Chapter 12 - Developing user interfaces

Overview

The group of classes supporting graphical user interfaces (GUIs) is one of the most complex parts of VisualWorks and its detailed coverage is beyond the scope of this book. Fortunately, to be able to use the tools for routine applications and to create interesting extensions of the widget library, you only need to understand a limited number of concepts, and specialized books are available for more advanced uses.

The first requirement for creating a user interface is to be able to draw objects such as lines, widgets, and text on the screen. VisualWorks perspective is that drawing requires a surface to draw on (such as a computer screen or printer paper), the graphical objects being drawn, and an object that does the drawing and holds the parameters that describe the drawing context (such as fonts, colors, and line widths).

Being able to draw an object on the screen is, however, only a part of a typical GUI. Unless the displayed windows are completely passive, the user must also interact with them via the mouse and the keyboard. A user interface must also ensure that the display reflects changes of the displayed data model, and that damage caused, for example, by resizing a window is automatically repaired.

To implement automatic response to model changes and user interaction, Smalltalk uses the model-view-controller paradigm - MVC for short. The model part of the MVC triad holds data and provides notification when the data changes. The view is responsible for drawing the component on the screen, response to model changes, and damage repair. The controller handles user input.

To maintain the UI painting paradigm supported by the UI Painter, we also need a way to integrate a new visual component into a canvas. This is achieved by a widget called the subview which implements access to its view part and the connection to its window container.

12.1 Principles of user interfaces - display surfaces, graphics contexts, and visual parts

This section is dedicated to the principles of VisualWorks user interfaces and provides background for customization of windows and their main components. In routine applications, no programming involving these components is necessary but understanding of these concepts is useful.

Displaying graphical objects on the screen is a complex task and different languages and even different dialogs of Smalltalk approach it differently. The view of VisualWorks is that drawing requires a drawing surface, objects to be displayed, and an object that can do the drawing and holds parameters describing how and where the drawing takes place. We will now deal with these concepts one by one and illustrate them on simple examples. A larger example follows in the next section.

Display surfaces

Display surfaces are the media on which all visuals, including geometric objects, images, widgets, and text, are displayed. The hierarchy of display media is large and its skeleton is as follows:

Object
- **GraphicsMedium**
  - **DisplaySurface**
    - UnmappableSurface
      - **Mask**
      - **Pixmap**
    - **Window**
      - **ScheduledWindow**
        - **ApplicationWindow**
      - TransientWindow
  - HostPrintJob
  - MockMedium
  - PostScriptFile

The abstract class GraphicsMedium at the top of the hierarchy defines the protocol for accessing information such as display defaults (paint, font, and look preferences which include background and foreground color), size of the display surface, number of bits used to represent the color of pixels, and graphics context (an object responsible for display of geometric objects, images, and text). GraphicsMedium Al the supporting instance variables are defined in subclasses.

Subclasses of GraphicsMedium can be divided into those displaying on the screen, those used to construct displayable surfaces in memory, and those dedicated to printing. We will deal only with the first two categories and leave printing to you as an exercise. The group of classes responsible for screen display and display construction are headed by DisplaySurface whose comment essentially says

Class DisplaySurface is the abstract class for all classes that act as destinations for graphics operations targeted for a bitmap video display (as opposed to printing, for example).

Instance variables:

handle	<GraphicsHandle | nil>	handle to the host resource, an interface to the operating system which performs the low level display functions
width	<SmallInteger>	width of the surface in pixels
height	<SmallInteger>	height of the surface in pixels
background	<Paint>	paint used to clear the surface

Even though DisplaySurface is abstract, it provides a lot of useful functionality such as closing the display surface (for example, a window) and access to any part of the display area. Subclasses of DisplaySurface can be subdivided into classes whose instances correspond to windows displayed on the screen, and classes that instantiate display surfaces in memory but don't display them.

Windows

Class Window at the top of the window hierarchy is a concrete class with much functionality but rather useless by itself because its instances have neither a controller (thus allowing only minimal user interaction) nor a component (thus incapable of containing widgets). A lot of the behavior inherited from Window is, however, essential. This includes the knowledge of the window's origin on the screen, its label, its icon, and its sensor which holds information about mouse and keyboard events that occur within the window's extent. Window also defines methods for opening a window (redefined in subclasses), collapsing it into an icon and re-expanding it, moving and resizing a window, and raising it to the top to become the active window on the screen. It also knows the currently active window (class message currentWindow).

As an example of window manipulation, execute the following code fragment to see some of the Window functionality:

```
"Open window, collapse it, expand it, and close it."
| window |
window := Window openNewIn: (100@100 corner: 200@200).
(Delay forSeconds: 3) wait.
window label: 'Test'.
(Delay forSeconds: 3) wait.
window collapse.
(Delay forSeconds: 3) wait.
window expand.
(Delay forSeconds: 3) wait.
window close      "Without this statement, you will not be able to close the window."
```

Class Window has very little direct use and we will thus proceed to its subclass ScheduledWindow. This class adds several behaviors and instance variables, the most important being controller (handles user interaction), component (makes it possible to include a visual part or a collection of visual parts in the window), and model (allowing another object to control a window). The controller of ScheduledWindow is an instance of StandardSystemController which provides the <window> menu and methods for closing,

resizing, moving, relabeling, and other window operations. The window controller is also the means by which Smalltalk keeps track of all its windows and the controllers of all open Smalltalk windows are kept in ScheduledControllers, an instance of ControlManager. The following is a simple example showing how to open a ScheduledWindow:

```
"Open window with label but with no component, and let user close it."
| window |
window := ScheduledWindow new.
window  insideColor: ColorValue blue;
        label: 'Test'.
window openIn: (100@100 corner: 200@200)
```

The user can close a ScheduledWindow in the usual way which is not true for a Window. The window in this example does not have any components so it's not of any interest but we will do an example with a component later.

Until recent VisualWorks releases, ScheduledWindow was the basis of all user interfaces - there was no UI Painter and all UIs had to be created programmatically using ScheduledWindow. With the introduction of the UI Painter, this has changed and applications based on the application model use the ApplicationWindow subclass of ScheduledWindow. ScheduledWindow has thus lost its importance except that it defines many behaviors inherited by ApplicationWindow. In the current class library, most references to ScheduledWindow are in examples because (as our example shows) it is easier to create a very simple demo window programatically than to paint one using the UI Painter.

From within an application, the ApplicationWindow of the interface can be accessed by sending message window to the application builder. As an example, the application model could send

```
self builder window label: 'New label'
```

to change the window's label at run time. The essential part of the comment of ApplicationWindow is as follows:

ApplicationWindow adds functionality required by the User Interface building classes. It also provides for tighter coordination with a corresponding ApplicationModel. In particular, an ApplicationWindow can notify its application of window events (e.g. closing or collapsing), so that other dependent windows can follow suit. Recognized events are #expand, #collapse, #bounds, #enter, #exit, #close, #hibernate, #reopen, and #release.

Instance Variables:

keyboardProcessor	<KeyboardProcessor>	Manages keyboard input
application	<Object | nil>	The application, typically an ApplicationModel
receiveWindowEvents	<Array | nil>	Window events the receiver should respond to
damageRepairIsLazy	<Boolean> Used to control when damage coming up from below is repaired.	

Variable damageRepairIsLazy is used to determine whether a damaged window should be repaired (redrawn) immediately or whether redisplay can wait until a suitable later time. We will talk about damage repair later.

Typical messages sent by the application model to an application include changing its label or background color at run time (messages label: and background:), moving it to a new position (message moveTo:) obtaining its bounding rectangle (displayBox), closing the window (close - usually achieved by sending closeRequest to the application model), collapsing it to an icon and expanding the icon back to a window (collapse and expand), and hiding the window without destroying its internal representation (message unmap) and restoring it without having to reconstruct it (message map). In many cases, a window changed at run time must be redisplayed by message display.

When an application model needs to access more than one window, create a new builder for each of the secondary windows, specify its source as the application model, specify the window's application as the application model, and ask the secondary builder to open the interface. You can also make the secondary window a slave of the master window (message slave). The result is that the slave will mimic closing, collapsing, and expanding of the master window. The slave window may also be made sensitive to

other master window events such as cursor entry or exit. Finally, windows can also be partners by sending bePartner to the window. Partner windows are tied together in the same way as master and slave windows.

We will see examples of some of this functionality in the following sections and other examples are provided in the cookbook.

Main lessons learned:

- In the perspective of VisualWorks, display requires a display surface, objects to be displayed, and a drawing engine (a graphics context).
- Display surfaces are subclasses of GraphicsMedium which has two subtrees, one for display on screens and one for printing on paper.
- Display on screen uses windows and the window hierarchy includes classes Window, ScheduledWindow, and ApplicationWindow.
- Windows created with the UI Painter and used by application models are ApplicationWindows. They inherit most of their functionality from ApplicationWindow superclasses.
- ApplicationWindow has numerous instance variables holding parameters such as bounding box, clipping rectangle, graphics context, background paint, and others.
- Pixmap and Mask are display surfaces used to construct displays before they are mapped to the screen.

12.2 An example of the use of windows – a Virtual Desktop

One annoying aspect of graphical user interfaces is that computer screens are always too small to display all the windows that one would like to see at a time. A partial solution to this problem is a 'virtual desk' program which allows the user to place individual open windows on 'desks' and view only one desk at a time. We will design and implement a very simple version of such a program. We will call it Desk Manager and implement it via class SmallDesk.

Our program will make the screen behave as a peep hole on a much larger desktop divided into four screen-size rectangular areas. The selection of the desired desk is via the user interface in Figure 12.1. The four buttons on the left represent the virtual desks and the button of the currently displayed desk is highlighted. When the user clicks an un-selected desk, SmallDesk hides all VisualWorks windows on the current desk and displays all VisualWorks windows assigned by the user to the selected desk. The Desktop window itself remains visible at all times.

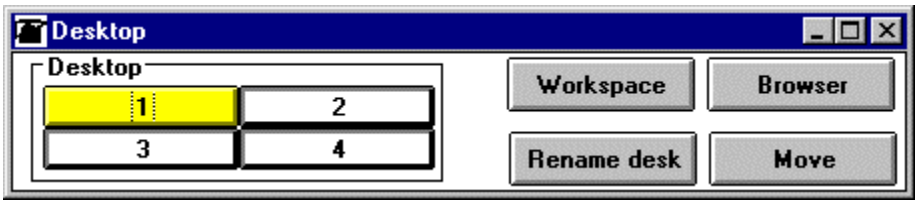

Figure 12.1. Default user interface of SmallDesk manager.

The buttons on the top right allow the user to open a Workspace and a Browser on the current desk because opening a Browser or a Workspace is the most common way to start populating a desk with windows. The two buttons on the bottom right allow the user to rename desk buttons and to move a selected window from one desk to another. A desktop can also be opened with other than default desk names and the user can specify which applications should be displayed on individual desks.
Additional specification details:

- When SmallDesk opens with the default open message, it selects desk 1 and maps (displays) all windows already on the screen on this desk. All other desks are empty.

- When SmallDesk closes, it maps all windows from all desks. In other words, it displays all VisualWorks windows assigned to all four desks on the screen.
- When the user clicks *Rename desk*, a prompt requests the new name for the current desk and displays it on the button.
- The *Move* button allows the user to move a window from the current desk to another desk. It opens two consecutive prompts that allow the user to identify the window to be moved and its new desk. If the destination is different from the current desk, the selected window is unmapped and assigned to the specified desk; no desk switch occurs. The Desk Manager window itself cannot be moved to another desk and is always displayed.
- The user can also open SmallDesk by one of several more powerful messages. As an example, to open the desk on desk 2 with button names, and assign application LecturePad to the first desk, two workspaces to the second, a Browser to the third, and two copies of Notes to the fourth, the user will execute

```
SmallDesk
        openWithManyApps: #(#(LecturePad) #(ComposedTextView ComposedTextView) #( Browser)
                #(Notes Notes))
        andNames: #('desk 1' 'desk 2' 'desk 3' 'desk 4')
        onDesk: 2
```

Preliminary design

We only need one class - an application model to be called SmallDesk. This class will be responsible for user interface specification, opening messages, and action buttons. We will keep the mapping of windows to desks in a dictionary, using windows as keys and integer numbers of desks as values.

