑ '200@200'
```

In our EllipseDemo, the extent represents the lengths of the ox and oy axis of the ellipse. We use it to draw it accordingly:

[1] Observe the backtricks to improve performance at execution.

```
drawOn: aCanvas
    | radius |
    radius := extent / 2.0.
    aCanvas fillColor: Color purple do: [
      aCanvas ellipseCenter: radius radius: radius]
```

Finally, we instruct Cuis-Smalltalk we want to use the Vector Graphic engine:

```
requiresVectorCanvas
    ↑ true
```

To display an instance of EllipseDemo, open a Workspace and execute EllipseDemo new openInWorld.

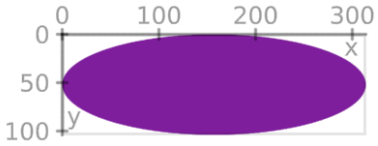

Figure 1.1: Ellipse with axes, resized to an extent approximatively equals to 300@100

The drawing always operates in the own morph coordinates system and we have to ensure our drawing operations remain in the bond defined by the morph origin, in the top-left corner, and its bottom-right corner delimited by its extent attribute, a point.

Before proceeding forward with events, we may want to add semantic to our protocol with #center and #semiAxises messages to use within the drawOn: method:

```
center
    ↑ extent / 2.0
```

```
semiAxises
    " the semi minor and major axis of the ellipse"
    ↑ extent / 2.0
```

```
drawOn: aCanvas
    aCanvas fillColor: Color purple do: [
      aCanvas ellipseCenter: self center radius: self semiAxises ]
```

1.2 Mouse event

Let's explore how custom morphs can react to mouse clicks, mouse hovers, and keystrokes.

1.2.1 Mouse click

Here is a modification of our previous example whose color toggles between red and green each time it is clicked.

As we need a color, we first modify our EllipseDemo to be a subclass of ColoredBoxMorph:

```
ColoredBoxMorph subclass: #EllipseDemo
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ArtOfMorph'
```

Then we initialize it with the red color:

```
initialize
  super initialize.
  color := Color red
```

First, we request we want to handle the mouse click down event:

```
handlesMouseDown: aMouseEvent
  "This enables the morph to handle mouse events such as button presses."
  ↑ true
```

Then at each mouse click, we toggle the color attribute between red and greed:

```
mouseButton1Down: aMouseEvent localPosition: aPosition
  color := (color = ‘Color red‘) ifTrue: [‘Color green‘] ifFalse: [‘Color red‘].
  self redrawNeeded
```

Of course we adjust the drawing method to use the color attribute:

```
drawOn: aCanvas
  aCanvas fillColor: color do: [
    aCanvas ellipseCenter: self center radius: self semiAxises ]
```

To render this, open a Workspace and evaluate EllipseDemo new openInWorld. Click the circle several times to toggle its color.

1.2.2 Mouse hovering

Now let's modify our EllipseDemo to toggle its color based on whether the mouse cursor is hovering over it.

This time, we want to handle event when the mouse pointer is hovering our EllipseMorph:

```
handlesMouseOver: aMouseEvent
    "This enables the morph to handle mouse enter and leave events."
    ↑ true
```

Of course we remove the handlesMouseDown: method, or alternatively we edit it so it return false, to let Cuis-Smalltalk handles this event:

```
handlesMouseDown: aMouseEvent
    "This enables the morph to handle mouse events such as button presses."
    ↑ false
```

There are two event handlers associated when handling mouse over: when entering and when leaving a morph. We edit the methods accordingly to toggle the morph color:

```
mouseEnter: aMouseEvent
    color := ‘Color green‘.
    self redrawNeeded


mouseLeave: aMouseEvent
    color := ‘Color red‘.
    self redrawNeeded
```

Create an instance as seen previously, the hover onto and off of the ellipse to toggle its color.

Observe how the frontier between inside and outside of the ellipse is a rectangle, this is because our EllipseDemo is a kind of BoxMorph optimised for rectangular shape. To have exact pixel detection, including shape drawn with holes, our EllipseDemo would require to be a direct subclass of PlacedMorph. In the process, we will lose the extent and color attributes, and we will have to define ones of our own.

1.2.3 Grow on user request

Now let's combine the mouse hover and mouse click events: at button 1 click, the ellipse shrinks slightly; at button 2 click, it grows greatly.

To do so we introduce a shrink attribute initialized to 0:

```
initialize
    super initialize.
    color := Color red.
    shrink := 0
```

Then it changes depending on user actions, its value increase slightly at button 1 click:

```
mouseButton1Down: aMouseEvent localPosition: aPosition
    shrink := (shrink + 0.5) min: (extent x min: extent y) // 2.
    self redrawNeeded
```

and decrease quickly at button 2 click,

```
mouseButton2Down: aMouseEvent localPosition: aPosition
    shrink := (shrink - 5) max: 0.
    self redrawNeeded
```

Of course we have to bound the shrink attribute between 0 and the smaller extent axis of the whole morph.

Then we adjust our semiAxies method used to draw the ellipse:

```
semiAxises
    ↑ (extent / 2.0) - shrink
```

1.3 Keyboard event

So far, we explored how a morph interacts with the mouse pointer, it may also respond to keyboard events. In this section, we modify our EllipseDemo to adjust its color at keyboard interaction.

First, identically to mouse event, we indicate our morph want to handle keyboard event:

```
handlesKeyboard
    "This enables the morph to handle key events if it has focus."
    ↑ self visible
```

We handle the keyboard event only when our morph is visible.

Keyboard event is associated with the concept of keyboard focus. In the world of morph, one or zero morph own the keyboard focus at a time, it means this morph will receive the keyboard event.

Moreover, in Cuis-Smalltalk there is this preference #focusFollowsMouse. When true, the keyboard focus is automatically changed to the morph the mouse pointer is hovering; when false, the keyboard focus is only changed to a morph at user mouse click on this specific morph.

To know what is the preference in your Cuis-Smalltalk system, execute the code:

```
Preferences at: #focusFollowsMouse
⇒ true
```

I personally prefer to explicitly inform the Cuis-Smalltalk system where the keyboard focus should go. Indeed, my mouse tends to slip on my desk, resulting on the keyboard focus to change annoyingly:

```
Preferences at: #focusFollowsMouse put: false
```

Our EllipseDemo honors this preference when the mouse pointer enter the morph:

```
mouseEnter: aMouseEvent
    color := ‘Color green‘.
    "If the user opted for focus to automatically
    move focus to the morph under the cursor then tell
    the cursor (event hand) to give focus to this morph."
    (Preferences at: #focusFollowsMouse) ifTrue: [aMouseEvent hand newKeyboardFocus: self].
    self redrawNeeded
```

The *hand* is the mouse pointer object in the Cuis-Smalltalk terminology. It manages the keyboard focus and it is informed when the focus should be affected to another morph.

When the mouse pointer leaves our morph we let its parent morph manages the focus:

```
mouseLeave: aMouseEvent
    super mouseLeave: aMouseEvent.
    color := ‘Color red‘.
    self redrawNeeded
```

accordingly to the #focusFollowsMouse system preference:

```
Morph>>mouseLeave: evt
    (Preferences at: #focusFollowsMouse)
      ifTrue: [evt hand releaseKeyboardFocus: self].
    ../..
```

To handle the keyboard strokes, we override the dedicated method keyStroke:. We first ensure the key stroke was not handled by the parent[2] then we do the handling specific to our EllipseDemo morph:

```
keyStroke: aKeyEvent
    | character |
    super keyStroke: aKeyEvent.
    aKeyEvent wasHandled ifTrue: [↑ self].
    character := Character codePoint: aKeyEvent keyValue.
    color := character
      caseOf: {
        [ $r ] -> [ ‘Color red‘ ].
        [ $g ] -> [ ‘Color green‘ ].
        [ $b ] -> [ ‘Color blue‘ ] }
      otherwise: [color].
    self redrawNeeded
```

[2] For example, to manage keyboard shortcut or tabulation.

The event object is interrogated with dedicated messages to detect modifier keys (i.e. #controlKeyPressed). Browse the UserInputEvent class to discover them all.

To have more flexibility on the color used in our ellipse demo, let's implement the following features:

- Pressing *h*, *s* or *v* increase, respectively, the hue, the saturation and the brightness of the ellipse color.
- Pressing *Ctrl-h*, *Ctrl-s* or *Ctrl-v* decrease these same values.

