Last modified on Sun Mar 8 1:01:58 PST 1987 by luca

VBT Toolkit

Luca Cardelli

1.Introduction

The VBT toolkit is a set of modules and programs for building certain classes of user interfaces. The general approach is to provide a set of ready-to-use interaction primitives, generically called "interactors" (buttons, menus, text areas, etc.). Interactors are the basic building blocks for user interfaces.

One way of assembling basic interactors into more complex objects is provided at the VBT level through the notion of "split", which provides for splitting one VBT into sub-VBTs, each containing other splits or VBTs (e.g. interactors). Another way of composing interactors is provided by the toolkit through the notion of "dialog boxes" (or "dialogs"), which present a standard simplified interface to client programs.

To put this into context, let me contrast the toolkit approach with other approaches. The obvious one is to let users build their own interfaces from scratch, using the bare VBT interface. This has many disadvantages, both in terms of discouraging people from building user interfaces, wasting people's time, generating bad interfaces and producing inconsistent interaction styles. Some of this will still be required, because the toolkit cannot cover all possible applications, but the hope is that specialized interactors will be created only when really needed, and they will be combined with the standard ones whenever possible.

Another approach is to provide high-level tools (e.g. libraries, languages or editors) for specifying the appearance and behavior of interactors (the VBT interface is a low-level example of this). This gives more flexibility and coherence in building libraries of interactors, but is very hard to do in a clean and effective way; note that ordinary sequential languages are ill suited, because of the complex non-deterministic and concurrent activities present in user interfaces. The approach taken by the VBT toolkit is to provide a fixed set of basic interactors known to be useful for most interfaces, and to structure the environment so that new interactors can be easily added. For example, dialog boxes do not care (or know) what kind of interactors they contain.

Finally, one could have a "user interface specification language" of some kind. This is extremely ambitious and I don't know whether it could be done. In any case, languages tend to be very messy when trying to specify two-dimensional information. Hence I believe that user interfaces should not be described by languages, but instead built by editors. An integral part of the toolkit is a "dialog editor" for assembling dialogs. At a lower level, one could imagine an "interactor editor" for assembling interactors; such things have been done, but are harder.

This document attempts to provide general information and background about the toolkit. It must be complemented by reading the comments in the def files of the various interfaces.

2. Interactors

An interactor is a "user interface primitive"; something which is seen as a

functional unit. It can be used to present some internal state, to operate a mechanism, or to modify a structure. Normally, an interactor has a well defined geometrical extent. Some interactors are not quite primitive (several have embedded scroll bars, which are also interactors). This is admissible when there is a very tight coupling between the embedded interactor and the rest of the parent interactor, otherwise separate interactors should be used.

Interactors have a given "reactivity" at any point in time, i.e. they can be functioning at various levels of activation or sleepiness. Here are the standard meanings for the reactivities currently in use:

Active:	Functions normally.
ReadOnly:	Functions but cannot be modified (to some degree).
Protected:	Needs special action to become temporarily active.
Passive:	Does not function, but looks normal.
Dormant:	Does not function and looks "dim".

Interactors have one or more "looks" specifying their appearance. There is a common "looks" data structure used by all interactors.

Tookit interactors have two interfaces, in a sense. One is a standard Modula2+ interface, providing routines to create interactors, activate them, set and inspect their properties and appearance. This interface is suitable for using interactors in isolation, independently of the rest of the toolkit, and is not described here in detail; see the def files.

The second interface is intended for interactors which are embedded in dialog boxes. It is based on the multiple-inheritance subclassing provided by VBTs. All interactors must be subclasses of the "Interactor" class (see InteractorVBT.def), which is defined as any VBT which provides the following four methods: (1) GetProperty: get the current value of a property of an interactor; (2) SetProperty: set the value of a property of an interactor; (3) GetDescription: produce a standard data structure which completely describes an interactor; (4) SetDescription: make an interactor conform to a given description. Moreover, two routines for creating and deleting interactors of a given class must be provided; such routines have the same type for all interactor classes.

To implement a new interactor class, one can take any existing VBT performing some function and supply the above four methods and two procedures. This will allow the new interactor to be inserted in dialog boxes and to be manipulated by the dialog editor.

The data structure returned by GetDescription (called a "description") is suitable to be pickled; it is a list of name-value pairs, where the name is a Text.T and the value is a REFANY.