The major *responsibilities* of SmallDesk are as follows:

Opening. The default opening message assigns all VisualWorks windows on the screen to desk 1 and highlights its button. Specialized opening messages do one or more of the following: rename buttons, assign specified applications to specified desks, open on specified desk.

Action methods. Action methods can be divided into the following groups:

- *Desk button* action methods. The user clicks a desk button either to switch to a different desk or to identify the destination desk when moving a window from one desk to another.
- *Browser* button and *Workspace* button open the Browser and the Workspace on the current desk.
- The *Rename* button displays a prompt and assigns the obtained string as the label to the currently selected desk button.
- The *Move* button displays a prompt to identify a window, another prompt to identify the destination desk, unmaps the window from the current desk, and assigns it to the destination map. An attempt to move the Desk Manager window itself will be refused with a warning.

Closing. This button will prompt the user to confirm when he or she attempts to close SmallDesk. It will then map all windows to the screen and close SmallDesk.

Design refinement

Before writing the specification of SmallDesk, we will examine some of the details that we ignored in preliminary design starting with instance variables. We need a window registry that holds information about the assignment of windows to desks. We also need to know the current desk's number so that we can rename it and map and unmap windows on desk change request, and check whether a new desk selection is different from the current one.

To deal with the two different modes of operation that affect the result of clicking a desk button (switching to a different desk versus renaming it), we must also keep track of the mode. We will represent it by a Symbol which will be identical to the name of the method that executes the operation so that we can execute the method by the perform: message. Finally, we will use 'lazy opening' to open applications specified by the user in the opening message. SmallDesk will initially open only applications on the opening desk, and the remaining applications will only be opened when the user switches to the desk assigned to them. We will keep unopened applications in a four-element array with one element corresponding to each desk.

We can now start thinking about the methods. Default initialization puts all VisualWorks windows on the screen on desk 1, sets current desk number to 1, and mode to 'switch to another desk'. It also initializes the array of unopened applications.

The method performing desk change checks whether the new desk is different from the current one (and do nothing if it is), change the colors of backgrounds of the current and new desk buttons, update window registry (windows may have opened or closed on the current desk since the last registry refresh and this must be recorded), unmap the windows on the current desk, map windows on the new desk, open the unopened applications assigned to this desk, and make new desk the current desk.

Finally, when the user clicks to close SmallDesk and confirms, all scheduled windows from all desks are mapped to the screen.

We can now write the following detailed specification of class SmallDesk:

SmallDesk. This class helps the user to reduce screen clutter by creating four virtual screens called desks. SmallDesk continuously displays a Desk Manager window with four buttons representing the virtual desks and allows VisualWorks windows to be associated with any one of them. Only windows assigned to the current desk are displayed. Desk buttons can be renamed and windows can be moved from one desk to another. The application can open on any desk, with any combination of desk button names, and with specified applications assigned to individual desks.

Superclass: ApplicationWindow.

Instance variables: windowRegistry <IdentityDictionary> pairs windows with desktops (integers), waitingApps <Array> contains an array with class names of unopened applications for each desk, currentDesk <Integer> identifies currently displayed desk, mode <Symbol> identifies the current context of operation - one of #changeTo:, #moveTo:

Responsibilities:

- Opening
  - openWithManyApps: andNames: onDesk: opens the application on the specified desk and with the specified applications. All currently displayed Smalltalk windows are assigned to desk 1.
  - open – special case of the above method that opens with default button names, on desk 1, and with no new applications.
- Initialization and closing.
  - initialize – the standard opening hook method. Gets all VisualWorks windows from ScheduledController to initialize windowRegistry, initializes currentDesk to 1 (overridden later when a different desk is specified), mode to #changeTo:.
  - closeRequest asks the user to confirm SmallDesk closing, maps all windows referenced by ScheduledControllers and allows the Desk Manager window to close.
- Action methods responding to UI buttons.
  - desk1, desk2, desk3, desk4 methods all act in the same way and the only difference between them is that each identifies a different desk. Consequently, they all send private message updateDesk: with their number as the argument and further processing happens in this method. Since the same method is used for changes, its operation depends on the context: It performs one of the two possible actions corresponding to the two possible contexts in which the button may be activated. These two actions will be implemented by
    - changeTo: - checks whether the new desk is different and if it is, hides currently displayed windows, displays windows assigned to the new desk, opens waiting applications assigned to this desk, and colors the buttons appropriately.

    - moveTo: is triggered by the *Move* button via the move method (below). It displays prompts to select the window to be moved and the destination desk, unmaps the window, and assigns the number of the specified desk to this window in the registry.
  - openBrowser and openWorkspace buttons open the Browser and the Workspace.
  - rename requests a new name for the currently selected desk button and assigns it to the button as its label.
  - move asks the user to select a window and click the destination desk button. The internal assignment of the desk is performed by updateDesk:.
- Private support methods.
  - updateDesk: is activated by a desk button. It performs the value of the mode variable and supplies the desk number as the argument. The result is either a change of desk (method changeTo:) or reassignment of a window to a new desk (method moveTo:), depending on the context.

Implementation

We will now list and explain some of the methods.

Opening

```
openWithManyApps: classArray andNames: nameArray onDesk: anInteger
"Open on desk anInteger, with classes as specified in classArray, and button names according to nameArray. Applications that are not on desk anInteger do not open yet."
| desk deskBuilder |
desk := self new.
“Create the desk manager but stop short of opening so that the manager window can be positioned, etc.”
deskBuilder := desk allButOpenInterface: #windowSpec.
“Rename desk buttons via the builder and button widget Ids.”
1 to: 4 do: [:index | | buttonView |
                buttonView := (deskBuilder componentAt: ('desk' , index printString) asSymbol) widget.
                buttonView label: (Label with: (nameArray at: index))].
“Position Desk Manager window in upper left.”
deskBuilder source finallyOpenIn: (10 @ 10 extent: deskBuilder window bounds extent) withType: #normal.
“Assign user specified applications to desks.”
1 to: 4 do: [:index | | classes |
                ((classes := classArray at: index) isNil or: [classes isEmpty])
                        ifFalse: [classes do:
                                        [:class | desk waitingApps at: (Smalltalk at: class) put: index]]].
“Open user-specified desk.”
desk openDesk: anInteger.
^deskBuilder
```

The opening of the desk is executed by

```
openDesk: integer
"Switch to the specified desk or edit button label or move window."
        self perform: mode with: integer
```

A part of the execution of new is execution of the hook method initialize which is defined as follows:

```
initialize
"Initialize registry of windows and desks, start in 'move' mode with existing windows in desk 1."
waitingApps := Array        with: Set new
                            with: Set new
                            with: Set new
                            with: Set new.
windowRegistry := IdentityDictionary new.
ScheduledControllers scheduledControllers do: [:controller | windowRegistry at: controller view put: 1].
```

```
self changeSystemController.
mode := #changeTo:
```

The response to a desk button clicks is handled by methods desk1, desk2, desk3, and desk4 defined as follows:

```
desk1
"Action method of button desk 1."
        self updateDesk: 1
```

The operation of updateDesk: depends on the current mode and its definition is as follows:

```
updateDesk: integer
"Depending on the context, switch to integer desk or edit button label or move window."
self perform: mode with: integer
```

Depending on the value of mode, this method performs one of changeTo: or moveTo:. Method changeTo: switches from one desk to another and its definition is

```
changeTo: newDesk
"Switch to the specified desk if different from the current one."
newDesk = currentDesk ifTrue: [^nil].
“Change button colors.”
currentDesk isNil ifFalse: [self colorButton: currentDesk color: ColorValue white]. self colorButton: newDesk color: ColorValue yellow.
“Update the registry – windows may have been added or removed.”
self refreshWindowRegistry.
currentDesk := newDesk.
“Go through registry, mapping windows on the current desk and unmapping the rest. Always keep the Desk Manager – the window corresponding to self.”
windowRegistry keysAndValuesDo: [:key :value | (key = self builder window or: [value = newDesk])
                        ifTrue: [key = self builder window ifFalse: [key map]]
                        ifFalse: [key unmap]].
“Open unopened applications waiting on this desk, if any.”
(waitingApps at: currentDesk) do: [:app | self open: app]
```

An important part of the definition is method refreshWindowRegistry which is defined as follows:

```
refreshWindowRegistry
"We are switching to a different desk. Make sure to update the desk being abandoned - windows may have been added or deleted."
| newRegistry |
“Create a new empty registry .”
newRegistry := IdentityDictionary new.
“Copy active windows in the registry and new windows to the new registry.”
ScheduledControllers scheduledControllers do:
                [:controller |
                | window |
                (windowRegistry includesKey: (window := controller view))
                        ifTrue: [newRegistry at: window put: (windowRegistry at: window)]
                        ifFalse: [newRegistry at: window put: currentDesk]].
“Replace old registry with new one.”
windowRegistry := newRegistry
```

The open: method on the last line of changeTo: opens a waiting application, removes it from its array and adds the resulting window to the registry:

```
open: anApplication
"Open anApplication, remove it from waitingApps, and add its window to the registry."
```

```
| window |
(waitingApps at: currentDesk) remove: anApplication.
window := anApplication open.
windowRegistry at: window put: currentDesk
```

Note that we assume that all unopened applications open with the open message. This restriction could, of course, be easily removed. The moveTo: method moves a selected window to a selected desk. Its definition is

```
moveTo: newDesk
"Move previously selected window to the specified desk (registry) and hide it."
newDesk = currentDesk ifTrue: [^nil].
windowRegistry at: currentWindow put: newDesk.
currentWindow unmap.
mode := #changeTo:
```

Note that the method ends by switching to the default mode of operation.

Who sends the moveTo: message? Its trigger is the activation of the *Move* button whose action method move first lets the user select a window by waiting until ScheduledControllers reports that the user pressed the red button, and then identifies the active window. After this, the user is asked to click a desk button (to identify the destination desk). The mode changes to #moveTo:, and when the user clicks one of the desk buttons, this sends updateDesk: (as listed above), which then performs moveTo:. The action method move is defined as follows:

```
move
"User clicked Move to move a window to a different desk. Identify the window and ask user to select
desk. The rest of the operation is handled in moveTo:."
Dialog warn: 'Select window to be moved'.
“Wait for user to press leftmost mouse button.”
[ScheduledControllers activeController sensor redButtonPressed]
        whileFalse: [].
“Get active window, assuming that this is the one the user wants.”
currentWindow := Window currentWindow.
“Don’t let the user try to move the Desk Manager window to a different desk.”
currentWindow = self builder window ifTrue: [^Dialog warn: 'You cannot move Desk Manager window'].
Dialog warn: 'Click destination desk'.
mode := #moveTo:
```

Finally, opening the browser and the workspace is done as follows:

```
openBrowser
"Open Browser on current desk."
        (Smalltalk at: #Browser) open
```

and

```
openWorkspace
"Open Workspace on current desk."
        (Smalltalk at: #ComposedTextView) open
```

Exercises

1. Design and write a method to open SmallDesk with applications whose opening message is not open.
2. An alternative to unmapping windows is to collapse them to icons. Change SmallDesk to use collapsing.
3. SmallDesk mechanism for moving a window to another desk is awkward: The user must click the *Move* button, complete the dialog, click a window, complete another dialog, and finally click the

destination desk. To simplify the process, add a new *'move to another desk'* command to the <window> menu *when SmallDesk is running*. When the user selects this command, SmallDesk asks him or her to click a desk button, and when the user does so, SmallDesk moves the window to that desk.

12.3 Principles of displaying - graphics contexts, geometric objects, and other concepts

In this section, we will explain the principles of displaying components inside a window and illustrate them on simple examples.

Graphics context

When we want to display a geometric object such as a rectangle, we must specify which part of the display surface we want to paint on, and which color and line width to use. For text, we must also know the font. VisualWorks keeps this information in a GraphicsContext object attached to the display surface. The GraphicsContext knows about its graphics medium and is also responsible for all display operations. We will see numerous examples of the use of GraphicsContext but first, we will examine its definition. The place of GraphicsContext in the class hierarchy is as follows:

```
Object
        GraphicsContext
                HostPrinterGraphicsContext
                PostScriptGraphicsContext
                ScreenGraphicsContext
```

and the essence of its comment says

I display graphical objects on a display medium and maintain graphics state parameters specifying how graphical objects should be rendered.