```
keyStroke: aKeyEvent
    | character increment h s v |
    ../..
    h := color hue.
    s := color saturation.
    v := color brightness .
    increment := aKeyEvent controlKeyPressed ifTrue: [-0.1] ifFalse: [0.1].
    character
      caseOf: {
        [ $h ] -> [ h := h + (increment * 13) ].
        [ $s ] -> [ s := s + increment ].
        [ $v ] -> [ v := v + increment ] }
      otherwise: [].
    color setHue: h saturation: s brightness: v.
    self redrawNeeded
```

Our gentle introduction ends here, we have exposed several facets of the Morph system to build from scratch your own morph: drawing of the morph and handling of the mouse and keyboard input. In the following chapters we explore more in detail the design from scratch of your own morphs and how to combine them with existing morph.

2 Design by reuse

In this chapter you will learn how to design new morph – in the idea of a new widget – by assembling existing ones. We first illustrate the topic with existing morphs from the Cuis-Smalltalk system and the `Cuis-Smalltalk-UI` repository. Then we dive-in the design of a new morph – a file selector – gradually improving it from a quick design in a `Workspace` to a class of its own. Finally, its design is bullet proof tested by reusing it in an end-user GUI.

2.1 From where to start?

Design by reuse indeed, but from where to start? Which classes should we make reuse from? As often, the Cuis-Smalltalk system may be our best guide, let's interrogate it to learn which morph has more subclasses.

We collect, for each existing morph in the Cuis-Smalltalk system, the quantity of subclasses, then we sort the result.

```
| hallOfFame |
hallOfFame ← Morph allSubclasses collect: [:each |
   Array with: each with: each subclasses size].
hallOfFame ← hallOfFame sort: [:array1 :array2 | array1 second > array2 second]
```

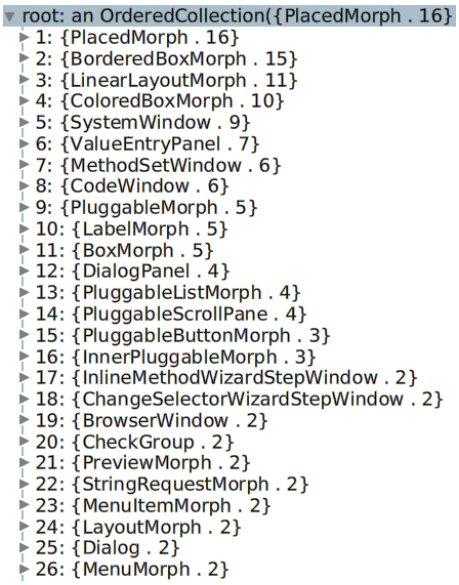

Figure 2.1: The hall of fame of morph subclasses count

Let's analyze some top ranked morphs:

1. PlacedMorph. Its subclasses need to override the drawOn: method. So not a candidate to design morph by reuse.

2. BorderedMorph, ColoredBoxMorph and BoxMorph. Those classes are PlacedMorph with a few additional characteristics. Subclassing these classes will require most of the time overriding the drawOn: method.
3. LinearLayoutMorph. It has 11 subclasses. It is designed to assemble morphs in a new morph, using this class for reuse makes perfectly sense and we already know how to use it.
4. SystemWindow. As a view representing a whole application, subclassing it makes sense to implement specific behaviors of one GUI application, but not as a morph you can reuse as a widget.
5. PluggableMorph. It seems very generic, may be a good candidate. It represents a view of an associated model, however its subclasses need to implement a specific drawOn: method. We may want to use it when designing a morph from scratch.
6. PluggableScrollPane. This morph encapsulates an arbitrary morph – called a *scroller* – in a pane with optional scrollbars when the scroller extent is too large. It is very handy, and in some circumstances it makes sense to subclass it.

All in all, we have two candidates to subclass when conceiving a morph by reuse: LinearLayoutMorph and PluggableScrollPane.

2.2 Layout

This section will be very familliar with the booklet *Design GUI with Morph*, and it is a good idea to read its https://DrCuis.github.io/DesignGUI/Layout-components.html (`Layout components`) chapter before.

Arranging a set of morphs is what does the LinearLayoutMorph class, therefore it makes sense to define a new morph based on layout then to install a set of morphs into. This is exactly what does the LabelGroup class of the `Cuis-Smalltalk-UI` package.

2.2.1 Arrange visually

This subclass of LinearLayoutMorph takes a collection of textual descriptions and morphs to arrange them in two columns of labels and morphs. The idea is to give a label to widgets presented in a view.

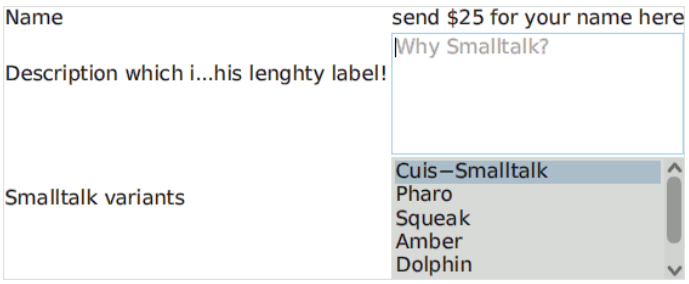

Figure 2.2: A group of three morphs, each with its own label

In one layout column, the label group arranges nicely each label and widget couple in a row so that all the label cells are of same width. What it takes is a collection of label and morph associations. The returned layout is to be added in a higher level view.

```
LabelGroup class>>example1
    ↑ self with: {
      'Name' -> (LabelMorph contents: 'send $25 for your name here').
      'Description which is very long...' -> (
        TextModelMorph withText: '' :: emptyTextDisplayMessage: 'Why Smalltalk?').
      'Smalltalk variants' -> (
        PluggableListMorph
          withModel: (ListModel with: #('Cuis-Smalltalk' 'Pharo' 'Squeak' 'Amber'))
          listGetter: #list
          indexGetter: #listIndex
          indexSetter: #listIndex: ) } ::
      color: Color white paler ;
      yourself
```

The LabelGroup is a passive object, its only purpose is to arrange visually morphs: all the user interactions are managed by the widgets. However, in some circumstance, we want both to arrange widgets and be notified about specific user interactions. This is what does the CheckGroup and RadioGroup classes.

2.2.2 Be notified

Once morphs are arranged in a layout, it makes sense to be notified through events when the user interacts with some of the arranged morphs. Under this perspective, the LabelGroup class is absolutely passive, contrary to the CheckGroup class we will present now.

As a LabelGroup, a CheckGroup presents a collection of labels with associated widgets, here CheckButtonMorph. In a CheckGroup, zero or more CheckButtonMorph can be selected at once. In a RadioGroup, a subclass, only one RadioButtonMorph is selected at once.

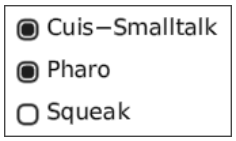

Figure 2.3: A check group to select among the baby squeaks

Creating a check group only requires a collection of labels[1]:

```
CheckGroup class>>example1
    | group |
    group := self fromList: #('Cuis-Smalltalk' 'Pharo' 'Squeak').
    group buttons do: [:each |
      each when: #checkSelection send: #show: to: Transcript].
    ↑ group
```

In the example, the #checkSelection event emitted by each check button is captured for report purpose. The attentive reader will observe this event is not specific to the check

[1] As an alternative to textual label, arbitrary morphs can be used instead.

group. Indeed, the check group itself emits another specific event #informCheckSelection when a button is selected:

```
CheckGroup>>newSelection: radioButton
    " Inform we have a new selection "
    self triggerEvent: #informCheckSelection with: (self symbolForButton: radioButton)
```

The event is triggered with the button label as attribute. To observe its use, experiment with the method CheckGroup class>>example2.

2.3 Scroll pane

Compared to the LinearLayoutMorph class, the PluggableScrollPane class doesn't have much subclasses. We found ones for all sort of list of items or text editor with need to scroll contents.

This class doesn't need to be subclassed to be useful, each time you want to present a morph with a large extent, embed it in a scroll pane:

```
PluggableScrollPane new ::
    scroller: Sample02Bezier new;
    color: Color white;
    openInWorld
```

Example 2.1: Bezier curves on a scroller

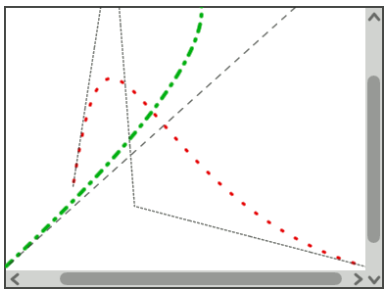

Figure 2.4: A scroll pane encapsulating Bézier curves with scroll bars

Among its few subclasses, FlowLayoutMorph extends the behavior of the layout morph, but doesn't inherit from, to present a collection of morphs in a strip spanning in several rows. It is flanked with a vertical scroll bar, if needed, inherited from the behavior of the PluggableScrollPane.

Its use is simple:

```
FlowLayoutMoprh class>>example1
    | flow cells |
    flow := self new openInWorld.
    cells := OrderedCollection new.
    50 timesRepeat: [ cells add: (
      ColoredBoxMorph new ::
        morphExtent: (5 to: 80) atRandom asPoint;
        color: Color random)].
    flow cells: cells
```

When resizing the morph, particularly its width, the flow of morphs is updated simultaneously.

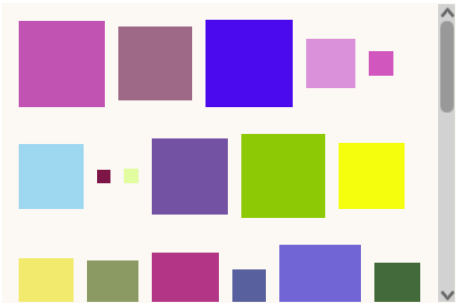

Figure 2.5: Flow of boxes of various sizes

2.4 File Selector

So far we have observed existing morphs. Let's now build our own new morph.

2.4.1 Poor man implementation

First, let's do something quick and fun: a poor man file selector. All it takes is to get the entries of a given directory, collect them as previews and add them all in a flow layout:

```
| directories |
directories := DirectoryEntry userBaseDirectory children collect: [:anEntry |
  FilePreviewMorph
    object: anEntry
   image: ((anEntry isFileEntry ifTrue: [Theme current genericTextIcon] ifFalse: [Theme cur-
rent fetch: #( '16x16' 'places' 'folder' )]) magnifyTo: 64@64)
    buttons: nil
    label: anEntry baseName ::
    borderColor: Color transparent;
    color: Color transparent].
FlowLayoutMorph new ::
  openInWorld;
  cells: directories
```

Of course at this stage, we can't browse in the directories tree:

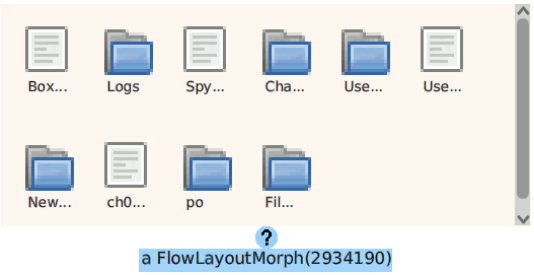

Figure 2.6: A basic tool to list the files in a folder

To add more behavior to our poor man file selector, we want to make our first morph by reusing existing components.

2.4.2 First morph design by reuse

Our morph presents visually files and directories at a given location in the disk of the host. As we want this morph to be reused by other GUI designer, it doesn't present itself in a window but in a simple surface, a pane. Therefore we name it FileSelectorPane, it emits event when the user selects a file. It updates itself with new contents when the user double-clicks on a folder.

Because we may have numerous files and directories to present, we create our FileSelectorPane as a subclass of FlowLayoutMorph:

```
FlowLayoutMorph subclass: #FileSelectorPane
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ArtOfMorph'
```

In its parents hierarchy, our FileSelectorPane has the PluggableMorph ancestor, this one is observing a model. In the present context, the model represents the currently observed DirectoryEntry. It is set by default to the user base directory:

```
initialize
  super initialize.
  self open: DirectoryEntry userBaseDirectory
```

When opening a new location, directories and files are collected and sorted separatly, in two groups, to build meaningful pre-views:

```
open: aDirectoryEntry
    | entryViews |
    model := aDirectoryEntry.
    entryViews := OrderedCollection new.
    model isRoot ifFalse: [ | parentView |
      parentView := self entryPreviewFor: model parent.
      parentView relabel: '..' bold.
      entryViews add: parentView].
    entryViews
      addAll: (self previewsFor: model directories);
      addAll: (self previewsFor: model files ).
    self cells: entryViews
```

The directory and file entries are sorted appropriately and each one is flanked with a preview:

```
previewsFor: entries
    ↑ (entries sort: [:a :b | a baseName asUppercase < b baseName asUppercase ])
        collect: [:anEntry | self entryPreviewFor: anEntry]
```

The special directory "**..**" above is inserted first for the user to browse to the parent directory of the model.

Observe below how each directory preview is listening to the double click event, in that circumstance the related directory is opened.

```
entryPreviewFor: fileEntry
    | fileView |
    fileView := FilePreviewMorph
      object: fileEntry
      image: ((fileEntry isFileEntry
        ifTrue: [Theme current genericTextIcon]
        ifFalse: [Theme current fetch: #( '16x16' 'places' 'folder' )]) magnifyTo: 48@48)
      buttons: nil
      label: fileEntry baseName ::
      borderColor: Color transparent;
      color: Color transparent.
    fileEntry isDirectoryEntry ifTrue: [
      fileView when: #doubleClick send: #open: to: self with: fileEntry].
    ↑ fileView
```

We are relying on FilePreviewMorph, a composition of several morphs, itself emitting events to notify about user activities.

To integrate this widget with other morph, particularly to behave as a file selector, we want it to trigger event when the user selects a file. A FilePreviewMorph emits a #selected event each time the user clicks it, let's capture this event to manage it internally:

```
entryPreviewFor: fileEntry
    ../..
    fileView when: #selected send: #toggleSelection: to: self with: fileView.
    ↑ fileView
```

Then we define the behavior for the #toggleSelection: message:

```
toggleSelection: fileView
    | selectedView |
    selectedView := cells detect: [:aFileView | aFileView isSelected] ifNone: [nil].
    selectedView = fileView
      ifTrue: [ " unselect view, no view selected anymore "
        fileView toggleSelection.
        selectedView := nil]
      ifFalse: [
        selectedView ifNotNil: [selectedView toggleSelection].
        fileView selected: true.
        selectedView := fileView].
    selectedView
      ifNil: [self triggerEvent: #noSelection]
      ifNotNil: [self triggerEvent: #selectedFile with: selectedView fileEntry]
```

To visualise in the `Transcript` window the events propagation, instantiate a new file selector and capture its events:

```
| selector |
selector := FileSelectorPane new openInWorld.
selector when: #noSelection send: #print to: 'no entry'.
selector when: #selectedFile send: #show: to: Transcript
```

To strop listening to events, just send the #removeAllActions message to the listener: selector removeAllActions.

Morph are both listener and emitter of events. Doing so is important for decoupling the objects between each others and to improve objects reuse.

We designed the FileSelectorPane to be itself reusable in other morph. In the next section we illustrate its use to improve the usability of the StringRequestMoprh, a morph used – among other things – to ask the user to key-in a file name; we want to improve to make it more user friendly.

2.4.3 Beyond string request morph

As illustrated by the sketch below, the idea is to have a file pane that expands on user request, the user then selects a file or a directory directly by pointing it on the pane.

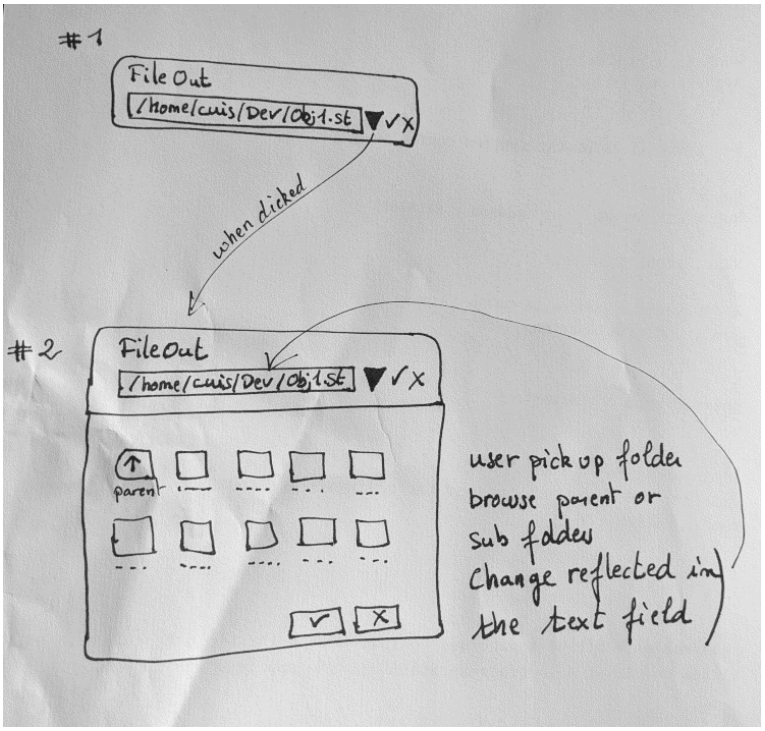

Figure 2.7: A sketch depicting a file selector and its behavior

Let's extend the behavior of StringRequestMoprh:

```
StringRequestMorph subclass: #FileRequestMorph
  instanceVariableNames: 'fileBrowser'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ArtOfMorph'
```

We add an the instance variable fileBrowser to hold our file selector when it is unfolded. We override a few methods from StringRequestMorph. We add a button to unfold/fold the file browser:

```
addTextPane
  super addTextPane.
  submorphs first addMorph:
    ((PluggableButtonMorph model: self action: #toggleFileBrowser)
      setBalloonText: 'Browse the file system to select a file or directory';
      icon: (Theme current fetch: #( '16x16' 'places' 'folder' )) )
```

When unfolding the file browser, we check its current status if absent we instantiate a new one just below the text entry, in the alternative we just delete it from screen:

```
toggleFileBrowser
    fileBrowser ifNotNil: [
      fileBrowser delete.
      fileBrowser := nil.
      ^ self].
    fileBrowser := FileSelectorPane new open: self directoryEntry.
    fileBrowser when: #selectedFile send: #entry: to: self.
    fileBrowser
      color: (Color white alpha: 0.8);
      morphPosition: self morphPosition + (0  (self morphHeight + 5));
      morphExtent: self morphExtent * (1@5);
      openInWorld
```

In this method two important points: the directoryEntry method answers the current directory as edited by the user and the #selectedFile event emitted by the FileSelectorPane is observed and dispatched to the entry: method. Each time the user selects a file, the message #entry: is sent to the FileRequestMorph instance:

```
entry: anEntry
    textMorph hasUnacceptedEdits: false.
    response := anEntry asString.
    self changed: #response.
    textMorph hasUnacceptedEdits: true
```

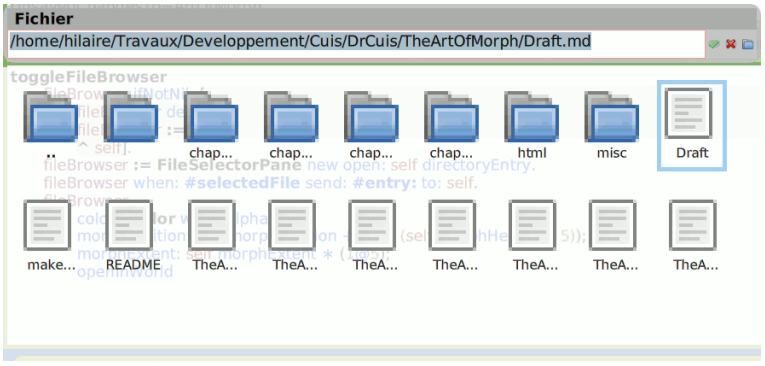

Figure 2.8: File request morph and its file browser unfolded

We end here our chapter on designing morph by reusing existing ones. From the usability perspective, our FileRequestMorph example is far to be complete; user interface grows on small details. There is a lot of room for improvements: the printed filenames below the icons are ridiculously short, a distinction between file and folder selection when the tool is invoked will allow a more precise usability behavior, the button to unfold & fold the file browser could have a selected state. These improvements are good exercises to strengthen your skill.

3 Design from scratch

When you imagine an unconventional way to interact with the computer, it may indicate you are on the verge to design new morphs from scratch: a new way to present information or/and to interact with. This chapter will help you in that circumstances.

As we did in the previous chapter, we take a look at existing morphs to learn from them, then we create our own morph.

3.1 A bit of introspection

Morphs implemented from scratch come with a drawOn: method to produce its visual representation. We already met this method in the Chapter 1 [Introduction], page 1. Let's interrogate our running Cuis-Smalltalk system to analyze which morphs are involved:

```
Morph allSubclasses size.
⇒ 155.

(Morph allSubclasses select: [:each | each selectors includes: #drawOn:]) size.
⇒ 69.
```

Example 3.1: How many morphs implement drawing operations?

We observe the majority of morphs are implemented by composing existing morphs, this was the topic of the previous chapter. We want to interrogate the Cuis-Smalltalk system a bit further by considering the line count of the drawOn: method of each morph.

```
| morphs |
morphs := Morph allSubclasses select: [:each | each selectors includes: #drawOn:].
morphs sort: [:classA :classB |
  (classA sourceCodeAt: #drawOn:) lineCount > (classB sourceCodeAt: #drawOn:) lineCount]
```

Example 3.2: Line count of each drawOn:

This script should be executed to open on an object explorer, *Alt-Shift-I*.

Sadly, this result is not representative of the complexity of the drawing operations. Indeed, well written Smalltalk code tends to span over small & well factored methods. Nevertheless, the object explorer let us quickly jump from one method to another.

To improve the exploring experience, we can go a bit further in our previous script to present both the classes and the source codes of the drawOn: methods:

```
| morphs |
morphs := Morph allSubclasses select: [:each | each selectors includes: #drawOn:].
morphs := morphs sort: [:classA :classB |
  (classA sourceCodeAt: #drawOn:) lineCount > (classB sourceCodeAt: #drawOn:) lineCount].
morphs := morphs collect: [:each | each -> (each sourceCodeAt: #drawOn:)].
morphs explore.
```

Example 3.3: Tooling to review each drawOn:

Once the code is executed, we have an object explorer to review methods. The items at the top of the object explorer are with longer drawOn: method:

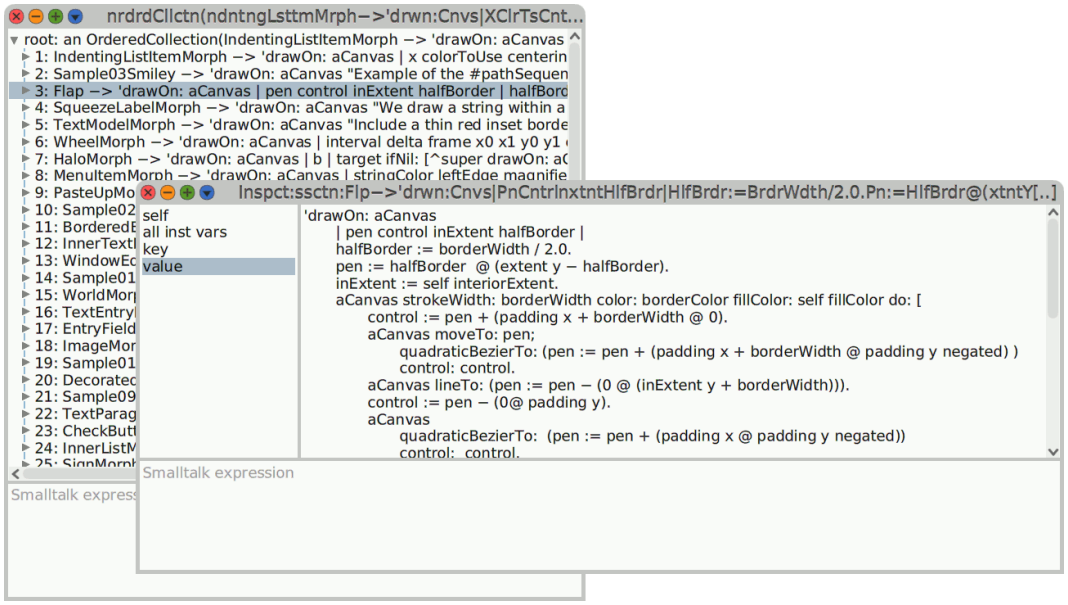

Figure 3.1: Explorer of the draw methods

In this list, several items are Sample*** classes, they are examples found in the `Morphic-Examples` class category. They demonstrate the capability of the `VectorGraphics` engine.

3.2 Red to Medic cross

Let's take a look at Sample01Cross and its unique drawOn: method:

```
drawOn: aCanvas
    aCanvas strokeWidth: 8 color: Color lightOrange fillColor: Color red do: [
      aCanvas polyLine: { 100@0 . 140@0 . 140@240 . 100@240 . 100@0 } ].