One detail about the Modula2+ interface. Client procedures are supplied to interactors to be called when something interesting (called an "event") happens to that interactor. Since user interaction is event-based, much interactor activity is achieved by passing procedures around. Modula2+ only supports this in a restricted way, so you should be familiar with standard "closure" techniques; here is a small tutorial.

A "first-class" procedure is a procedure which can be passed as an argument or returned as a result of another procedure, in and out of arbitrary scoping levels. Such procedures may refer to non-local variables. Statically scoped languages (such as the Algol family and Scheme, unlike the Lisp family and APL) define the meaning of such non-local variables to be determined by the procedure definition scope (i.e. statically), not by the procedure invocation scope (i.e. dynamically). In Modula2+, a first-class procedure can be simulated by a top-level procedure which takes a record as a parameter (passed as a REFANY) defining the values of what would be its non-local variables in inner scopes. Such record is called a "closure", because it closes the procedure with respect to its non-local variables.

The rest of this section describes the interactors currently available.

2.1 Passive Areas

A passive area constantly show one "looks". Passive areas are mostly used to provide backgrounds and borders, for text labels, and for bitmap icons. As interactors, they have the following description:

```
"Kind" = "PassiveArea"
"Reactivity" = "Active" or "Dormant"
"NormalLooks" : IconLooks.T
"DormantLooks" : IconLooks.T
```

Where "Kind" is a property of all interactors; a constant string describing the kind of interactor.

2.2 Trill Buttons

A trill button generates events on mouse down-transitions, and then generates more events (auto-repeat) while the mouse cursor is on top of the button, up to the corresponding up-transition. As an interactor, it has the following description:

```
"Kind" = "TrillButton"
"Event" : Text.T
"Reactivity" = "Active" or "Dormant"
"NormalLooks" : IconLooks.T
"ExcitedLooks" : IconLooks.T
"DormantLooks" : IconLooks.T
```

The excited looks are shown while the button is generating events. The "Event" string is passed to the dialog box containing this interactor, whenever an event is generated.

2.3 Trigger Buttons

Trigger buttons are stateless buttons which generate events on mouse transitions. There are both down-triggered and up-triggered flavors. If "protected", one has to click twice to activate them. As interactors, they have the following description:

```
"Kind" = "TriggerButton"
"Event" : Text.T
"Transition" = "Up" or "Down"
"Reactivity" = "Active", "Dormant" or "Protected"
"NormalLooks" : IconLooks.T
"ExcitedLooks" : IconLooks.T
"DormantLooks" : IconLooks.T
"ProtectedLooks" : IconLooks.T
```

2.4 Radio Buttons

An individual radio button has one bit of state; when used in isolation it is also called an on/off button, since it can be put into an "on" or "off" state, and toggled by mouse clicks. Events are generated on state changes.

Radio buttons can be logically "grouped" (while remaining physically separated). If there are n buttons in the same group, only one of them can be "on", (they can all be "off"). Clicking one button in a group will switch off the other button (if any) in the "on" state. There is no way to set them all off by using the mouse.

An individual radio button is an interactor with the following description:

```
"Kind" = "RadioButton"
"Group" : Text.T                (shared by the button in the same group)
"Event" : Text.T
"StateOnName" : Text.T
"Transition" = "Up" or "Down"
"Reactivity" = "Active", "Dormant" or "Protected"
"OffLooks" : IconLooks.T
"ExcitedOffLooks" : IconLooks.T
"OnLooks" : IconLooks.T
"ExcitedOnLooks" : IconLooks.T
"DormantLooks" : IconLooks.T
"ProtectedLooks" : IconLooks.T
```

A radio button group is also an interactor, but it is never visible. It maintains the global state of the group, and has the following description:

```
"Kind" = "RadioButtonGroup"
"Group" = "Group"               (meaning this is a group interactor)
"State" : Text.T
"StateOffName" : Text.T
```

where "State" is equal to "StateOffName", or to the "StateOnName" of one of the radio buttons.

2.5 PullDown Menus

A pulldown menu is a button which pops up a list of selections when the mouse is pressed on it. As an interactor, it has the description:

```
"Kind" = "PullDown"
"State" : Text.T
"Reactivity" = "Active" or "Dormant"
"NormalLooks" : Text.T
"ExcitedLooks" : Text.T
"DormantLooks" : Text.T
```

where "State" is a list of menu items terminated by '\n', each with format: "event '\t' looks", where "event" is the event when that item is selected, and "looks" is the string which appears in the menu. The other properties apply to the pulldown button, not the menu items.