Instance variables:

medium	<GraphicsMedium>	where to render graphics
clipOriginX	<nil | SmallInteger>	in untranslated device coordinates
clipOriginY	<nil | SmallInteger>	In untranslated device coordinates
clipWidth	<nil | SmallInteger>	
clipHeight	<nil | SmallInteger>	
lineWidth	<SmallInteger>	draw lines with this width in pixels
capStyle	<SmallInteger>	style to render line caps
offsetX	<SmallInteger>	translation
offsetY	<SmallInteger>	translation
phaseX	<SmallInteger>	tile phase for tiling the background
phaseY	<SmallInteger>	tile phase
scaleX	<Float>	scaling factor
scaleY	<Float>	scaling factor
font	<ImplementationFont>	font for drawing strings
paint	<Paint>	color or tile for displaying graphical entities
paintPolicy	<PaintPolicy>	how to render paints that do not exactly correspond to host paints
fontPolicy	<FontPolicy>	how to map FontDescriptions to the fonts available on the graphics device

A more detailed description of some of the less obvious instance variables is as follows:

clipOriginX, clipOriginY, clipWidth, clipHeight describe the clipping rectangle - the part of the display surface on which the GraphicsContext will draw. This rectangle acts as a stencil and all drawing that falls outside it is ignored.

capStyle specifies the style used for straight line endings; it is visible only on thick lines.

offsetX, offsetY: translates coordinates of graphics context with respect to the origin of its medium
scaleX, scaleY: scaling factor
paint: Default paint used when the visual part being drawn does not specify its paint. The concept of a paint requires a brief comment. Display surfaces can be painted either with colors or with patterns. The distinction is similar to the distinction between painting a wall and wallpapering it: A color paints the whole object the same color (filling a rectangle with red color, for example) whereas pattern repeats a tiling pattern across the whole area.
phaseX, phaseY: Tile phase specifies the starting point of the first tile of a painting pattern.
paintPolicy, fontPolicy: If the platform cannot provide the required color or font, this object finds a suitable available replacement.

Most display methods are defined in this class, and subclasses implement differences between screen display and printing such as conversion between pixels (screen distance units) and printer units (corresponding but different printer distance units).

We will now present two examples showing how to use the GraphicsContext to display text and geometric objects in a window, how to control display via the graphics context, and how to use patterns. You will note that our code does not provide automatic damage repair which means that if you change window size, collapse and expand it, or partially cover the window and then uncover it, the drawing will be lost. As we noted earlier, automatic damage repair is a separate mechanism that will be covered later.

Example 1: Drawing geometric objects and controlling their color

Problem: Implement an application with the user interface in Figure 12.2. Clicking *Color* opens a multiple choice dialog listing all available colors and accepts the selected color for drawing. Clicking *Draw* opens a dialog requesting information needed to display the object selected by the radio button, for example the end points of a straight line when *Line* is selected. A predefined clipping rectangle inside the window is used to restrict display to the upper part of the window.

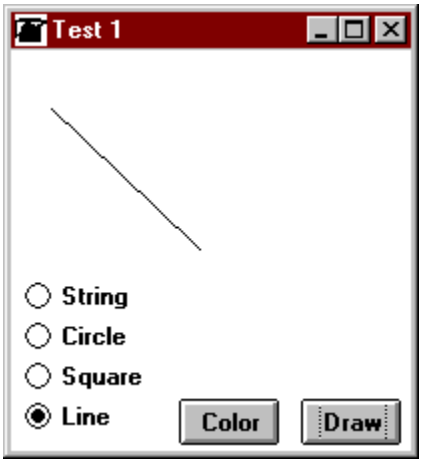

Figure 12.2. Example 1: Clipped drawing of a line with end points in the upper left and lower right corners of the application window.

Solution: The solution revolves around the graphics context which controls the parameters and does the drawing. The graphics context is obtained from the window which, in turn, is known to the builder of the application. From within the application model, the graphics context of the window is thus accessed by

```
builder window graphicsContext
```

Although this expression suggests that a window keeps its graphics context, this is not so. Instead, a window manufactures a graphics context with default parameters whenever it is asked for one. Changes

made to a graphics context are thus transient, and when a graphics context is modified and then retrieved another time, it will again be a graphics context with default parameters. This is an important but often forgotten principle.

To implement the program, we painted the user interface installed it on class DrawObjects, and defined instance variables color, clippingRectangle, object, and width and initialized them as follows:

```
initialize
"Set initial parameters for use on graphics context."
        clippingRectangle := 20 @ 20 corner: 180 @ 100.     "Upper part of the window."
        color := ColorValue black.                          "Default color."
        object := #Line asValue.                            "Aspect variable of the radio button group."
        width := 1
```

The color action method gets a color Symbol from a dialog and executes it to get a ColorValue:

```
color
“Get color from user and save it.”
| colorName |
colorName := Dialog     choose: 'Choose color'
                        fromList: ColorValue constantNames      “Names of built-in named colors.”
                        values: ColorValue constantNames
                        lines: 8
                        cancel: [nil].
colorName isNil ifFalse: [color := ColorValue perform: colorName]
```

The action method of the *Draw* button obtains the window’s graphics context, changes its paint, line width, and clipping rectangle according to instance variables, and sends a message to draw the selected object:

```
draw
"Get graphics context, set its parameters, and execute appropriate drawing message.
        | gc |
        gc := self builder window graphicsContext.
        gc paint: color.
        gc lineWidth: width; clippingRectangle: clippingRectangle.
        self perform: 'draw', object value, 'On:' with: gc
```

The last statement is possible because we assigned radio buttons aspects to match the required display message, such as #Circle. As a consequence, when the *Circle* button is selected, the last line is equivalent to self drawCircleOn: gc. Each of the drawing methods then obtains the appropriate parameters from the user and asks the graphics context to draw the object. As an example, the method to draw a square is defined as follows:

```
drawSquareOn: aGraphicsContext
"Get origin coordinates and side length from user, draw a square.
        | startx starty side |
        'Obtain origin and side of square from user.”
        startx := (Dialog request: 'Upper left x' initialAnswer: '30') asNumber.
        starty := (Dialog request: 'Upper left y' initialAnswer: '30') asNumber.
        side := (Dialog request: 'Side' initialAnswer: '20') asNumber.
        "Draw unfilled square inside the clipping rectangle using current parameter settings."
        ^aGraphicsContext displayRectangularBorder: (startx @ starty extent: side @ side)
```

It is worth noting that instead of asking a graphics context to draw a visual object, we can also ask the object to draw itself using the graphics context. As an example, the effects of

```
aGraphicsContext displayRectangularBorder: aRectangle
```

and

```
aRectangle displayOn: aGraphicsContext
```

are exactly the same. The advantage of the second method is that it can be used even if we don't know the kind of visual part being drawn, as when we have a whole collection of objects to draw:

```
objects do: [:object| displayOn: aGraphicsContext]
```

We will use this approach in another example.

Example 2: Drawing with patterns and drawing with color

Problem: Design an application with the user interface Figure 12.3. When the user selects a color via the *Color* button and clicks *Draw*, the program requests square parameters from the user and draws a square filled with the selected color. When the user clicks *Pattern*, the program displays the cross hair cursor and the user selects a rectangle on the screen. If the user then clicks *Draw*, the program requests square parameters and tiles the square with copies of the selected area.

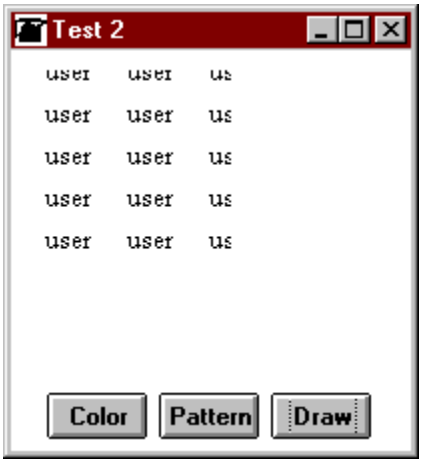

Figure 12.3. Example 2: Displaying a square filled with a pattern copied from the screen.

Solution: In class Patterns, we define instance variable paint, and initialize it to black color:

```
initialize
        paint := ColorValue black
```

The color method is as in Example 1 and the action method of the *Pattern* button is as follows:

```
pattern
"Ask user to select screen area with desired pattern."
        Dialog warn: 'Select a rectangle on the screen'.
        paint := Screen default contentsFromUser asPattern
```

The action method of the *Draw* button obtains a rectangle from the user, assigns the paint to the graphics context, and asks the graphics context to draw the square with the assigned fill:

```
draw
"Obtain square parameters from user and draw the square filled with the current paint."
        | startx starty side gc |
        "Get parameters."
```

```
gc := self builder window graphicsContext.
gc paint: paint.
startx := (Dialog request: 'Upper left x' initialAnswer: '30') asNumber.
starty := (Dialog request: 'Upper left y' initialAnswer: '30') asNumber.
side := (Dialog request: 'Side' initialAnswer: '20') asNumber.
“Disply.”
^gc displayRectangle: (startx @ starty extent: side @ side)
```

Main lessons learned:

- All display within a window is performed by an instance of GraphicsContext.
- GraphicsContext is central to everything related to display of visual components.
- In addition to being the drawing engine, a GraphicsContext also holds display parameters such as line width, paint, and fault.
- Display parameters stored in the graphics context can be controlled programmatically.

Exercises

1. Browse and explain the operation of GraphicsContext.
2. Browse and explain class Paint.
3. Modify Example 1 to obtain the clipping rectangle from the user rather than using a fixed preprogrammed value. (Hint: Look for implementers of method fromUser:.)
4. Modify Example 2 to ask the user to specify a rectangle interactively instead of obtaining square parameters via a dialog. (Hint: Look for implementers of method fromUser:.)
5. We mentioned that asking an object to draw itself on a graphics context is equivalent to asking the graphics context to draw the object. This suggests that one of the two approaches is implemented by double dispatching. Check whether this is indeed the case.

12.4 Images, pixmaps, masks, and paints

In addition to displaying geometric objects and text, we can also display images and use masks to control which part of the image is visible. In this section, we will explain the principles of these concepts and illustrate them on simple examples. We will also talk about cursors because they are related but we will leave an example of their use for later.

Image

An Image (more accurately an instance of one of its concrete subclasses) is a rectangular array of pixels, each represented internally by a of group of bits and displayed as a colored dot. Since different images may use a different number of bits to represent pixel color, a single class cannot represent all possible kinds of images. Image is thus an abstract class which leaves concrete image representation to its subclasses:

Image
- Depth1Image
- Depth2Image
- Depth4Image
- Depth8Image
- Depth16Image
- Depth24Image
- Depth32Image

The depth of an image is the number of bits used to represent each pixel. An image with depth 1 (monochrome image) uses one bit per pixel and allows only two colors (represented as 0 and 1 respectively), depth of 2 allows $2^2 = 4$ different colors encoded as 00, 01, 10, and 11, depth of 8 means that each pixel may have any one of $2^8 = 256$ colors, and so on. The greater the depth, the more subtle an image (if your computer can display them) and the more memory required to store the image. In ordinary situations, you don't need to be concerned about depth, and address all your messages to class Image.

The most common ways of creating an image are capturing an image on the screen or reading it from a file. We will see that cursors, which use an Image as a part of their definition, often create an image by specifying the individual pixels.

Example 1: Capturing an image and painting it in a window

Problem: Write a program to let the user capture an image from the screen and paint it in the currently active window. The upper left corner of the image should be located in the center of the window. Figure 12.4 shows what the outcome might look like.