    aCanvas strokeWidth: 8 color: Color lightOrange fillColor: Color red do: [
      aCanvas polyLine: { 0@100 . 0@140 . 240@140 . 240@100 . 0@100 } ]
```

Figure 3.2: Sample01Cross shape with two rectangles

Edit the drawOn: method to render a cross as shown in the image below (Figure 3.3).

Figure 3.3: A better looking cross

Exercise 3.1: Better looking shape

The method names involved in the code above will look familiar to users acquainted to SVG (Scalable Vector Graphic)[1]. It is indeed inspired by its specifications, and it may serve as a loose reference[2]. Nevertheless, your first reference to look at regarding the available drawing operations is the AbstractVectorCanvas class then some bits of its parent class, MorphicCanvas.

In this class, the methods are divided in two important categories: the helper methods of the `drawing -` categories and the fundamental Vector Graphics protocol in the `paths` & `strokes` categories.

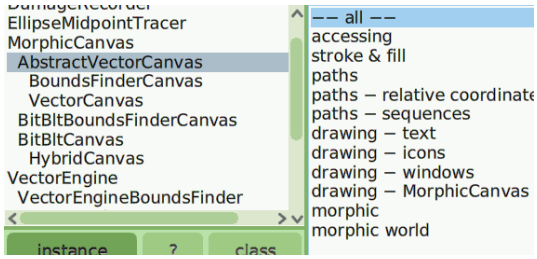

Figure 3.4: Method categories of the AbstractVectorCanvas

In its `drawing - MorphicCanvas` category observe how its helper method to draw a straight line relates to Vector Graphics engine:

[1] `https://en.wikipedia.org/wiki/SVG`

[2] `https://www.w3.org/TR/SVG11/`

```
line: pt1 to: pt2 width: morphStrokeWidth color: aStrokeColor
    self strokeWidth: morphStrokeWidth color: aStrokeColor do: [
      self moveTo: pt1.
      self lineTo: pt2 ]
```

In the strokeWidth:color:do: method above, the argument of the do: keyword is the path followed by the sketch, it describes the outline of the shape to construct. It blends very nicely in the Smalltalk syntax as a bloc of code. Your drawing operations should always be presented in such a bloc. Of course, it can be factored in several smaller methods or iterative processes, making it a very powerful mean for drawing.

The `stroke & fill` method category shows the methods to adjust the style of the path stroke. All in all, there are four categories of parameters to play with:

- **width.** A float number representing the width of the line.
- **color.** The color of the stroke, it's a Cuis-Smalltalk Color instance.
- **foll color.** The filling color, a Color instance.
- **dash style.** There are three different parameters to play with to represent the style of the line.

To be able to use floating numbers instead of integers, help to obtain smoother visual rendering. To illustrate it, let's turn our simple cross as a flashing medical cross as seen above the drugstore.

Let's rename the class and add a width attribute:

```
PlacedMorph subclass: #MedicCross
    instanceVariableNames: 'width'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'ArtOfMorph'
```

We initialize it properly:

```
initialize
    super initialize.
    width := 0
```

To pulse the cross, we use the morphic's step mechanism, generally involving 3 methods: in our MedicCross the step method is continuously executed at a period in milliseconds indicated by its stepTime method,

```
CrossMedic >> stepTime
    ↑ 10
```

then, to indicate we want the stepping to occur, its wantsSteps method must return true,

```
CrossMedic >> wantsSteps
    ↑ true
```

Implement the step method in MedicCross so that its width increments by 0.2 until 30, then it is zeroed.

Exercise 3.2: step method makes it pulse

Figure 3.5: A flashing medic cross

In the next section we go a bit further by representing a real world object, a simple ruler.

3.3 Ruler

Rulers come with different length, our morph ruler is a PlacedMorph[3] with a length attribute:

```
PlacedMorph subclass: #Ruler
    instanceVariableNames: 'length'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'ArtOfMorph'
```

Our ruler is to be graduated with metric unit, so we define our scale between pixels and centimeters:

```
ppcm
    " pixels per cm "
    ^ 50.0
```

To both ease reading of the graduations and to reveal as much as possible the visuals underneath the ruler, it is painted in plain color under the sticks and with transparency elsewhere.

[3] It can be grabbed and moved arround.

```
drawOn: canvas
    | font extent |
    font := FontFamily familyName: FontFamily defaultFamilyName pointSize: 8.
    extent := length * self ppcm + (self ppcm / 2) @ 60.

    canvas fillRectangle: (-5 @ 0 corner: extent x @ 25) color: Color yellow.
    canvas fillRectangle: (-5 @ 25 corner: extent) color: (Color yellow alpha: 0.5).
    canvas frameRectangle: (-5 @ 0 corner: extent) borderWidth: 0.5
      color: Color yellow muchDarker .

    canvas strokeWidth: 0.8 color: Color black do: [
      0 to: length do: [:posX |
        canvas moveTo: posX * self ppcm  0.5; lineToY: 10.
        canvas moveTo: posX * self ppcm + (self ppcm / 2) @ 0.5 ; lineToY: 6] ].

    0 to: length do: [:posX |
      canvas drawString: posX asString
        atCenterX: posX * self ppcm @ 12 font: font color: Color black]
```

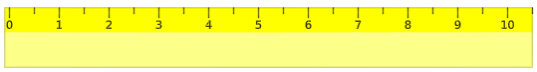

Figure 3.6: Ruler with a centimeter graduation

When drawing, the coordinates we are using are always considered as part of a coordinates system specific to the morph we are drawing into. This makes the task both simple and powerful, we will be able to benefit from this feature to implement smart user interactions. In the method above, the zero on the ruler coincide with the origin in the morph coordinates system.

The drawOn: method above is written to be compact and easy to read; however the method ppcm is used extensively to convert between centimeters and pixels, it implies numerous multiplications and some divisions. These operations are more expensive than additions. Rewrite drawOn: to avoid multiplication and division, particularly in loops.

Exercise 3.3: Avoid expensive calculus

Our ruler should be graduated each millimeter as shown in the figure below – Figure 3.7. Extend the drawOn: to do so.

Exercise 3.4: Millimeter graduation

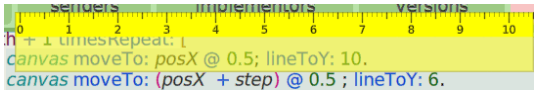

Figure 3.7: Ruler with a millimeter graduation

Very likely, you already noticed the ruler can be dragged around. When activating its halo of handles – ***Alt-clic*** or ***middle-clic*** – it rotates with the blue handle at its left bottom position.

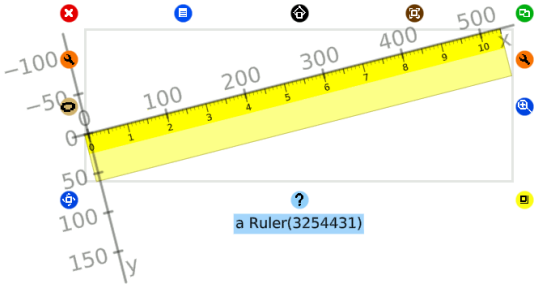

Figure 3.8: Rotating the ruler with its handle

As you experience it, it is rotated around the center of its bounding box. From the ruler inspector, interrogate the ruler about its center:

```
self rotationCenter
⇒ 260.9997510912408 @ 29.999971389797793
```

These coordinates are relative – as always in the Cuis-Smalltalk system – to the own ruler coordinates system. When pivoting the ruler, it will be more practical it rotates around its origin, this will ease doing measure on the screen. Hopefully it is easy to adjust it:

```
Ruler >> rotationCenter
    ↑ '0@0'
```

It now rotates around this point!

Using the halo handle to rotate the ruler is not very practical, we need a direct handle on the ruler to do so. In the next section, you will learn how to compose our morph designed from scratch with additional morphs.

3.4 Composing

What we want is a direct handle on the ruler to rotate it. It is just a morph button to be inserted somewhere in the ruler, that's all we need to do.

```
insertButtons
    | btn |
    btn := ButtonMorph model: self action: #rotateRuler ::
      actWhen: #buttonStillDown;
      icon: Theme current refreshIcon;
      color: Color transparent;
      selectedColor: Color yellow darker;
      morphExtent: Theme current refreshIcon extent * 1.5.
    self addMorph: btn.
    btn morphPosition: length * self ppcm @ 30
```

The button morph added to the ruler is positioned according to the coordinates system of the ruler – its *owner*. Whenever it is scaled or rotated later with its halo handles, its sub-morphs – here the buttons – are positioned accordingly. We decide to place the button at the end of the ruler, it is more practical when pivoting the ruler. The button acts when the user keeps pressing the mouse button down, the method rotateRuler is then called.

In this method we calculate the angle between two consecutive vectors going from the ruler rotation center to the mouse positions. As the mouse positions are in world coordinates, we ask for the ruler rotation center – conceitedly also its origin – to be converted in the world coordinates system – self externalizeToWorld: self rotationCenter.

As we want both the angle and the direction of the user gesture – upward or downward – we want a signed angle. We use the vector product[4] to deduce the angle to rotate the ruler accordingly.