Pulldowns can be "grouped". Menus in the same group can be pulled down by rolling the mouse over them, after the first down-click on any one of them. A pulldown group interactor has the following description:

```
"Kind" = "PullDownGroup"
```

2.6 Scroll Bars

A scroll bar maintains four numeric quantities; a low and high bound, and a low and high range, in the relation lowBound <= lowRange <= highRange <= highBound. The relative size of the ranges with respect to the bounds determines the size of a visible cursor which the user can "thumb" along a bar. The user can also click on the bar on either side of the cursor to "page", click two buttons on either side of the bar to "scroll", and click two buttons on either side of the scroll buttons to move the cursor to the "top" and "bottom" of the admissible range. Although a standard interpretation has just been given for the admissible user actions, the events generated by those actions are uninterpreted, and can be defined to have different meaning. A scroll bar has the following description:

```
"Kind" = "ScrollBar"
"ToLowEvent" : Text.T
```

```
"StepDownEvent" : Text.T
"SlideDownEvent" : Text.T
"ThumbEvent" : Text.T
"SlideUpEvent" : Text.T
"StepUpEvent" : Text.T
"ToHighEvent" : Text.T
"Reactivity" = "Active", "Dormant" or "Passive"
"Axis" = "Hor" or "Ver"
"LowBound" : Text.T     (an integer)
"LowRange" : Text.T     (an integer)
"HighRange" : Text.T    (an integer)
"HighBound" : Text.T    (an integer)
"ScrollBgLooks" : IconLooks.T
"SliderSlotLooks" : IconLooks.T
"SliderThumbLooks" : IconLooks.T
"ScrollJumpOffLooks" : IconLooks.T
"ScrollJumpOnLooks" : IconLooks.T
"ScrollStepOffLooks" : IconLooks.T
"ScrollStepOnLooks" : IconLooks.T
```

where the "Jump" and "Step" looks describe the bitmaps used for the "jump to bottom" and "step down" buttons in their north-pointing orientation.

2.7 Browsers

A browser is used to select from a variable size (possibly very large) set of items. Only a subset of the items is visible, and a scroll bar is used to access the rest of them. In "Single Selection" mode, items are selected by clicking and possibly dragging; the up-transition determines the selected item. In "Multiple Selection" mode, items are toggled between the selected and unselected state by clicking on them or sweeping over them. An option-click "between" two items, inserts all the currently selected items between those items, rearranging the list. A browser has the following description:

```
"Kind" = "Browser"
"State" : Text.T
"SelectionMode" = "Single" or "Multiple"
"Reactivity" = "Active", "Dormant", "ReadOnly" or "Passive"
"TextLooks" : IconLooks.T
"ScrollBgLooks" : IconLooks.T;
"SliderSkidLooks" : IconLooks.T;
"SliderSlotLooks" : IconLooks.T;
"SliderThumbLooks" : IconLooks.T;
"ScrollStepOffLooks" : IconLooks.T;
"ScrollStepOnLooks" : IconLooks.T;
"ScrollJumpOffLooks" : IconLooks.T;
"ScrollJumpOnLooks" : IconLooks.T;
```

where "State is a list of menu items terminated by '\n', each with format: "['+' | '-'] looks '\t' upevent '\t' dnevent". If the first character is a '+' the item is selected; if it is a '-' the item is non-selected; if it is something else it is considered part of "looks" and the item is non-selected. "looks" is the string which appears in the browser, "upevent" is the event generated when that items is deselected, and "dnevent is the event generated when that item is selected.

The following property is defined for the GetProperty method:

```
"Selection" : Text.T
```

this is a list of looks fields, separated by '\n', one for each currently selected item (at most one in single selection mode).

2.8 Fatbits

A fatbits interactor maintains a magnified display of bits. The bits themselves are not maintained by fatbits, so that this interface can be used for textures, cursors, bitmaps, etc. Here is the description:

```
"Kind" = "Fatbits"
"HorSize" : Text.T      (a cardinal)
"VerSize" : Text.T      (a cardinal)
"ClickEvent" : Text.T
"DragEvent" : Text.T
"Reactivity" = "Active", "Dormant" or "Passive"
"BackgroundLooks" : IconLooks.T
"BgTintLooks" : IconLooks.T
"FgTintLooks" : IconLooks.T
```

where "ClickEvent" is generated when one fatbit is activated by clicking and "DragEvent" is generated when one fatbit is activated by dragging; the coordinate of the fatbits is passed as the "eventValue" (an InteractorVBT.PointRef) to the registered DialogVBT.EventProc's for those events.