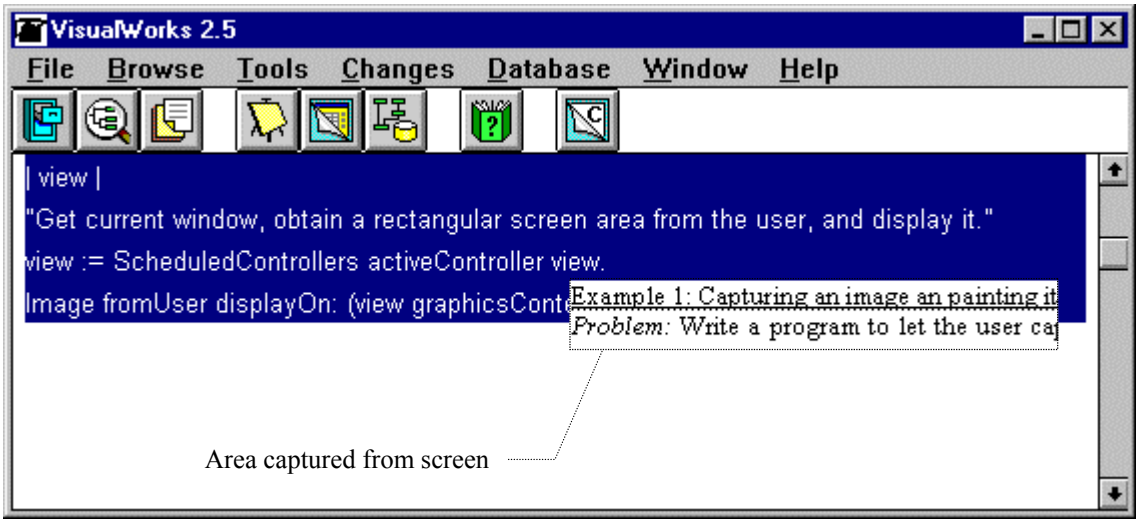

Figure 12.4. Example 2: Displaying image from user in the active window.

Solution: To get an image from the user send message fromUser to Image. To find the currently active window, ask Window. To find its center, ask for its bounds rectangle and its center. Finally, get its graphics context and display the image. The graphics context displayOn:at: message displays the image at the specified point. The whole solution is as follows:

```
| view |
"Get current window, obtain a rectangular screen area from the user, and display it."
view := Window currentWindow.
Image fromUser displayOn: (view graphicsContext) at: (view bounds center)
```

The following example is from the library,

```
ScheduledWindow new
        component: ((BorderDecorator on: Image fromUser)
                useHorizontalScrollBar);
    open
```

It opens a default-size window showing an image captured from the screen by the user. Note how we specify the image as the window's component. We can improve on this code a bit by making the window the same size as the captured window as follows:

```
| image |
ScheduledWindow new
        component: ((BorderDecorator on: (image := Image fromUser))
                useHorizontalScrollBar);
    openIn: ((10@10) corner: image bounds extent)
```

Example 2: Image from file

Problem: Write a code fragment to read a graphics file and display it in a scheduled window.
Solution: Abstract class ImageReader can read a limited number of image file formats, delegating its conversion to an Image to a subclass corresponding to the encoding. Sending message fromFile: to ImageReader returns an object containing the image; the image can then be extracted with message image and displayed in the usual way. As an example,

```
(ImageReader fromFile: 'C:\MSOffice\Clipart\flower.bmp' ) image
    displayOn: Window currentWindow graphicsContext
```

displays an image from the specified file. Message display: displays the image in the upper left corner (origin) of the window but displayOn:at: allows you to specify the position.

Cursor

A cursor is the visual representation of the current position of the mouse on the screen. On first thought, a cursor is just an image that can be moved around. On closer examination, cursors are partially transparent, showing some of the underlying background as the cursor moves across the screen. This is achieved by combining the cursor's image with a *mask*, a concept discussed in more detail later.

Class Cursor defines more than 20 predefined cursor shapes (such as write, hand, and crossHair) and some interesting cursor behaviors. In addition, the user can create new cursors as well.

As an example of a built-in cursor, the cross hair cursor is defined as a class variable in class Cursor and initialized as follows:

```
initCrossHair
	CrossHairCursor := (self
		imageArray: #(
			2r0000000000000000
			2r0000000100000000
			2r0000000100000000
			2r0000000100000000
			2r0000000100000000
			2r0000000100000000
			2r0000000100000000
			2r0111111111111110
			2r0000000100000000
			2r0000000100000000
			2r0000000100000000
			2r0000000100000000
			2r0000000100000000
			2r0000000100000000
			2r0000000000000000
			2r0)
		maskArray: #(
			2r0000001110000000
			2r0000001110000000
			2r0000001110000000
			2r0000001110000000
			2r0000001110000000
			2r0000001110000000
			2r1111111111111111
			2r1111111111111111
			2r1111111111111111
			2r0000001110000000
			2r0000001110000000
			2r0000001110000000
			2r0000001110000000
			2r0000001110000000
			2r0000001110000000
			2r0)
		hotSpot: 7@7
		name: 'crossHair')
```

The method defines the four Cursor instance variables: the image, the mask, the position of the hot spot, and the cursor name. The *hot spot* is the Point returned when a program requests cursor position and represents the mouse position on the screen. The individual pixels of the image and the mask are specified as binary numbers such as 2r0000001110000000 (2r means radix 2 - binary representation). The 1's in the *image* array are the black pixels and 0's are white pixels and when you half close your eyes and look at the

pattern of 0's and 1's, you can see the crosshair shape easily. The 1's in the *mask* define the pixels that are opaque (*not* transparent), in this case the crosshair and a one-pixel band around it. 0's are transparent and allow you to see the background behind the cursor. You can use this same method to define your own cursors and Appendix 4 shows how you can create a cursor with the Image Editor tool.

Class Cursor is typically accessed to display a special cursor during a special operation. As an example, VisualWorks displays a special cursor while reading a file, while performing garbage collection, or while waiting for the user to select a point on the screen (for this, it uses the crossHair cursor above). The two common messages for changing the cursor are showWhile: aBlock and show. The showWhile: message changes the cursor for the duration of execution of its block parameter, and changes it back to its original shape when aBlock executes. A typical example of its use is in the following method from class FileBrowser:

```
hardcopyStream: aStream
"Use the 'wait' cursor while printing aStream. "
        Cursor wait
                showWhile: [aStream contents asText asParagraph hardcopy]
```

Method show also changes the cursor but the programmer must remember to change it back. Its typical use is shown in the definition of showWhile: itself:

```
showWhile: aBlock
"While evaluating aBlock, make the receiver be the cursor shape. Then revert to the original cursor."
        | oldcursor |
        oldcursor := self class currentCursor.
        self show.
        ^aBlock  valueNowOrOnUnwindDo:
                        [oldcursor show]
```

Pixmaps – drawing surfaces behind the scenes

A Pixmap is a display surface and you can 'display' on it, just as on the screen. However, displaying on pixmap does not show on the screen until you ask the screen's graphics context to display the pixmap. Pixmaps are thus used to construct displayable rectangles before they are displayed on the screen. The typical procedure is to transfer an Image to a Pixmap, modify it by a mask to display only some selected part of it, and ask a graphics context to display the result. We will show an example of this shortly. Pixmaps can also be created by copying the contents of the clipboard of your machine using the clipBoard

Extracting a part of an image - masks

To display a part of an image, transfer the original image to a Pixmap, combine it with a Mask, and display the result. If the desired mask is rectangular, a mask is not necessary because the Pixmap can do rectangular clipping itself. We will now describe and illustrate both techniques.

Displaying a rectangular part of an image without using a mask

1. Create the image as a Pixmap object or convert an Image to a Pixmap using asRetainedMedium.
2. Construct the clipping rectangle.
3. Apply the clipping rectangle to the Pixmap and extract the corresponding area as Image. Use message completeContentsOfArea: clippingRectangle.
4. Display the obtained Image on a graphics context.

Example 3: Extracting a rectangular portion of an image without using a mask
Problem: Display the image from the previous example and its lower half side by side (Figure 12.5).

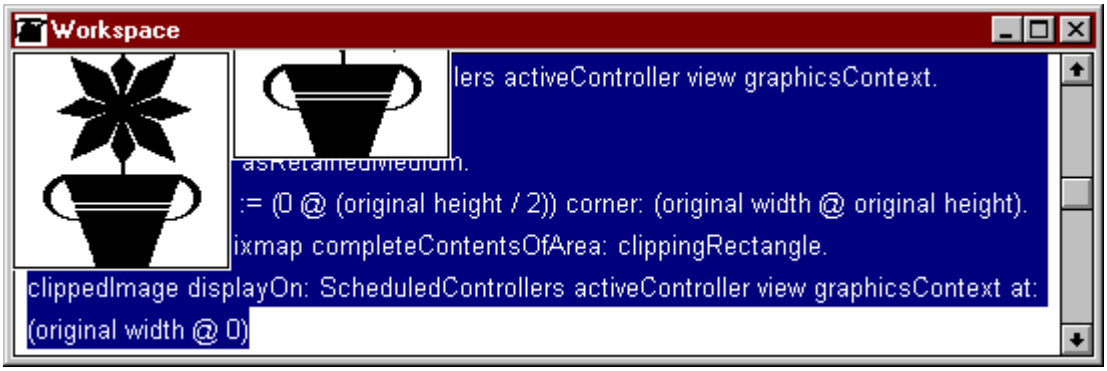

Figure 12.5. Example 3: Image from the Microsoft Office clip art library and its rectangular part.

Solution: Following the procedure described above and using a .bmp file, the solution is as follows:

```
| gc clippingRectangle clippedImage original pixmap |
"Get image from file."
original := (ImageReader fromFile: 'C:\MSOffice\Clipart\flower.bmp' ) image.
"Display original image on window's graphics context."
gc := Window currentWindow graphicsContext
original displayOn: gc.
"In memory, construct a Pixmap containing the original image."
pixmap := original asRetainedMedium.
"Construct clipping rectangle."
clippingRectangle := (0 @ (original height / 2)) corner: (original width @ original height).
"Extract area corresponding to rectangle from pixmap."
clippedImage := pixmap completeContentsOfArea: clippingRectangle.
"Display the clipped image on the graphics context next to the original."
clippedImage displayOn: gc
                    at: (original width @ 0)
```

An alternative approach is to use a Mask. This method is more powerful because it allows the clipping area to have any shape. During the discussion of Cursor, we saw that a mask acts as a decal – a display area painted in such a way that it is transparent in some places but opaque in others. A mask is thus like a surface covered with a 'paint' that is either transparent or opaque and this is why the paint is called *coverage value*.

Extracting a non-rectangular part of an image with a mask

1. Create the image as a Pixmap object or convert an Image to a Pixmap using asRetainedMedium.
2. Construct the clipping mask. The shape drawn on the mask will be the transparent area.
3. Display the Pixmap overlaid with the mask on the graphics context using copyArea:from:sourceOffset: destinationOffset:. You can control the position of the mask with respect to the pixmap by the argument of sourceOffset:, and the position of the pixmap with respect to the display area by the argument of destinationOffset:.

Example 4: Extracting a triangular portion of an image with a mask

Problem: Display an image from user and its above-the-diagonal part in the current window (Figure 12.6). Shift the clipped part so that its origin coincides with the corner of the original image.

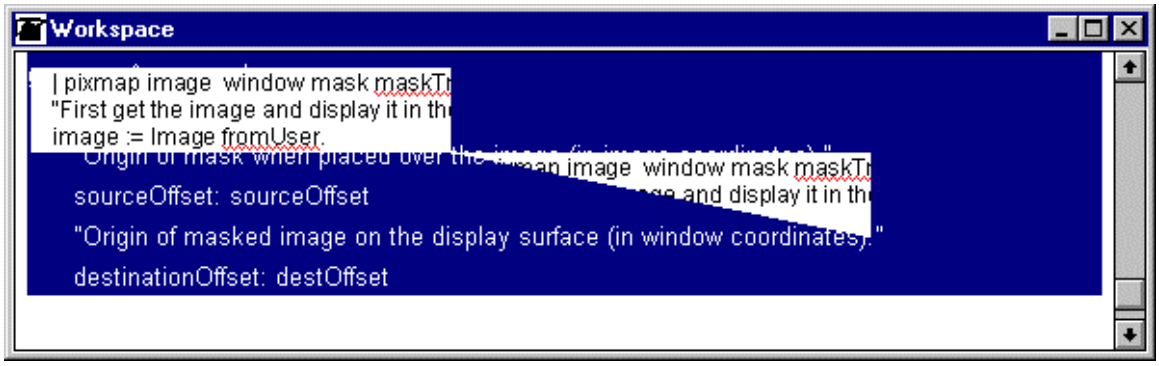

Figure 12.6. Example 4: Bitmap image clipped with a mask.