```
rotateRuler
    | event p1 v1 v2|
    "any thing new to do?"
    event := self activeHand lastMouseEvent.
    event isMove
      ifTrue: [
        p1 := self externalizeToWorld: self rotationCenter.
        v1 := lastHandPosition - p1.
        lastHandPosition := event eventPosition.
        v2 := lastHandPosition - p1.
        (v1 isZero or: [v2 isZero]) ifTrue: [↑ self].
        self rotateBy:  ((v1 crossProduct: v2) / (v1 r * v2 r)) arcSin ]
      ifFalse: [lastHandPosition := event eventPosition]
```

[4] https://en.wikipedia.org/wiki/Cross_product#Definition

You may want to take a second look at this method, it is a bit complex at first, but it also expose the wonderful design of Cuis-Smalltalk's Morph 3 to manage this kind of user interaction.

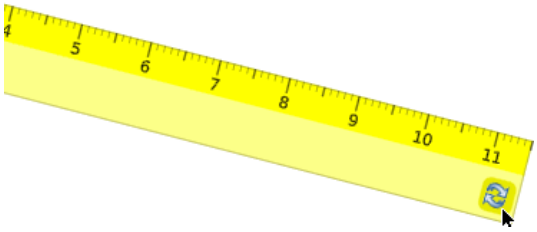

Figure 3.9: A handy button to rotate the ruler

Our next move is to give the ability to the user to change the length of her ruler, the length attribute. Therefore, we insert a second button to do so and in the process refactor the method:

```
insertButtons
    | btn buttonExtent |
    buttonExtent := Theme current refreshIcon extent * 1.5.
    btn := ButtonMorph model: self action: #rotateRuler ::
      actWhen: #buttonStillDown;
      icon: Theme current refreshIcon;
      color: Color transparent;
      selectedColor: Color yellow darker;
      morphExtent: buttonExtent.
    self addMorph: btn.
    btn := ButtonMorph model: self action: #resizeRuler ::
      actWhen: #buttonStillDown;
      icon: (Theme current fetch: #( '16x16' 'actions' 'go-last' ));
      color: Color transparent;
      selectedColor: Color yellow darker;
      morphExtent: buttonExtent.
    self addMorph: btn.
    self positioningButtons
```

As the length will vary, so do the ruler morph width, it makes perfect sense to have a distinct method to put the buttons at the right place:

```
positioningButtons
    | buttonWidth position |
    buttonWidth := submorphs first morphWidth.
    position := length rounded * self ppcm -4 @ 30.
    submorphs do: [:btn |
      btn morphPosition: position.
      position :=  position translatedBy: - 4 - buttonWidth @ 0 ]
```

Now the resizeRuler method to effectively change the length of the ruler looks a bit similar to rotateRuler in its general shape. But there are subtule differences to benefit from the Morph 3 design.

```
resizeRuler
    | event prev |
    "any thing new to do?"
    event := self activeHand lastMouseEvent.
    event isMove
      ifTrue: [
        prev := lastHandPosition.
        lastHandPosition := self internalizeFromWorld: event eventPosition.
        self length: length + (lastHandPosition x - prev x / self ppcm)]
      ifFalse: [lastHandPosition := self internalizeFromWorld: event eventPosition]
```

When the user keeps pressing the resize button, the ruler should shrink when the mouse pointer moves in direction of the smallest value of the ruler x-axis and its length should increase when the mouse pointer moves in direction of the greatest value of its x-axis.

To be able to determine these behaviors of the mouse pointer, the coordinates involved must be converted in the ruler coordinates system. This is exactly what is done by internalizing the mouse position self internalizeFromWorld: event eventPosition. Then the abscissa delta between the two last mouse positions is calculated to determine the change to the ruler length. And because we are doing this calculus in the ruler coordinates system, it works whatever its pivoted state.

Figure 3.10: A handy button to resize the ruler

Of course, when the length is adjusted, the positions of the buttons is recomputed as the whole ruler:

```
length: newLength
    length := newLength max: 1.
    self positioningButtons.
    self redrawNeeded
```

Because the drawOn: expected an integer value for the length, its value is rounded before doing the drawing:

```
drawOn: canvas
    | font grad posX extent step roundedLength |
    font := FontFamily familyName: FontFamily defaultFamilyName pointSize: 8.
    roundedLength := length rounded.
    extent := roundedLength * self ppcm + (self ppcm / 2) @ 60.
    ../..
```

Appendix A Documents Copyright

Cuis-Smalltalk mascot

The southern mountain cavy (Microcavia australis) is a species of South American rodent in the family Caviidae.

Appendix B The Exercises

Appendix C Solutions to the Exercises

Design from scratch

Exercise 3.1

To avoid the unpleasant overlapping of the two rectangles, all the drawing operations are to be conducted in an unique stroke. The polyline summit is then extended to follow the outline of the cross.

```
drawOn: aCanvas
    aCanvas strokeWidth: 8 color: Color lightOrange fillColor: Color red do: [
      aCanvas polyLine: {
        100@0 . 140@0 . 140@100 . 240@100 . 240@140 .140@140 .
        140@240 . 100@240 . 100@140 . 0@140 . 0@100 . 100@100 .
        100@0 } ]
```

Exercise 3.2

To enjoy the update, the #redrawNeeded is to be sent to self.

```
step
    width := width + 0.2.
    width > 30 ifTrue: [width := 0].
    self redrawNeeded
```

Exercise 3.3

The principle is to handle *manually* the indexes and to use the timesRepeat: method on integer. It reflects to how it will be coded with lower level languages.

```
drawOn: canvas
    | font grad posX extent step |
    ../..
    step := self ppcm / 2. " half centimeter step "
    canvas strokeWidth: 0.8 color: Color black do: [
      posX := 0.
      length + 1 timesRepeat: [
        canvas moveTo: posX @ 0.5; lineToY: 10.
        canvas moveTo: (posX + step) @ 0.5 ; lineToY: 6.
        posX := posX + self ppcm] ].

    grad := posX := 0.
    length + 1 timesRepeat: [
      canvas drawString: grad asString
        atCenterX: posX @12 font: font color: Color black.
      grad := grad + 1.
      posX := posX + self ppcm]
```

Exercise 3.4

We add the following code at the end of the method and make the stick thinner to 0.3 pixel.

```
drawOn: canvas
    step := self ppcm / 10. "millimeter step "
    canvas strokeWidth: 0.3 color: Color black do: [
      posX := step.
      length * 2 + 1 timesRepeat: [
        4 timesRepeat: [
          canvas moveTo: posX  0.2; lineToY: 4.
          posX := posX + step].
        posX := posX + step] ]
```

Appendix D The Examples

Appendix E The Figures

Appendix F Art of Morph package

Download Art of Morph package (https://github.com/DrCuis/TheArtOfMorph/blob/main/misc/ArtOfMorph.pck.st)

```
'From Cuis7.5 [latest update: #7387] on 1 August 2025 at 12:00:33 am'!
'Description '!
!provides: 'ArtOfMorph' 1 12!
!requires: 'UI-Widgets' 1 54 nil!
!requires: 'UI-Panel' 1 130 nil!
SystemOrganization addCategory: #ArtOfMorph!


!classDefinition: #MedicCross category: #ArtOfMorph!
PlacedMorph subclass: #MedicCross
   instanceVariableNames: 'width'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'ArtOfMorph'!
!classDefinition: 'MedicCross class' category: #ArtOfMorph!
MedicCross class
   instanceVariableNames: ''!

!classDefinition: #Ruler category: #ArtOfMorph!
PlacedMorph subclass: #Ruler
   instanceVariableNames: 'length lastHandPosition font'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'ArtOfMorph'!
!classDefinition: 'Ruler class' category: #ArtOfMorph!
Ruler class
   instanceVariableNames: ''!

!classDefinition: #EllipseDemo category: #ArtOfMorph!
ColoredBoxMorph subclass: #EllipseDemo
   instanceVariableNames: 'shrink'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'ArtOfMorph'!
!classDefinition: 'EllipseDemo class' category: #ArtOfMorph!
EllipseDemo class
   instanceVariableNames: ''!