The following properties are defined for InteractorVBT.SetProperty:

```
     "Bitmap" : InteractorVBT.RefBitmap
(paint a fatbits from a bitmap, using the fg and bg tints)
     "Fg" : InteractorVBT.RefPoint
(set a fatbits point to the foreground tint)
     "Bg" : InteractorVBT.RefPoint
(set a fatbits point to the background tint)
     "Paint" = "Fg" or "Bg"
(paint the entire fatbits in fg or bg tint)
     "Transform" = "TumbleW", "TumbleE", "TumbleN", "TumbleS",
                   "RotateCW", "RotateCCW", "FlipH" or "FlipV"
(perform a transformation)
```

2.9 Text Areas

A text area is an editable region of text. See a later section for a description of the user interface. As an interactor it has the following description:

```
"Kind" = "TextArea"
"Event" : Text.T
"State" : Text.T
"Reactivity" = "Active", "Dormant", "ReadOnly" or "Passive"
"TextLooks" : IconLooks.T
```

Where "Event" is generated (if non NIL) when the text is changed in any way, and "State" is the entire text in the text area.

2.10 Text Ports

A text port is a text area with a scroll bar. As an interactor it has the following description (see Text Areas and Scroll Bars):

```
"Kind" = "TextPort"
"Event" : Text.T                ("something changed" event)
"State" : Text.T                (the entire text in the text port)
"Reactivity" = "Active", "Dormant", "ReadOnly" or "Passive"
"TextLooks" : IconLooks.T
"ScrollBgLooks" : IconLooks.T;
"SliderSlotLooks" : IconLooks.T;
"SliderThumbLooks" : IconLooks.T;
"ScrollStepOffLooks" : IconLooks.T;
```

```
        "ScrollStepOnLooks" : IconLooks.T;
        "ScrollJumpOffLooks" : IconLooks.T;
        "ScrollJumpOnLooks" : IconLooks.T;
```

2.11 Text Lines

A text line is a text area showing a single line of text. When the text overflows, buttons for horizontal scrolling become visible. A text line intercepts carriage returns so that they are not inserted in the text, and are interpreted as "completion events" (meaning "do it"). As an interactor it has the following description:

```
        "Kind" = "TextLine"
        "Event" : Text.T                ("something changed" event)
        "CompletionEvent" : Text.T      (on carriage return)
        "State" : Text.T                (the entire text in the text line)
        "Reactivity" = "Active", "Dormant", "ReadOnly" or "Passive"
        "TextLooks" : IconLooks.T
        "LftArrowHeadLooks" : IconLooks.T
        "LftArrowTailLooks" : IconLooks.T
```

The "Arrow" looks are the bitmaps used for scroll buttons in their left-pointing versions.

2.12 Numeric Areas

A numeric area is a text line maintaining a numeric quantity between two bounds. The numeric value can be modified by editing, or by clicking "increment" and "decrement" buttons. Legal strings are numeric strings, "infinity", "+infinity" and "-infinity". Unrecognized strings are ignored, and the numeric value is unchanged. The strings are parsed and checked for legality and in-boundness when typing carriage return, or when clicking the increment and decrement buttons. This is the description:

```
        "Kind" = "NumericArea"
        "Event" : Text.T                ("something changed" event)
        "CompletionEvent" : Text.T      (on carriage return)
        "State" : Text.T                (the integer)
        "LowBound" : Text.T             (the lower integer bound)
        "HighBound" : Text.T            (the upper integer bound)
        "Reactivity" = "Active", "Dormant", "ReadOnly" or "Passive"
        "IncDecButtons" = "Yes" or "No" (increment/decrement butt. visible)
        "TextLooks" : IconLooks.T
        "PrimarySelectionLooks" : IconLooks.T
        "SecondarySelectionLooks" : IconLooks.T
        "LftArrowHeadLooks" : IconLooks.T
        "LftArrowTailLooks" : IconLooks.T
        "NumIncOffLooks" : IconLooks.T
        "NumIncOnLooks" : IconLooks.T
        "NumDecOffLooks" : IconLooks.T
        "NumDecOnLooks" : IconLooks.T
```