Solution: Following the steps listed above we create a Pixmap, a Mask containing the masking triangle, and display the combined image and mask on the graphics context of a window:

```
| pixmap image  window mask maskTriangle gc imageRectangle destOffset sourceOffset |
"First get the image and display it in the current window."
image := Image fromUser.
window := Window currentWindow.
gc := window graphicsContext.
gc display: image at: 10@10.
"Now start constructing the masked image. First transfer image to pixmap."
pixmap := image asRetainedMedium.
"Find the bounding box of the image and use it to construct mask rectangle."
imageRectangle := image bounds.
mask := Mask extent: imageRectangle extent.
maskTriangle := Array       with: 0 @ 0
                            with: (imageRectangle width) @ 0
                            with:  imageRectangle width @ imageRectangle height.
"Display the triangle on the mask."
mask graphicsContext displayPolygon: maskTriangle.
"Display masked image on window's graphics context at an offset."
sourceOffset := 0@0.        “No offset from pixmap origin to mask origin.”
destOffset := imageRectangle extent + (10@10). “Window origin to pixmap origin offset.”
gc          copyArea: mask
            from: pixmap graphicsContext
            "Origin of mask when placed over the image (in image coordinates)."
            sourceOffset: sourceOffset
            "Origin of masked image on the display surface (in window coordinates)."
            destinationOffset: destOffset
```

Main lessons learned:

- An Image is a pixel-by-pixel representation of an image displayed on screen or printed by printer.
- An Image may use one of several depths, resulting in greater or smaller color variety and a corresponding increase or decrease of memory requirements.
- An Image can be read from a file, copied from the screen, or constructed from pixels.
- Class Cursor defines the visual representation and behaviors of the mouse cursor.
- Cursor shape is defined by a 16@16 image and a 16@16 mask that determines which part of the cursor is transparent.
- An Image can be clipped to any shape with a mask.
- A Pixmap is a display surface stored in memory but not necessarily displayed on the screen.

Exercises

1. You may have noticed that our image from use in Example 1 does not show the whole rectangle selected by the user. Why is it? Can you correct this shortcoming?
2. In Example 4, we set the offset of the origin of the mask at the origin of the pixmap. What happens when you use a non-zero offset?
3. Write a simple clip art browser that finds all .bmp files on a user-specified disk drive, lists their names in a multiple choice dialog, and displays user's selection in a new ScheduledWindow.
4. It would be nice if CoverageValue could handle any values between 0 and 1, making pixels less and less transparent as the value moves from 1 to 0. A reasonable approximation of this behavior is to make some of the pixels transparent and leave the remaining pixels opaque. As an example, if the coverage value of all pixels in a 10 by 10 rectangle is 0.6, we could make 60 percent of all pixels opaque and 40 percent transparent using random assignment of 0's and 1's in proportions corresponding to the specified coverage value. Implement this idea.
5. Explore class Pixmap, in particular its comment and displaying protocol. Write a summary.
6. Explore class Mask and write a summary.
7. Explore class Image, in particular its bit accessing and image processing protocols. Write a summary.
8. Write a method that creates a smooth visual transformation of one image to another. Allow several styles of transitions such as left to right, top to bottom, and so on.

12.5 Models, views, and controllers revisited

Although the graphics context, display surfaces, images, and the other concepts presented above provide a lot of power, they don't address three essential aspects of user interfaces: interactive user control, window damage repair, and automatic dependency on domain data. We will now explain what we mean by these concepts and how VisualWorks deals with them.

- By *user control*, we refer to the ability of the user to interact with the application using the mouse and the keyboard. Clearly, the programs that we wrote in the previous sections did not allow any user interaction with the painted objects and images. We will see shortly that the concept of a *controller* provides a mechanism for user control.
- *Automatic damage repair* is another GUI feature that we take for granted: If a window is 'damaged', for example collapsed to an icon and then expanded, or covered by another window and then uncovered, or simply resized, we expect that the damage will be automatically repaired. In other words, we expect a certain permanence of the GUI. You may have noticed that the drawing and painting in our example programs did not have this permanence. If you painted something on the sample window and then collapsed and restored it, the painting was gone. We will see that the basic mechanism for automatic repair can be obtained by subclassing views to the *View class*.
- Finally, we expect that if the value of the *domain model changes* in a way that should be reflected in the view, the user interface will automatically adjust to this change. As an example, if the window displays

a circle and the application changes its diameter, we expect that the circle will be automatically redrawn. The previous sections do not give any hints as to how this could happen but we will see that the principle of *dependency* implements this mechanism.

In the rest of this section, we will explain the principles of views, controllers, and their dependency on models. The remaining sections will then explain the basic details and demonstrate how they can be used to build new UI components by using the 'any component' widget of the UI painter called the *view holder* or the *subview*.

As we already know from Chapter 6, the principle of user interface components in Smalltalk is the *model-view-controller (MVC) paradigm*. According to this principle, every UI component that represents a value of a domain object uses this value as its *model*, the graphical representation of the model is the responsibility of a *view*, and the object that manages user interaction within the boundaries of a view is the view's *controller*. We will now outline how the MVC paradigm implements the three UI operations listed above.

User interaction

User interaction via mouse and keyboard is constantly monitored by the operating system. When an event occurs, the operating system sends information about it to the running application, in this case VisualWorks. After some opening message sends, an instance of a concrete subclass of Event sends a message corresponding to the event that occurred to the active controller, usually the controller responsible for the screen area under the cursor. The message contains information about the kind of event that has occurred and its parameters. If the controller is interested in the event, it must have its definition of the method, otherwise a 'do nothing' hook definition inherited from class Controller is executed.

As an example, the controller of a chess board view will need its own definition of redButtonPressedEvent: that will respond when the player presses the red button over a chess board square.

Damage repair

When a window or its part is damaged, it records the smallest rectangle containing the damage. When processing priority allows it, the window redraws the damaged rectangle or the union of all damage rectangles. To do this, the window constrains its GraphicsContext to the clipping rectangle corresponding to the damaged area and sends a message to its components, asking them to redraw themselves.

Unless the window consists of only one widget, it contains a 'composite part' containing other parts, as in Figure 12.7. Damage repair sends message displayOn: graphicsContext down the tree to all parts of the window and eventually reaches all views. Each view must have a definition of displayOn: (possibly inherited) and this definition redraws the view within the limits of the clipping area of the graphics context.

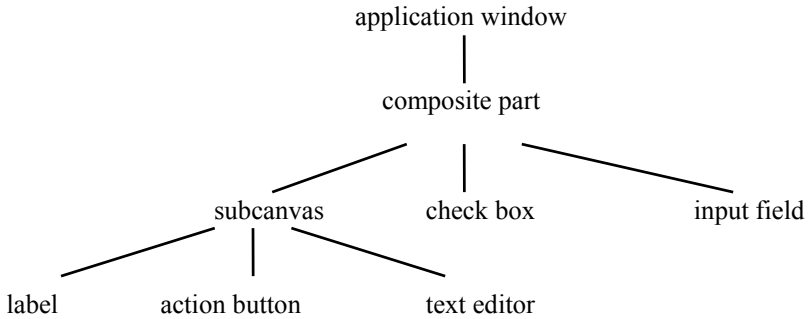

Figure 12.7. Simplified structure of a multi-component window.

Redrawing caused by a change in the model

In an active application, the value of a model frequently changes. As an example, a stopwatch application using integer seconds as its model changes the value of the model every second. There are two ways in which this change may be translated into a view change:

The preferred approach is to use *dependency*, already introduced earlier. To use dependency, the programmer includes a changed message in each method that changes the model in a way that affects the view. In the chess example (Figure 12.8), the model method that changes the configuration of pieces on the board when a player makes a move calculates new piece positions and sends changed to self[1]. In response to this, the model automatically sends the update message to each of its dependents - in this case the chess board view. The chess board view's customized update method then asks the model for the necessary data and redraws itself appropriately. To redraw itself, the view usually sends invalidate to self and the invalidate message travels up the component tree to the window at the top, and the window then sends display down the component tress in the same way as after a request for damage repair.

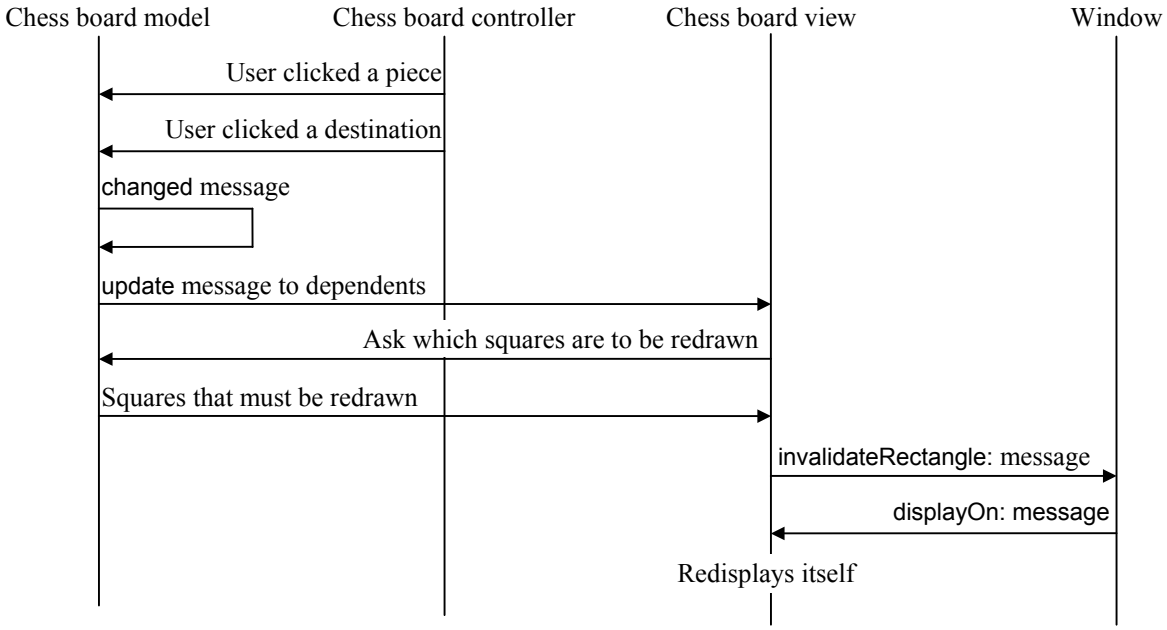

Figure 12.8. Communication involved in moving a chess piece from one square to another. Redrawing based on dependency and changed message is assumed.

The other way in which a model change may be translated into view redrawing is to send the invalidate message directly from the model's method when the change occurs. The result is the same as when we use dependency but the implementation is less centralized.

It is useful to note that invalidation provides two refinements. One is that we can specify whether the redrawing of the view must occur immediately or not. (This gives rise to the concept of 'lazy repair'.) The message may also specify that only a part of the view rather than the whole view should be redrawn and this can greatly improve the behavior of the user interface. As an example, when the position of a chess piece changes, only the start and the destination squares need to be redrawn. (The exceptions are capturing 'en passant' which affects three squares, and castling where four squares must be redrawn.) Invalidating only two squares instead of redrawing the whole board means that only 2/64 of the view must be redrawn and although this does not speed up the process, it eliminates unpleasant flashing that would otherwise occur. We use this technique in our implementation of the chess board in Appendix 3.

[1] The best way to do this is to change the aspect only via an accessing method that includes the changed message.

After this general introduction, we are now ready for details and examples. In the next two sections, we will explain how to create views using the view holder widget and how to create controllers. A larger example showing the implementation of the chessboard is given in Appendix 3.