!classDefinition: #FileRequestMorph category: #ArtOfMorph!
StringRequestMorph subclass: #FileRequestMorph
   instanceVariableNames: 'fileBrowser'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'ArtOfMorph'!
!classDefinition: 'FileRequestMorph class' category: #ArtOfMorph!
FileRequestMorph class
   instanceVariableNames: ''!

!classDefinition: #FileSelectorPane category: #ArtOfMorph!
FlowLayoutMorph subclass: #FileSelectorPane
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'ArtOfMorph'!
!classDefinition: 'FileSelectorPane class' category: #ArtOfMorph!
FileSelectorPane class
   instanceVariableNames: ''!
```

```
!MedicCross commentStamp: 'hlsf 7/29/2025 15:53:25' prior: 0!
A flashing medical cross

   MedicCross new openInHand

!

!MedicCross methodsFor: 'initialization' stamp: 'hlsf 7/29/2025 15:52:21'!
initialize
   super initialize.
   width := 0! !

!MedicCross methodsFor: 'drawing' stamp: 'hlsf 7/29/2025 15:53:09'!
drawOn: aCanvas
   aCanvas strokeWidth: width color: Color green lighter fillColor: Color green do: [
      aCanvas polyLine: {
         100@0 .   140@0 .   140@100 . 240@100 . 240@140 .140@140 .
         140@240 . 100@240 . 100@140 . 0@140 .    0@100 . 100@100.
         100@0 } ].! !

!MedicCross methodsFor: 'stepping' stamp: 'hlsf 7/29/2025 22:42:13'!
step
   width := width + 0.2.
   width > 30 ifTrue: [width := 0].
   self redrawNeeded.! !

!MedicCross methodsFor: 'stepping' stamp: 'hlsf 7/29/2025 15:46:34'!
stepTime
   ^ 10 ! !

!MedicCross methodsFor: 'stepping' stamp: 'hlsf 7/29/2025 15:52:40'!
wantsSteps
   ^ true ! !

!Ruler methodsFor: 'drawing' stamp: 'hlsf 7/31/2025 23:59:02'!
drawOn: canvas
   | grad posX extent step roundedLength |
   roundedLength := length rounded.
   extent := roundedLength * self ppcm + (self ppcm / 2) @ 60.

   canvas fillRectangle: (-5@0 corner: extent x @ 25) color: Color yellow.
   canvas fillRectangle: (-5@25 corner: extent ) color: (Color yellow alpha: 0.5).
   canvas frameRectangle: (-5@0 corner: extent) borderWidth: 0.5 color: Color yellow muchDarker .

   step :=  self ppcm / 2. " half centimeter step "
   canvas strokeWidth: 0.8 color: Color black do: [
      posX := 0.
      roundedLength + 1 timesRepeat: [
         canvas moveTo: posX @ 0.5; lineToY: 10.
         canvas moveTo: (posX  + step) @ 0.5 ; lineToY: 6.
         posX := posX + self ppcm] ].

   step := self ppcm / 10. "millimeter step "
   canvas strokeWidth: 0.3 color: Color black do: [
      posX := step.
      roundedLength * 2 + 1 timesRepeat: [
         4 timesRepeat: [
            canvas moveTo: posX @ 0.2; lineToY: 4.
            posX := posX + step].
         posX := posX + step] ].

   grad := posX := 0.
   roundedLength + 1 timesRepeat: [
      canvas drawString: grad asString atCenterX: posX @12 font: font color: Color black.
      grad := grad + 1.
```

```
      posX := posX + self ppcm].! !

!Ruler methodsFor: 'accessing' stamp: 'hlsf 7/31/2025 23:53:55'!
length: newLength
   length := newLength max: 1.
   self positioningButtons.
   self redrawNeeded ! !

!Ruler methodsFor: 'accessing' stamp: 'hlsf 7/31/2025 18:34:21'!
ppcm
" pixels per cm "
   ^ 50.0! !

!Ruler methodsFor: 'initialization' stamp: 'hlsf 7/31/2025 23:57:21'!
initialize
   super initialize.
   font := FontFamily familyName: FontFamily defaultFamilyName pointSize: 8.
   length := 10 "cm".
   self insertButtons ! !

!Ruler methodsFor: 'initialization' stamp: 'hlsf 7/31/2025 20:28:28'!
insertButtons
| btn buttonExtent |
   buttonExtent := Theme current refreshIcon extent * 1.5.
   btn := ButtonMorph model: self action: #rotateRuler ::
      actWhen: #buttonStillDown;
      icon: Theme current refreshIcon;
      color: Color transparent;
      selectedColor: Color yellow darker;
      morphExtent: buttonExtent.
   self addMorph: btn.
   btn := ButtonMorph model: self action: #resizeRuler ::
      actWhen: #buttonStillDown;
      icon: (Theme current fetch: #( '16x16' 'actions' 'go-last' ));
      color: Color transparent;
      selectedColor: Color yellow darker;
      morphExtent: buttonExtent.
   self addMorph: btn.
   self positioningButtons
   ! !

!Ruler methodsFor: 'initialization' stamp: 'hlsf 7/31/2025 20:28:28'!
positioningButtons
| buttonWidth position |
   buttonWidth := submorphs first morphWidth.
   position := length rounded * self ppcm -4 @ 30.
   submorphs do: [:btn |
      btn morphPosition: position.
      position :=  position translatedBy:  - 4 - buttonWidth @ 0 ]
      ! !

!Ruler methodsFor: 'geometry' stamp: 'hlsf 7/31/2025 14:36:31'!
rotationCenter
   ^ `0@0`! !

!Ruler methodsFor: 'as yet unclassified' stamp: 'hlsf 7/31/2025 23:29:50'!
resizeRuler
   | event prev |
   "any thing new to do?"
   event := self activeHand lastMouseEvent.
   event isMove
      ifTrue: [
         prev := lastHandPosition.
         lastHandPosition := self internalizeFromWorld: event eventPosition.
         self length: length + (lastHandPosition x - prev x / self ppcm)]
```

```
      ifFalse: [   lastHandPosition := self internalizeFromWorld: event eventPosition].! !

!Ruler methodsFor: 'as yet unclassified' stamp: 'hlsf 7/31/2025 23:29:57'!
rotateRuler
   | event p1 v1 v2|
   "any thing new to do?"
   event := self activeHand lastMouseEvent.
   event isMove
      ifTrue: [
         p1 := self externalizeToWorld: self rotationCenter.
         v1 := lastHandPosition - p1.
         lastHandPosition := event eventPosition.
         v2 := lastHandPosition - p1.
         (v1 isZero or: [v2 isZero]) ifTrue: [^self].
         self rotateBy:  ((v1 crossProduct: v2) / (v1 r * v2 r)) arcSin   ]
      ifFalse: [   lastHandPosition := event eventPosition].! !

!EllipseDemo methodsFor: 'accessing' stamp: 'hlsf 6/14/2025 12:17:51'!
center
   ^  extent / 2.0! !

!EllipseDemo methodsFor: 'accessing' stamp: 'hlsf 6/14/2025 13:32:35'!
semiAxises
" the semi minor and major axis of the ellipse"
   ^  (extent / 2.0) - shrink ! !

!EllipseDemo methodsFor: 'initialization' stamp: 'hlsf 6/11/2025 18:54:04'!
defaultExtent
   ^ 200@200! !

!EllipseDemo methodsFor: 'initialization' stamp: 'hlsf 6/19/2025 23:45:40'!
initialize
   super initialize.
   color := Color yellow.
   shrink := 0.
   'Hover over the circle to change its color and unhover to change it back.' print.
   'Click it with left or right button to shrink or to grow the ellipse.' print.
   'Move mouse over the circle and press r, g, or b to change its color.' print.! !

!EllipseDemo methodsFor: 'drawing' stamp: 'hlsf 6/14/2025 12:56:07'!
drawOn: aCanvas
   aCanvas fillColor: color do: [
       aCanvas ellipseCenter: self center radius: self semiAxises ]! !

!EllipseDemo methodsFor: 'event handling testing' stamp: 'hlsf 6/19/2025 23:48:44'!
handlesKeyboard
"This enables the morph to handle key events if it has focus."
    ^ self visible! !

!EllipseDemo methodsFor: 'event handling testing' stamp: 'hlsf 6/14/2025 13:31:52'!
handlesMouseDown: aMouseEvent
"This enables the morph to handle mouse events such as button presses."
    ^ true! !