2.13 Transcripts

A text line (input) and a text port (output) packaged to provide a typescript with editable history. As an interactor it has the following description (see Text Areas and Scroll Bars):

```
        "Kind" = "Transcript"
        "Event" : Text.T                (input area modified)
        "CompletionEvent" : Text.T      (cr in the input area)
        "InState" : Text.T              (the entire text in the input area)
        "OutState" : Text.T             (the entire text in the output area)
        "Reactivity" = "Active", "Dormant", "ReadOnly" or "Passive"
```

```
"TextLooks" : IconLooks.T
"ScrollBgLooks" : IconLooks.T;
"SliderSkidLooks" : IconLooks.T;
"SliderSlotLooks" : IconLooks.T;
"SliderThumbLooks" : IconLooks.T;
"ScrollStepOffLooks" : IconLooks.T;
"ScrollStepOnLooks" : IconLooks.T;
"ScrollJumpOffLooks" : IconLooks.T;
"ScrollJumpOnLooks" : IconLooks.T;
"LftArrowHeadLooks" : IconLooks.T
"LftArrowTailLooks" : IconLooks.T
```

GetProperty recognizes the following property, extracting the contents of the input area and resetting it to empty.

```
"State" : Text.T
```

SetProperty recognizes the following property, appending a text to the end of the output area.

```
"State" : Text.T
```

3. Dialogs

A dialog is "user interface subroutine"; in first approximation it is just a collection of interactors. However, a simple collection of interactors would be hard to handle because one would have to know the peculiar interface and properties of each interactor. Client programs using the dialog would have this information wired in, and would then be difficult to maintain.

Hence a dialog is, visually, just a collection of interactors, but from the client point of view is a more abstract object. Dialogs maintain (1) a list of named interactors, and (2) an association of client procedures with abstract "events" (text strings) generated by the interactors.

Interactors in a dialog are setup to generate such events in response to user actions; the dialog then takes those events and calls the corresponding client procedures. The client procedures can then inspect and modify the state of the dialog and its interactors.

The "events" are abstract because the same event can be generated by different interactors, and single interactors can generate many events. Hence, the dialog can be restructured to some degree without changing its event behavior.

The state of a dialog consists mostly of the state of its interactors, and this is accessible only by knowing the name of a particular interactor and operating on it by "GetProperty" and "SetProperty" methods. Hence it would seem that again we are fully dependent on the inner structure of the dialog. However, some simple conventions help here in operating in an abstract way. For example, the "main" property of each interactor is normally called "State". This can have very different meanings depending on the interactors (a number, a menu selection, a large piece of text), but it helps in operating uniformly on some classes of interactors.

Dialogs also provide a notion of "interactor groups". Each interactor is optionally associated to a "group" interactor (no groups of groups). Groups are useful for dealing with interactors which are geometrically unconnected but logically related, like a mutually exclusive set of radio buttons, or a set of pulldown menus in a menu bar.

Dialogs have a minimum size; when placed in a window smaller than their minimum size they are incompletely shown. Dialog elements have stretching information,

which determines what happens to them when the dialog changes sizes beyond its minimum size.

The stretching model, intuitively, is as follows. A dialog is a rectangular piece of rubber (cartesian rubber, which can only be stretched uniformly and orthogonally). The dialog elements are similar rectangular pieces of rubber, placed on top of it, with each edge "stapled" to the underlying dialog. A "staple" connects one point on the edge of an element to one point of the dialog (for vertical edges, the vertical coordinates of such points are irrelevant, similarly for horizontal edges). The length of the "staples" does not change under stretching.

Suppose that the edges are stapled "in place" to the underlying dialog. If we now stretch the dialog, the elements will stretch proportionally, as if they were painted on the dialog.

But we can also staple edges to some other point. For example, we can staple all four edges of an elements to the corresponding edges of the dialog. When we stretch the dialog, the element will stretch so that the distance of its edges from the edges of the dialog remains unchanged, i.e. the element becomes "as big as possible" while maintaining its borders.

Another common situation is to staple both vertical edges of an element to the west edge of the dialog, and both horizontal edges to the north of the dialog. Under stretching, the element will preserve its size, and will "stick" to the north-west corner of the dialog.

4. Chassis

A "window chassis" makes it easier to turn an application into a legal Trestle window. All applications should be implemented as VBTs with specialized behavior. This allows applications to be embedded in other applications, if needed, and to be composed with other VBTs by the use of "splits".