Main lessons learned:

- The principle of Smalltalk's user interfaces is the separation of domain objects, display objects, and user interaction objects into a model, a view, and a controller. This principle is called the MVC paradigm.
- The model of a visual component holds the displayed domain object and treats the view as its dependent. When the model changes in a way that should affect the view, it sends itself a changed message which triggers notification of the view and any other dependents via update messages. The view is responsible for having an appropriate definition of update to redisplay itself.
- The view is responsible for drawing the model's visual representation on the screen. It should be a subclass of View to inherit basic damage repair mechanism and connections to its model and controller.
- The controller is responsible for handling mouse and keyboard events within the view's limits.

12.6 Creating UI components with the view holder widget

Older versions of Smalltalk did not have user interface painters and the only way to create a graphical user interfaces was programmatic. With the current technology, the UI Painter almost eliminates the need to program views and controllers because standard widgets are available from the Palette and non-standard UI components can be created rather easily with the *view holder* (*subview*) widget. (We will use the term view holder for the widget, and subview for the visual component displayed in the view holder.)

The process of creating a new component using the view holder widget is simple. It consists of painting the view holder on the canvas, defining its properties, creating the view that it will display, and defining its controller. We will now demonstrate the procedure on two simple applications that do not allow user interaction and therefore do not require a custom controller. In the next section, we will then give an example that requires a custom controller and show how to design it.

Example 1: Image display.

*Problem: D*esign and implement an application with the user interface in Figure 12.9. Before the window opens, the program asks the user to select two images on the screen. When the window opens, the view shows help text and when the user clicks one of the image buttons, the view displays the corresponding image.

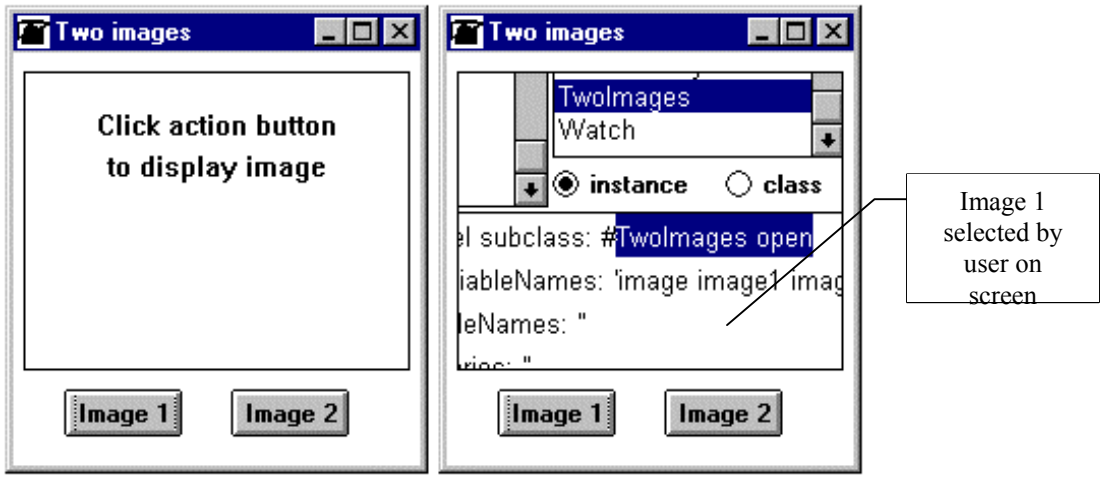

Figure 12.9. Example 1: Desired user interface when the application first opens (left), and when the user clicks an image button (right).

Solution: We will solve this problem by using dependency. The application model will be the model of the subview and when the user clicks an action button, the application will execute the changed message. This will trigger the update message in the subview which will then request the image from its model (the application) and invalidate itself. This will cause the subview to redraw itself.

The solution requires an application model (class TwoImages) and a view class (class ImageView) for the subview displayed by the view holder. Before discussing the details of implementation, we will first describe how to create the view holder:

Creating a view holder

1. Create the canvas and paint the view holder (Figure 12.10).
2. Define view holder properties, at a minimum the *View* property which is the name of the method that returns the subview to be displayed in the view holder. We called this method imageView.
3. Install the canvas.

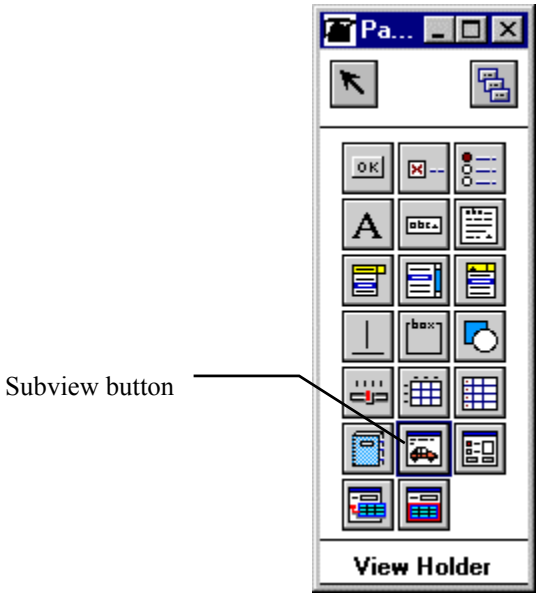

Figure 12.10. The view holder action button on the UI Palette.

Design

Class TwoImages is the application model class. It will have instance variables to hold the subview (imageView), the currently displayed image (image), and the two images selected by the user on the screen (image1 and image2). During initialization, TwoImages will ask the user to select two rectangles on the screen and save the corresponding images in image1 and image2. When the user clicks an action button, the corresponding action method assigns the appropriate image to image and sends the changed message.

Initialization must also create an instance of a view (class ImageView), assign it to imageView, and assign itself as its model. (ApplicationModel inherits dependency from its superclass Model.) The definition of initialize is thus as follows:

```
initialize
"Get two images from user and define subview."
        Dialog warn: 'Select the first image.'.
```

```
        image1 := Image fromUser.
        Dialog warn: 'Select the second image.'.
        image2 := Image fromUser.
        imageView := ImageView new.
        imageView model: self       “Create dependency of the subview class on the model.”
```

The action method for *Image 1* assigns image1 as the current image and triggers the change mechanism:

```
image1
"Assign image1 as current image and send change notification to dependent (subview)."
        image := image1.
        self changed
```

and image2 is similar. The remaining methods in TwoImages are only accessing messages for image and imageView.

Class ImageView

Class ImageView implements the subview. As in all situations involving the view holder widget, the superclass of ImageView will be View because it implements the damage repair mechanism.

When a view is being built by the builder, a new instance is created and in the process, message defaultControllerClass is sent. Our view does not allow any user interaction and we will thus return the NoController class - a controller that ignores all input events:

```
defaultControllerClass
        ^NoController
```

Note that a controller is required even if it is not active.

When a change occurs in the model (TwoImages), the model sends itself the changed message which then sends an update message to its view holder dependent. In response to this, the view holder must redraw itself, usually by sending itself the invalidate message which travels up the component tree of the window and causes the window to send a display message with its graphics context clipped to the damage area down the tree again. The definition of update is thus simply

```
update
"Trigger a redisplay."
        self invalidate
```

but since this behavior is inherited, we don't need this method at all.

The only remaining method (and the method that does all the work) is displayOn: which redisplays the contents of the view holder whenever it is required (during initial display, during each model change, and during invalidation). To display itself, our subview must ask its model for the current image and if the image is not nil, ask the graphics context to display it. Since we have not initialized image, its value is initially nil. In this case, we will display a help text in the center of the subview. To get the proper location for the text, we will ask the subview for its bounding rectangle, ask the graphics context to measure the width of the text, and display the text at an appropriate place. The whole definition is as follows:

```
displayOn: aGraphicsContext
"Display yourself on the graphics context supplied by the window."
| string width center |
        model image isNil
                ifTrue: “Display help text.”
                        [center := self bounds center.
                        string := 'Click action button'.
                        width := aGraphicsContext widthOfString: string.
                        aGraphicsContext displayString at: (center x ( width/2)) @ 30).
                        string := 'to display an image.'.
```

```
                width := aGraphicsContext widthOfString: string.
                aGraphicsContext displayString at: (center x ( width/2)) @ 50)]
        ifFalse: "Display currently selected image."
                [aGraphicsContext displayImage: image at: 0@0]
```

The program is now fully functional. Note that unlike our previous programs it automatically repairs window damage due to collapsing, reshaping, and other window events.

Improvements

We know that it is better to centralize shared behavior and in our case both image methods share the changed message which is always executed when image changes. In such a case, image should be changed by an accessing message which sends the changed message. As a consequence, image1 should be

```
image1
        self image: image1
```

and the corresponding accessing method should be

```
image: anImage
        image := anImage.
        self changed
```

We conclude that the process of creating a subview and inserting it into a canvas is quite simple and can be summarized into the following three steps:

1. Paint the *canvas* with a view holder widget, specifying the name of the application model method that returns the corresponding view object as its *View* property.
2. In the *application model*,
   a. create an instance variable to hold the view displayed in the view holder, and the corresponding accessing method specified as the *View* property in Step 1
   b. initialize the view holder instance variable to an instance of the subview class and specify the application model (self) as the view's model (needed only if you use dependency)
   c. when using dependency, every method that changes the model and affects the view must send changed to self. This is best implemented by busing accessing messages and including changed in them. When not using dependency, send self invalidate to the view when a change of the subview is desired. You can also redraw parts of the view selectively using invalidateRectangle: or invalidateRectangle:repairNow:.
3. In the definition of the subview class
   a. specify View as the superclass
   b. define a controllerClass method to return the class of the desired controller
   c. define an update method if you use dependency to control the subview; if updating consists of sending invalidate and nothing else, this behavior is inherited and update: is not required.
   d. define a displayOn: method (always); it generally obtains data from the model (the application) and uses it to redraw itself

As another illustration of the procedure, we will now do an example that shows how to display geometric shapes. The problem will give us an opportunity to see that geometric objects are not displayable in their raw form and require additional care.

Example 2: Displaying geometric figures

Problem: Implement an application with the user interface in Figure 12.11. When the user selects a radio button and clicks *Draw*, the window displays the corresponding geometric shape.

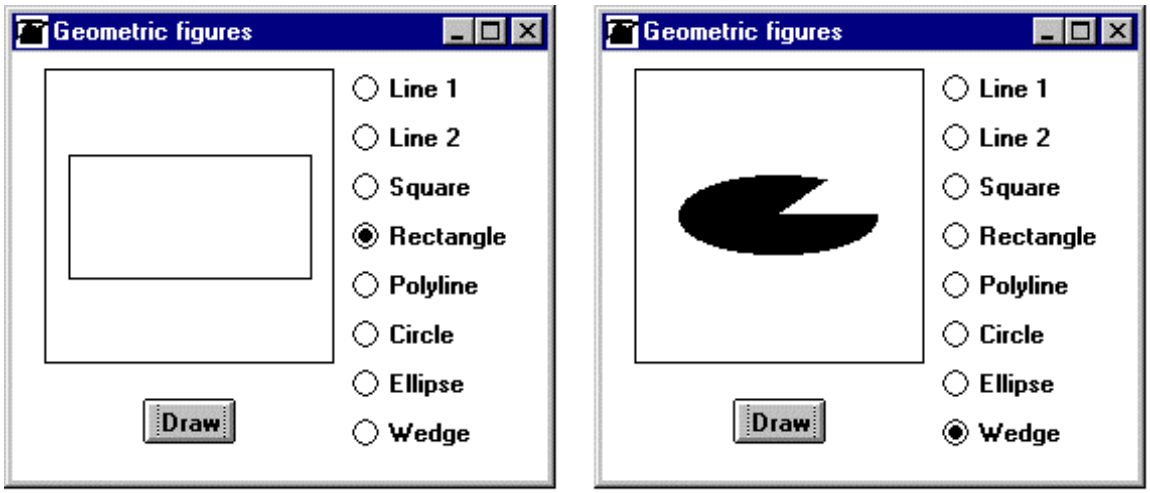

Figure 12.11. Example 2: Display of stroked rectangle (left), display of filled wedge (right).