!EllipseDemo methodsFor: 'event handling testing' stamp: 'hlsf 6/14/2025 12:53:26'!
handlesMouseOver: aMouseEvent
"This enables the morph to handle mouse enter and leave events."
    ^ true! !

!EllipseDemo methodsFor: 'events' stamp: 'hlsf 6/29/2025 12:10:49'!
keyStroke: aKeyEvent
   | character increment h s v |
   super keyStroke: aKeyEvent.
   aKeyEvent wasHandled ifTrue: [^ self].
   character := Character codePoint: aKeyEvent keyValue.
```

```
   color := character
      caseOf: {
         [ $r ] -> [ `Color red` ].
         [ $g ] -> [ `Color green` ].
         [ $b ] -> [ `Color blue` ] }
      otherwise: [color].

   h := color hue.
   s := color saturation.
   v := color brightness .
   increment := aKeyEvent controlKeyPressed ifTrue: [-0.1] ifFalse: [0.1].
   character
      caseOf:   {
         [ $h ] -> [ h := h + (increment * 13) ].
         [ $s ] -> [ s := s + increment ].
         [ $v ] -> [ v := v + increment ]
      }
      otherwise: [].
   color setHue: h saturation: s brightness: v.
   self redrawNeeded! !

!EllipseDemo methodsFor: 'events' stamp: 'hlsf 6/19/2025 23:52:50'!
mouseButton1Down: aMouseEvent localPosition: aPosition
    shrink := (shrink + 0.5) min: (extent x min: extent y) // 2.
    (Preferences at: #focusFollowsMouse) ifFalse: [aMouseEvent hand newKeyboardFocus: self].
    self redrawNeeded! !

!EllipseDemo methodsFor: 'events' stamp: 'hlsf 6/14/2025 13:33:22'!
mouseButton2Down: aMouseEvent localPosition: aPosition
    shrink := (shrink - 5) max: 0.
    self redrawNeeded! !

!EllipseDemo methodsFor: 'events' stamp: 'hlsf 6/19/2025 23:52:35'!
mouseEnter: aMouseEvent
    color := `Color green`.
    "If the user opted for focus to automatically
     move focus to the morph under the cursor then tell
     the cursor (event hand) to give focus to this morph."
    (Preferences at: #focusFollowsMouse) ifTrue: [aMouseEvent hand newKeyboardFocus: self].
    self redrawNeeded.! !

!EllipseDemo methodsFor: 'events' stamp: 'hlsf 6/28/2025 09:36:28'!
mouseLeave: aMouseEvent
   super mouseLeave: aMouseEvent.
   color := `Color red`.
   self redrawNeeded.! !

!EllipseDemo methodsFor: 'events' stamp: 'hlsf 6/14/2025 13:09:40'!
wantsContour
   ^ true! !

!EllipseDemo methodsFor: 'geometry testing' stamp: 'hlsf 6/11/2025 19:05:04'!
requiresVectorCanvas
   ^ true ! !

!FileRequestMorph methodsFor: 'initialization' stamp: 'hlsf 7/27/2025 14:46:09'!
addTextPane
   super addTextPane.
   submorphs first addMorph:
         ((PluggableButtonMorph model: self action: #toggleFileBrowser)
            setBalloonText: 'Browse the file system to select a file or directory';
            icon: (Theme current fetch: #( '16x16' 'places' 'folder' )) )! !

!FileRequestMorph methodsFor: 'accessing' stamp: 'hlsf 7/27/2025 15:10:38'!
directoryEntry
```

```
   | editedContents |
   editedContents := textMorph scroller contents asPlainString.
   ^ editedContents asFileEntry exists
      ifTrue: [editedContents asFileEntry parent]
      ifFalse: [editedContents asDirectoryEntry exists
         ifTrue: [editedContents asDirectoryEntry ] ifFalse: [editedContents asDirectoryEntry parent] ]! !

!FileRequestMorph methodsFor: 'accessing' stamp: 'hlsf 7/27/2025 14:33:32'!
response: aText
   | answer |
   answer := super response: aText.
   (answer and: [fileBrowser notNil]) ifTrue: [fileBrowser delete].
   ^ answer! !

!FileRequestMorph methodsFor: 'accessing' stamp: 'hlsf 7/27/2025 12:59:21'!
toggleFileBrowser
   fileBrowser ifNotNil: [
      fileBrowser delete.
      fileBrowser := nil.
      ^ self].
   fileBrowser := FileSelectorPane new   open: self directoryEntry.
   fileBrowser when: #selectedFile send: #entry: to: self.
   fileBrowser
      color: (Color white alpha: 0.8);
      morphPosition: self morphPosition + (0 @ (self morphHeight + 5));
      morphExtent: self morphExtent * (1@5);
      openInWorld

   ! !

!FileRequestMorph methodsFor: 'events' stamp: 'hlsf 7/29/2025 10:48:13'!
entry: anEntry
   textMorph hasUnacceptedEdits: false.
   response := anEntry asString.
   self changed: #response.
   textMorph hasUnacceptedEdits: true.! !

!FileRequestMorph methodsFor: 'private' stamp: 'hlsf 7/27/2025 12:50:16'!
cancel
   super cancel.
   fileBrowser ifNotNil: [fileBrowser delete]! !

!FileRequestMorph methodsFor: 'private' stamp: 'jmv 7/28/2025 14:40:16'!
ok
   self delete.
   fileBrowser ifNotNil: [fileBrowser delete]! !

!FileRequestMorph class methodsFor: 'as yet unclassified' stamp: 'hlsf 7/27/2025 18:09:49'!
example
"
   FileRequestMorph example
"
   ^ FileRequestMorph
      request: 'Select a file or directory'
      initialAnswer: DirectoryEntry userBaseDirectory asString
      orCancel: nil.
! !

!FileRequestMorph class methodsFor: 'instance creation' stamp: 'hlsf 7/29/2025 10:54:20'!
request: queryString
   ^ self request: queryString initialAnswer: DirectoryEntry userBaseDirectory asString! !

!FileSelectorPane methodsFor: 'initialization' stamp: 'hlsf 7/19/2025 13:26:16'!
initialize
   super initialize.
```

```
   self open: DirectoryEntry userBaseDirectory
   ! !

!FileSelectorPane methodsFor: 'private' stamp: 'hlsf 7/28/2025 23:53:42'!
entryPreviewFor: fileEntry
   | fileView |
   fileView := FilePreviewMorph
      object: fileEntry
      image: ((fileEntry isFileEntry
         ifTrue: [Theme current genericTextIcon]
         ifFalse: [Theme current fetch: #( '16x16' 'places' 'folder' )]) magnifyTo: 64@64)
      buttons: nil
      label: (fileEntry isRoot ifTrue:['/'] ifFalse: [fileEntry baseName]) ::
      borderColor: Color transparent;
      color: Color transparent.
   fileEntry isDirectoryEntry ifTrue: [
      fileView when: #doubleClick send: #open: to: self with: fileEntry].
   fileView when: #selected send: #toggleSelection: to: self with: fileView.
   ^ fileView ! !

!FileSelectorPane methodsFor: 'private' stamp: 'hlsf 7/19/2025 11:57:59'!
previewsFor: entries
   ^ (entries sort: [:a :b | a baseName asUppercase < b baseName asUppercase ]) collect: [:anEntry |
      self entryPreviewFor: anEntry]! !

!FileSelectorPane methodsFor: 'accessing' stamp: 'hlsf 7/29/2025 10:48:53'!
open: aDirectoryEntry
   | entryViews |
   model := aDirectoryEntry.
   entryViews := OrderedCollection new.
   model isRoot ifFalse: [ | parentView |
      parentView := self entryPreviewFor: model parent.
      parentView relabel: '..' bold.
      entryViews add: parentView].
   entryViews
      addAll: (self previewsFor: model directories);
      addAll: (self previewsFor: model files ).
   self cells: entryViews.
   self updateLayout ! !

!FileSelectorPane methodsFor: 'events' stamp: 'hlsf 7/25/2025 19:22:15'!
toggleSelection: fileView
   | selectedView |
   selectedView := cells detect: [:aFileView | aFileView isSelected] ifNone: [nil].
   selectedView = fileView
      ifTrue: [
         fileView toggleSelection. " unselect view, no view selected anymore "
         selectedView := nil]
      ifFalse: [
         selectedView ifNotNil: [selectedView toggleSelection].
         fileView selected: true.
         selectedView := fileView].
   selectedView
      ifNil: [self triggerEvent: #noSelection]
      ifNotNil: [self triggerEvent: #selectedFile with: selectedView fileEntry]! !
```

Appendix G Conceptual index