A chassis takes any VBT and installs it in a split structure which handles icons, title bars, borders, dragging, iconization, reshaping, etc. It also maintains a pulldown menu bar, and a place where menus and dialogs can be popped up.

5. Root Dialogs and standard popup dialogs

Dialogs were not conceived, and are not suitable, for implementing arbitrarily complex user interfaces. The range of applications which can be implemented this way depends a great deal on the range of interactors one provides, but even then there are limitations.

However, many simple and not-so-simple applications can be built entirely out of dialogs. For people who do not see user interfaces as the central part of their application-building work, this can be quite useful.

For these situation, a single dialog taking over the application window can be "the" user interface (often with the help of additional popup dialogs). This principal dialog is called a "root" dialog.

The main dialog interface (DialogVBT.def) is very general; it is the same used by the dialog editor, and hence supports arbitrary manipulations. For simpler usages such as root dialogs, a simplified interface (RootDialog.def) can be used. A root dialog is essentially a dialog sitting in a chassis, and hence provides all the facilities needed to be hooked into Trestle.

Other common needs of simple applications include (1) give warning messages, (2) ask simple questions to the user, and (3) open and close files. The first two cases are covered by MessageDialogVBT.def, and the third by FileDialogVBT.def.

6. Tints, Frames and Looks

"Looks" are a pervasive data structure in the toolkit. They are used to display textures, bitmaps and text. In principle, it should be possible to have an arbitrary image as the looks of, say, a button. For various practical reasons, looks are a somewhat more specialized data structures.

Looks are composed of a "frame" and an "image" to be shown in the frame. Just like a physical picture frame has a back board, a border and a piece of glass, a "frame" has a background, a border and an "overlay" to achieve special translucence effects. The image can be either a uniform tint, a texture, a bitmap, or text in some font. An important piece of information missing from looks is their size: looks stretch to cover the available space (nonetheless, they have a minimum bounding rectangle).

A "tint" is a function from colors to colors which is applied to the background during painting. Only some such functions are actually supported. Basically, there are three classes: (1) "constant" tints opaquely paint a given color over the background (black or white for monochrome, RGB for color); (2) "swap" tints swap two given colors and have unpredictable effects on other colors (in monochrome, the unique swap tint inverts black and white); (3) the "transparent" tint leaves the background unaffected. Color tints are specified by RGB values (24 bits), and painted in the "closest" color found in the color map (8 bits). Color tints can also be set to "high quality", in which case the given RGB value is approximated by four color map colors painted in 2x2 squares.

A "tint pair" is a pair of tints, used to paint background/foreground images.

A "texture" consist of two 16x16 bit patterns (normally also called textures), one for painting in monochrome and one for painting in color, plus a tint pair to specify the coloring (background tint for the "white" pixels, and foreground tint for the "black" pixels of the pattern).

A "bitmap" consists of an arbitrary-size bit pattern with a tintpair for painting. Optionally, one can specify an additional bit pattern of the same size (the "mask") and an additional tint pair. This allows one to paint bitmaps in three colors, normally the background, the borders and the inside of an image.

A "font" consists of a screen font and a tint pair for coloring.

A "frame" consists of (a) border thicknesses for the four edges, (b) a border tint, (c) an overlay texture (d) an image which is either a tint, a texture, a font or a bitmap.

Finally, a "looks" specifies: (a) a frame; (b) for font frames, a piece of text; (c) for font and bitmap frames, horizontal and vertical alignments and margins. The alignments are left, center and right, horizontally, and top, middle and bottom, vertically. Margins determine distances between the edge of the looks, and the corresponding edge of non-centered text or bitmaps.

7. Dialog Editor

The toolkit provides a library of functions which could be used to build dialogs and user interfaces by brute force programming. However, this becomes an

unmanageable task even for simple user interfaces, because of all the boring details involved in describing geometric layouts. This makes experimentation very difficult, and results in uglier and less effective user interfaces.

To obviate this problem, a dialog editor is provided as an integral (and probably the most interesting) part of the toolkit. In judging some of the peculiar properties of the toolkit, one should keep in mind that it has been structured with the primary goal of making the dialog editor work smoothly.

Using the dialog editor, one can assemble dialogs very quickly by direct manipulation. This involves (a) fixing the geometric layout (b) specifying the looks of the interactors and of the whole dialog (c) specifying the events generated by the interactors (d) fixing the initial state and properties of the interactors (e) specifying the behavior of the dialog under stretching.