Solution: We will use a subview with NoController to implement the drawing area. The application model (class GeometricFigures) keeps reference to the view in instance variable drawingView, and aspect variable object holds the current radio button selection. The draw action method will use dependency and send changed to the model The corresponding update method will only invalidate the view so we don't have to define it. update will trigger displayOn: which will ask the application model for the selected geometric object (derived from the aspect value of the selected radio button), and display it. Implementation of the corresponding methods is as follows:

Initialization of GeometricFigures

```
initialize
"Select the top radio button, create a binding to the subview, and define yourself as the subview's model."
        object := #line1 asValue.    "Initial selection when the window opens."
        drawingView := GeometricView new.
        drawingView model: self
```

The action method of the *Draw* button simply triggers the dependency-based update mechanism:

```
draw
"To redraw the subview, trigger the dependency mechanism."
        self changed
```

The display:at: method triggered by invalidate is defined in GeometricView as follows:

```
displayOn: aGraphicsContext
"Obtain currently selected geometric object and display it in the center of the view."
        aGraphicsContext display: model currentObject at: self bounds center
```

The general-purpose display:at: method in GraphicsContext can display any displayable object. In our earlier example, we used shape-specific messages such as displayLineFrom:to: but this strategy would be very awkward in this case. The currentObject message is defined in GeometricFigures and uses the radio button selection to get the desired object as follows:

```
currentObject
"Use currently selected radio button value to obtain the corresponding geometric object."
        ^self perform: object value
```

If the selected button is, for example, *Rectangle* (aspect object == #rectangle), the last line executes

```
self rectangle
```

and to complete our implementation, we must define the methods that calculate and return the appropriate displayable geometric objects. A very important feature of all of them is that display:at: cannot display geometric objects in their raw state and the object must first be 'wrapped' by conversion messages asStroker (for empty figures such as the rectangle on the left of Figure 12.11) or asFiller (for filled figures such as the wedge on the right of Figure 12.11). As an example, the method creating the wrapped rectangle is

```
rectangle
"Return a displayable rectangular border. The coordinates are calculated on the premise that the view displays the object in the center."
        ^(Rectangle origin: -60 @ -30 corner: 60 @ 30) asStroker
```

and the method creating the wrapped wedge is

```
wedge
"Return a displayable rectangle. The coordinates are calculated on the premise fact that the view displays the object in the center."
        ^(EllipticalArc
                boundingBox: (-50 @ -20 corner: 50 @ 20)
                startAngle: 0.0
                sweepAngle: 300.0) asFiller
```

We leave the remaining methods as an exercise.

Main lessons learned:

- Custom components can be created with the view holder widget which contains a subview with a controller.
- The only required property of a view holder is *View*. Its value is the name of the method that returns the subview's view object.
- In order to inherit the damage repair mechanism, the subview should be a subclass of View.
- The view class must define the displayOn: message which is sent automatically whenever the window needs to draw or redraw the subview. Also required is message defaultControllerClass.
- If response to changes of the view's model uses dependency, the view must also define an update method. If the only required action is invalidate, the behavior is inherited and update: is not needed.

Exercises

1. Does View define a default controller class? If so, what consequences does it have for view design?
2. Describe in detail what the asStroker and asFiller messages do. Inspect the result of sending such a message to a rectangle.
3. Write a description of wrappers.
4. Reimplement Example 1 using direct invalidation instead of dependency.
5. Extend Example 2 by adding a group of radio buttons labeled *stroked* and *filled* that allow the user to select how the geometric object should be displayed.
6. Which class defines the update message inherited by views?
7. Record a complete trace of an update of a view.

12.7 Controllers

Controllers are responsible for managing user input and the class library contains many controller classes for various kinds of widgets and tools. Only a few, however, are of interest to most programmers. The essence of the controller class hierarchy (with controllers commonly needed in applications shown in bold) is as follows:

```
Object ()
        Controller ('model' 'view' 'sensor')
                ControllerWithMenu
                        ModalController
                        ParagraphEditor
                                TextEditorController
                        SequenceController
                LauncherController
                MenuItemController
                NoController
                ScrollbarController
                StandardSystemController
```

The Controller class on the top defines all essential instance variables and behaviors and is often used as the direct superclass of custom controllers. Its instance variables include model and view which are responsible for communication with the other two components of the MVC triad, and sensor - an instance of a concrete subclass of the abstract class InputSensor which handles mouse and keyboard input and can provide information on such parameters as cursor coordinates.

In the past, all VisualWorks controllers used *polling*, a technique which constantly tests whether the user performed an input event, and takes the appropriate action if necessary. Polling requires extra processing time and this may cause some of several input events issued in quick succession to be missed. The preferred modern implementation of controllers is thus *event-driven* where VisualWorks is immediately notified by the operating system when an input event takes place, and sends this information to a controller in the form of an appropriate predefined notification message[2]. These notification messages are defined in the events protocol in class Controller and include entryEvent: and exitEvent: (automatically sent to the controller when the cursor enters or leaves the area of the controller's view), doubleClickEvent:, redButtonPressedEvent:, keyPressedEvent:, and many others.

All event messages have a single argument called event - an instance of an appropriate Event subclass such as CloseEvent, KeyPressedEvent, and MouseMovedEvent. The Event object contains information relevant for the event that triggers it such as the x and y coordinates of the current position of the hot spot of the cursor or the activated key. All of these predefined messages are *stumps* containing only an empty body, and concrete subclasses redefine them to perform whatever actions are necessary when the corresponding event occurs. In other words, event messages are hooks. We will see examples of them shortly.

Besides Controller, the most important controller classes are

- StandardSystemController - in charge of window interaction, in particular its <window> pop up menu
- ControllerWithMenu - has a menuHolder for the <operate> menu, and a performer (the object that performs menu commands)
- TextEditorController - used by the text editor widget
- ParagraphEditor- superclass of TextEditorController defining most of its functionality
- NoController - controller that does not respond to any user input

In this chapter, we are interested in controllers in the context of creating new UI components. We have already seen that this is done by using the view holder widget and defining a view-controller pair for it. In the previous section, we used NoController because our subviews did not allow any user interaction.

[2] The actual implementation of events in VisualWorks is still based on polling but the outward appearance is event-driven.

In the following example, we will create a UI component with an active user interface and show how to create an active controller that responds to user input.

Example: Subview with clickable hot spots

Problem: Implement an application that displays a bordered subview (Figure 12.12) equipped with an <operate> menu and capable of responding to mouse clicks. When the subview is empty, the menu contains only the *Add hot spot* command. When the view contains at least one hot spot, the menu also displays a *Remove hot spot* command.

When the user selects *Add hot spot*, the program requests a string (hot spot name) and the cursor changes to cross hair. (The cursor changes back to the original shape whenever it leaves the subview area and changes back to cross hair when it re-enters the subview.) Clicking the left mouse button inside the subview creates a 'hot spot' and displays it as a small red rectangle. The cursor then changes back to its original shape.

When the user clicks *Remove hot spot*, the cursor changes to cross hair and clicking the left mouse button over a hot spot displays a warning with the name of the hot spot and deletes the hot spot from the subview. (During this procedure, the cursor again changes to the original shape whenever it leaves the subview area.) After redisplaying the view, the cursor changes back to its original shape.

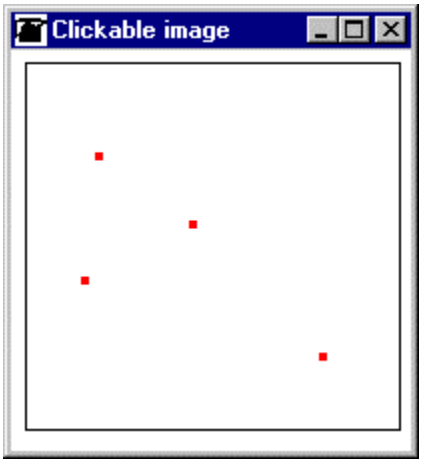

Figure 12.12. Example: Clickable subview with four hot spots.

Solution: We will need three classes - the application model (ClickableImage), the subview class (ClickableView), and its controller (ClickableController).

Class ClickableController

ClickableController will be a subclass of ControllerWithMenu because the specification requires an <operate> menu. It will define menu building methods, and event behaviors for subview entry and exit and for red button press.

To implement the desired cursor behavior, ClickableController must have an instance variable to hold the shape of the cursor before it changed to cross hair so that we can restore it upon exit from the subview or when a hot spot related operation ends. We will call this variable cursor. Since cursor is undefined before the first user action, the exitEvent: method will access it via the following lazy accessing method:

```
cursor
"Lazy initialization of cursor."
^cursor isNil
        ifTrue: [Cursor current cursor]        "Get current cursor shape from the system."
```

```
        ifFalse: [cursor]
```

The enterEvent: message is sent when the cursor enters the subview. It must ask the application model which command is currently executing (we will call this method state), and display the cross hair cursor if the command is not nil:

```
enterEvent: event
"If the model is executing the Add or the Remove command, change cursor to cross hair."
model state isNil
        ifFalse: [cursor := Cursor currentCursor.
                Cursor crossHair show]
```

Since this method is defined in the subview's controller, it will only be executed when the controller is in charge and this happens only when the cursor is in the subview's area. The exitEvent: method resets the cursor if state is not nil. Note that we would get the same result if we did not check state at all:

```
exitEvent: event
"If the model is executing Add or Remove, reset cursor on exit from subview."
model state isNil
        ifFalse: [cursor show]
```

The state method in ClickableImage returns #addHotSpot, #removeHotSpot, or nil, depending on the command currently being executed.

The handler of the red button click event (again executed only if the cursor is within the subview) asks the model to execute the operation corresponding to the current state (add/remove hot spot or do nothing), sending the coordinates of the cursor as the argument:

```
redButtonPressedEvent: event
"Send message with cursor coordinates to model."
|position|
        position := self sensor cursorPointFor: event. "Extract position from event."
        model executeOperationWith: position                "Result depends on the state of the model."
```

The next task of the controller is to implement the menus. Since the menu depends on the state of the application, we will leave it to the application to construct the menu as necessary. We are thus finished with the controller!

Class ClickableView

Our next class is ClickableView, naturally a subclass of View. From the previous section, we know that its responsibilities include accessing its controller class, responding to update (we will again use dependency), and responding to displayOn:. The controller is ClickableController, hence

```
defaultControllerClass
        ^ClickableController
```

There is no need for update since it only invalidates and this is an inherited behavior. And the display message displays all hot spots held by the model as little red squares:

```
displayOn: aGraphicsContext
"Display all hot spots as red squares centered at hot spot coordinates."
        aGraphicsContext paint: ColorValue red.
        model hotSpots do: [:hotSpot|
                aGraphicsContext displayRectangle: (hotSpot at: 1) -2 extent: 4 @ ]
```

Class ClickableImage

The last undefined class is ClickableImage - the application model. Its instance variables will be clickableView (for accessing the view displayed in the subview - the view holder's *View* property), hotSpots (a collection of two-element arrays containing hot spot coordinates and names), and state. We will also find useful two extra variables: hotSpotName will hold the name of a new hot spot, and controller will provide direct access to the controller for menu control - although the controller could be accessed via imageView, the need is frequent and 'caching' the controller in a variable is better.

Variable state requires closer attention. As we noted above, *Add* and *Remove* actions are triggered by message executeOperationWith: from the controller. The easy way to execute a message that depends on aSymbol is perform: aSymbol. Since the operation requires cursor coordinates, the perform message must be a keyword message with cursor coordinates. To satisfy the different possible states, we will use the following three values of state: nil (not executing a menu command), #addHotSpotAt:, and #removeHotSpotAt:, and define methods corresponding to the last two values.