Dialogs are saved to disk as pickles (in fact, as pickled "descriptions"). To use them, client programs have to (a) load them from disk (b) register procedures to respond to events (c) pop them up and take them down.

New interactors can be introduced in a dialog from a "Create" menu. Interactor properties can be inspected and changed from a "Change" menu, which also has a command for extending groups of interactors, if applicable ("Create" always creates new groups).

The geometric layout is defined (in the dialog editor "Edit" mode) by creating interactors, reshaping them and dragging them around. The interface is MacDraw-like; one can select many items, move them around in groups, cut and paste groups, and change the shape of all the elements of a group in a single operation. A single selections is done by clicking on an item (which highlights 8 control points for reshaping that item). A multiple selection is done by shift clicking on other items, or by sweeping a rectangle with the mouse (all the elements intersecting the rectangle will be selected). Moving is done by dragging items, and reshaping by dragging control points.

In "Stretch" mode one can define the stretching parameters of the elements. Group selections and operations are available here too. Four control points ("staples") are available for each element; they are originally on the edge, and can be dragged to other locations. Double clicking a control point projects it to the closest dialog edge. Double clicking an item projects all its control points. A good way to start is to select all items and double click one of them so that all control points will be projected to some dialog edge. Often this does most of the work, by making some items (close to corners) stick to corners, and other items (close to the center) expand under stretching. Note: moving an item in "Edit" mode resets its stretching control points.

The dialog editor has no knowledge of the kind of interactors it is dealing with; how can it then change their properties and looks? It works this way. When a new interactor (I) is added to the dialog editor, a new "matching" dialog (D) must be supplied with it. This dialog has one field for each property of the interactor (i.e. for each property P of I, D has has one interactor whose name in the dialog is P, and whose "State" property represents the value of P).

When the user selects "Change Attributes" in the dialog editor, the interactor is asked to supply a "description" of itself (a list of <name,value> pairs). This description is transformed, in uniform way, into a "dialog setting" (which is also a description consisting of <name, <"State", value>> triples). The dialog is than told to assume this setting and is popped up, hence showing the current state of the interactor. The user can then modify the dialog; when done, the dialog is asked to supply its dialog setting, which is converted back into a description. The description is then forced upon the interactor, which assumes the new state. In all this process, the dialog editor has no idea of what the interactor does, what the dialog contains, or what the properties mean.

Some of the properties which are defined this way are the "events" generated by the interactors. These events are just strings, and must match the event strings which are used by the client program to attach event procedures to a dialog.

The dialog editor has a "Test" mode, in which a dialog behaves like it would when running in an application (except that no procedures are attached to its events). There is also a "Test+Debug" mode; this is like test mode with an additional "Debug" dialog. This dialog reports all the events generated by a dialog, and can be used to set and inspect the "State" of its interactors.

The looks of an interactor are just some of its properties, hence they can be modified as described above (the "matching" dialog of an interactor may contain some special buttons, called "EditLooks" buttons, which pop up dialogs for defining looks properties).

8. Dialog Resources

While the looks of an interactor are local to it, the tints, tintpairs, textures, bitmaps, fonts and frames which define such looks are shared among all the interactors of a dialog. This allows one to change the looks of many interactors at once (e.g. their color).

The above shared quantities are called "dialog resources"; a menu in the dialog editor gives access to them. Each resource is identified by a resource name (a string) within another resource or a looks.

For example, a looks L will have a frame called e.g. "FrameDefault" which can be shared by many other looks. To change the appearance of L, one replaces the string "FrameDefault" by, e.g. "FrameSpecial". If "FrameSpecial" already exists in the list of dialog resources, the looks L will change accordingly, otherwise a new "blank" frame is created. To change the appearance of all the looks which mention "FrameDefault" one selects "FrameDefault" from the resources menus, and modifies it (which involves modifying more resource names).

New dialogs have a predefined collection of resources, which are used for the default looks of the available interactors. The following conventions are being used for the names of such resources. The name of frame resources begin with "Frame", similarly for tints, etc. Frames beginning with "FrameItem" defined neutral button-like looks (e.g. a white rectangle with a black border, when the button is in normal state). Frames beginning with "FrameText" define text frames with a border. Frames beginning with "FrameLabel" define text frames without border.

Unused dialog resources are garbage collected when saving to disk. The standard resources are made available again when loading a dialog in the dialog editor, unless there are already different resources with the same name.