We are now ready to implement ClickableImage. Initialization involves setting up instance variables and telling the controller that the performer of the menu messages (the executor of menu commands) will be the model. We must also initialize the menu:

```
initialize
        hotSpots := OrderedCollection new.
        imageView := ClickableView new.
        imageView model: self.
        controller := imageView controller.    "Construct and assign the controller."
        controller performer: self.            "The performer of menu commands is the application model."
        controller menuHolder value: self menuWithAdd        "The initial value of <operate> menu."
```

The following two methods construct the <operate> menu when there are no hot spots and the menu for a view with at least one hot spot. The definitions are as follows:

```
menuWithAdd
"Menu when no hot spots exist."
| menuHolder |
        menuHolder := MenuHolder new.
        menuHolder add: 'add hot spot' -> #addHotSpot.
        ^menuHolder menu
```

and

```
menuWithRemove
"Menu when at least one hot spot exists."
| menuHolder |
        menuHolder := MenuHolder new.
        menuHolder add: 'add hot spot' -> #addHotSpot.
        menuHolder add: remove hot spot' -> #removeHotSpot.
        ^menuHolder menu
```

When the user clicks *add hot spot* or *delete hot spot* in, the controller sends addHotSpot or removeHotSpot to its menu performer - the ClickableImage object and our next step is to define these two methods. Method addHotSpot asks the user for the name of the new hot spot and changes state to #addHotSpotAt:. Control then passes into the hands of the controller (defined above) which tracks cursor entry and exit into the subview and sends executeOperationWith: to ClickableImage when the user clicks the red mouse button in the subview. This analysis leads to the following definition:

```
addHotSpot
"Start 'add hot spot' operation - get hot spot name and change state; leave the rest to the controller."
        hotSpotName := Dialog request: 'Enter hot spot name.'.
        state := #addHotSpot        "Name of method to be executed on click button event."
```

Method removeHotSpot only changes the state:

```
removeHotSpot
"Start remove hot ' operation - change state; leave the rest to the controller."
        state := #removeHotSpot    "Name of method to be executed on click button event."
```

After the execution of any of these two methods, the application waits for the user to move the cursor or click the mouse button which then sends the appropriate mouse event to the controller. This triggers redButtonPressedEvent: defined above, which sends executeOperationWith: aPoint. As we already explained, the method simply performs the state symbol with a Point argument provided by the controller:

```
executeOperationWith: aPoint
"Execute state operation with point supplied by controller. Ignore if not in 'add' or 'remove' state."
        state isNil             "If nil, we are not in executing a menu command – no action."
                ifFalse: [self perform: state with: aPoint.
                        self changed]
```

The real work of adding and removing hot spots is performed by addHotSpotAt: and RemoveHotSpotAt: and by the dependency mechanism invoked by changed. Method addHotSpotAt: resets the cursor to what it was before it changed to cross hair. It then adds the new hot spot to the collection, makes sure that the <operate> menu now includes *remove* because we now have at least one hot spot, and resets state to nil (end of add state):

```
addHotSpotAt: aPoint
"Add hot spot to collection, reset state and cursor, update menu."
        controller cursor show.
        hotSpots add: (Array with: aPoint with: hotSpotName).
        controller menuHolder value: self menuWithRemove.
        state := nil
```

Method removeHotSpotAt: is slightly more complicated because it must check whether the clicked point lies within one of the red squares:

```
removeHotSpotAt: aPoint
"Check if aPoint corresponds to a hot spot and if it does, inform user, display name, remove hot spot from
collection, reset state and cursor, and update menu if necessary."
| hotSpot |
        controller cursor show.
        (hotSpot := self hotSpotAt: aPoint) isNil        "Return hot spot under cursor or nil."
                ifTrue: [^Dialog warn: 'Not a hot spot'].
        "Display name of hot spot being removed."
        Dialog warn: 'Hot spot to be removed: ', (hotSpotAt: 2).
        hotSpots remove: hotSpot.
        hotSpots isEmpty                                "Change menu if there are no more hot spots."
                ifTrue: [controller menuHolder value: self menuWithAdd.
        state := nil
```

The only remaining method is hotSpotAt: aPoint which checks whether aPoint corresponds to a hot spot. It returns a hot spot corresponding to the position if the cursor is in a small area surrounding the hot spot, and nil otherwise:

```
hotSpotAt: aPoint
"Is aPoint within 2 pixels from a hot spot center? If so, return the hot spot."
        ^hotSpots detect: [:hotSpot | (aPoint - (hotSpot at: 1)) abs <= (2 @ 2)]
                        ifNil: [nil]
```

Note the use of point arithmetic and comparison.

Main lessons learned:

- Although VisualWorks provides polling as one of the ways to program controllers, event driven controllers are now preferred.
- Every view must have a controller, even if it does not allow user interaction.
- Custom controllers are usually subclasses of Controller or ControllerWithMenu.
- Class Controller defines many stump event methods that are triggered by the operating system when a input event occurs. When defining a custom controller, redefine those events methods that must be handled by the subview.
- All event methods have an event argument which contains all necessary information about the event that has occurred.
- Class ControllerWithMenu adds a menu holder for the <operate> menu, and a reference to the performer of the menu commands. The performer is often the application model but the default is the controller itself.
- Class ParagraphEditor and its subclass TextEditorController add text editing and code execution menu and its support.

Exercises

1. Reimplement our Example without lazy accessing and evaluate both approaches.
2. Reimplement the hot spot example by replacing the array representation of hot spots with a class called HotSpot and consisting of a name and a rectangle with the appropriate dimensions.
3. Draw a scenario diagram showing what happens when the user clicks *add* or *remove* in our example.
4. Explore classes Controller, ControllerWithMenu, StandardSystemController, and NoController and write a description of their essential behaviors.
5. It is not always necessary to specify the performer of a menu because performer has a default. What is this default and what does this definition imply?
6. Explore classes ParagraphEditor and TextEditorController and write a description of their essential behaviors.
7. Explore event methods and event classes and write a description of their essential behaviors.
8. Method displayOn: in our example redraws the whole subview which is not necessary. If we just added a new hot spot or deleted an existing one, we only need to redraw the corresponding small part of the subview and this can be achieved by using invalidateRectangle: instead of invalidate. Modify the program to use this approach and comment on the result.
9. Add command *change background color* to the <window> menu. The command will display a multiple choice dialog listing all predefined colors and when the user selects one, it will change the color of the currently active window. (Notes: You must reinitialize class StandardSystemController after changing the menu - why? After changing the color, send display to the window for the change to take effect.)
10. The concept of a clickable view has uses beyond our simple example. List several applications that could use mouse sensitive areas and design a general purpose ClickableController suitable for as many uses as possible.
11. Develop an application for drawing curves defined by mathematical formulas. We leave the detailed specification to you but the general principles should be as follows: The user interface allows the user to enter a function as a one-argument block, the start and end points of the argument, and the color in which the curve is drawn. Several curves may be drawn in the window at the same time and on the same scale. The window is self-repairing.
12. The previous exercise suggests the idea of a curve-drawing widget, along the lines of the Calendar widget described in Appendix 2. Write the specification and develop such a widget. Test your widget by reimplementing the previous exercise.
13. Reimplement the Calendar widget using a subview.
14. The TextCollector class which is the model of the Transcript is a very interesting counterpart of the Text Editor because it maintains focus at the end of its text contents: All output to it is automatically appended at the end. This is very useful in applications such as system logs or logs of network activity

which must be maintained in the order in which they are generated. Unlike the Text Editor, however the TextCollector does not have its own button on the UI Palette and if you wish to use it, you must use other means. Give a detailed description of how you could incorporate Transcript-like behavior in an application window.

Conclusion

VisualWorks user interfaces are based on three kinds of objects: display surfaces, the objects being displayed, and the object doing the drawing and holding the drawing parameters.

Display surfaces can be divided into those dedicated to displaying on the screen and those dedicated to printing. Screen related display surfaces include windows, masks, and pixmaps. Among the several kinds of windows, ApplicationWindow is most important because it implements canvases drawn with the UI Painter. Pixmaps are used to construct displayable objects in memory before they are displayed on the screen, and masks are used to control which part of an underlying image is displayed and which is hidden or transparent.

Programming of a window is usually restricted to controlling window parameters such as window label or background paint. The programming of a pixmap usually consists of displaying an image on it and masking it with a mask. Sometimes, a pixamp is used just to construct an image, for example as a composition of geometric objects. The programming of a mask usually consists of displaying a shape object that defines the mask's transparent area.

The drawing of visual objects is performed by a GraphicsContext. This crucial class defines methods for displaying objects such as images, text, and geometric objects, and holds display parameters such as line width, text font, color, and clipping rectangle which defines the part of the window or subview used for display. All of these parameters can be modified programmatically at run time. GraphicsContext is essential for the design of custom GUIs because it implements all display.

Visual objects that can be drawn by a GraphicsContext on a display surface include strings, geometric objects, and images. Geometric objects can be drawn either as anonymous objects using display:at: or by object-specific messages such as displayRectangle:at:. In the first case, the object must first be wrapped as a stroker or as a filler. Object-specific methods such as displayRectangle:at: include this operation.

Although most applications of paint use instances of ColorValue and paint the surface or display the object in a single uniform color, paint can also be an instance of a Pattern. A Pattern is essentially an image used as a tile to wallpaper a rectangular area on a display surface.

Images are pixel-by-pixel representations of rectangular areas. Their public protocol is defined in the abstract class Image which transparently delegates work to its concrete subclasses that implement images for specific image depth. (Image depth is the number of bits used to represent the color of one pixel.)

Another object useful in GUIs is the cursor representing the position of the mouse on the screen. Class Cursor contains a variety of predefined cursor shapes, and new ones can be created by specifying cursor image, mask (defines the see-through area), hot spot (the Point returned when the cursor is queried for its position), and name. A convenient way to create a cursor shape is by using the Image Editor tool which contains a special command for saving an image as a mask.

To create a new GUI component such as a chess board or the face of a clock, select the subview widget (view holder) from the UI Palette, paste it on the canvas, and define at least its *View* property - the method that returns the view object displayed in the subview. As the next step, construct your view class, defining at least method displayOn: aGraphicsContext which is the basis of the display of the subview. The method displays on a GraphicsContext supplied by the application window, with its default parameters which can be changed in displayOn:. If you use dependency, you must also sometimes define an update method. In most cases, update sends invalidate to self and this behavior is defined in a superclass of View and inherited. If your update is more complex, for example requiring invalidateRectangle: you must define update in your view class. Invalidate messeages travel up the component tree to the window which then calculates the clipping area of its GraphicsContext and sends displayOn: aGraphicsContext down the tree to its components with the clipped GraphicsContext. Each component then redraws that part of itself which

falls within the clipping rectangle, using the supplied default graphics context parameters or redefining them temporarily.

Older versions of VisualWorks used polling controllers which repeatedly queried the sensor for UI events such as mouse movement or button clicks. Since polling requires extra overhead and since UI events occur fast, this approach often missed UI events. Modern implementations still provide polling controllers but add event-driven controllers which obtain event notification directly from the operating system, eliminate the polling overhead, and thus provide better performance. All you have to do to create a custom event-driven controller is create a new controller class subclassed to ControllerWithMenu (if you require an <operate> menu) or Controller (if you don't need an <operate> menu) and redefine those of its event tracking methods that are important for your subview. If your subview does not have an active user interface, specify NoController as the default controller class.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

ApplicationWindow, **Controller**, **ControllerWithMenu**, CoverageValue, *Cursor*, **GraphicsContext**, *Mask*, **NoController**, Paint , *Pattern*, *Pixmap*, **TextCollector**.

Terms introduced in this chapter

controller - part of GUI responsible for user interaction via mouse and keyboard events
clipping rectangle - rectangle restricting display area
cursor hot spot - Point reported by cursor when it is queried about its position
damage repair - repainting of surfaces damaged by closing or overlaying a window
display surface - screen or printer page on which painting is taking place; memory representation in the case of mask
event-driven controller - controller whose UI event messages are triggered by the operating system (see also *polling controller*)
filler - filled geometric object (see also *stroker*)
graphics context - object that holds display parameters and performs the display
mask - shape overlaid over display area; determines which part of area is visible or transparent
image depth - number of bits used to represent one pixel
invalidation - the process of notifying a window that a view or its part should be redisplayed; the propagation of this request to the components
pattern – a form of paint, a tile that can be used to wallpaper a rectangular area
paint - a color used to paint a surface or a pattern used to tile it
pixel – picture element, an addressable dot on the screen
polling controller - a controller that repeatedly asks its event sensor for UI events and responds appropriately; outdated (see *event-driven controller*)
stroker - contoured geometric object (see also *filler*)
subview - contents of a *view holder*
view - object responsible for displaying a predefined or custom GUI component
view holder - widget used to create custom visual components