9. Resource Paths

An application using the vbt toolkit will normally have a number of privated resources (pickled textures, bitmaps and dialogs) which are loaded when the application is started. This gives flexibility in customizing applications, since each site and even each user can modify and replace resources without changing the program source, recompiling, or relinking.

The problem is to make sure that these resources are located correctly by the application, and that when the application executable is moved, its resources are moved with it.

This is the solution currently adopted. An application is a directory (as opposed to a simple executable binary). The directory contains executables (normally called "Mainz" for topaz and "Mainx" for ultrix), an optional pickled bitmap for the icon (normally called "Icon") and all other pickled resources the application may need.

A resource path is a list of directories separated by ":", to be searched in sequence from left to right. Many routines in the vbt toolkit take a resource path as an argument. This argument can have the format described above, but the normal default is the string "$1", indicating that the resource path is to be taken from the first program argument, as defined by Params.GetParameter(1).

An application (e.g.: "Chickens") is normally invoked as follows (assuming it has been placed in /proj/topaz/lib):

```
/proj/topaz/lib/Chickens/Mainz /proj/topaz/lib/Chickens
```

i.e. the first argument to the program is the directory (or directories) where the resources are to be found.

10. How to build a RootDialog application

The principal dialog of an application is called the "root" dialog, since it takes over the application window. The RootDialog interface is sufficient for applications which only need a root dialog.

You will probably also need MessageDialogVBT (for simple messages and Yes/No questions) and FileDialogVBT (for opening and closing files from a popup dialog). If your application needs additional customized dialogs to be popped on top of the root dialog, see DialogVBT.

To make a RootDialog application:

0) Look at /proj/packages/nullapplication as an example. Create a package called "myapplication" (if your application is to be called "MyApplication").

1) Do "man DialogEditor". Enter the dialog editor and read the Help information from the "File" menu. Create a dialog with the dialog editor and write a pickled version of it on disk. Some components of the dialog will be able to generate "events," and each event will be given a string name.

2) Look at NullApplication.mod; this is a trivial example. Look at FileApplication.mod; this is a more interesting example. Write a handler procedure for each named event that your dialog can generate. Write a program which reads a root dialog (from your pickled dialog), registers the handler procedures, and then calls RootDialog.Interact to get things started.

3) Look at Makefile.NullApplication. Modify it to create a makefile for your application.

4) Look at the directory NullApplicationDir/. Create a directory called "myapplication/MyApplicationDir". It should contain (A) the program executables, called "Mainz" and "Mainx" by convention, generated by the standard makefile; (B) the pickled bitmap for the icon, called "Icon" by convention (use the BitmapEditor to create it; you can omit in test phase). (C) the root dialog, called "Dialog" by convention, (D) any other pickled resources your application needs. (E) a shell script called "MyApplicationz" containing the line

"/proj/topaz/lib/MyApplicationDir/Mainz
/proj/topaz/lib/MyApplicationDir:/proj/topaz/lib/ToolkitResources $* &",
and another shell script called "MyApplicationx" containing the line
"/proj/ultrix/lib/MyApplicationDir/Mainx
/proj/ultrix/lib/MyApplicationDir:/proj/ultrix/lib/ToolkitResources $* &".
Make the scripts executable ("chmod +x MyApplicationz MyApplicationx").
(the standard makefile will install MyApplicationDir into /usr/topaz/lib,
MyApplicationDir/MyApplicationz as /proj/topaz/bin/MyApplication, and
MyApplicationDir/MyApplicationx as /proj/ultrix/bin/MyApplication).

5) To test your program (on a firefly), cd to "myapplication" and type (or prepare a script containing) "MyApplicationDir/Mainz MyApplicationDir:/proj/topaz/lib/ToolkitResources $* &". All this works assuming you have given the default "$1" parameter to RootDialog.New; see ResourcePath.def for more details.

6) Write a man page file called "MyApplication.manpage".

11. Text Editing

This is the text editing interface for text-like interactors. Note that even the smallest ones (like text lines and numeric areas) support search, popup menus, and cross-address-space cut&paste.

A text area is an area of the screen where text can be presented and edited. It supports mouse positionings and selections, and menu-driven (as well as function-key-driven) editing operations.

The interface is carefully split into two levels. The first level is functionally complete, supports a single text selection, requires only one hand (and no function keys) for editing operations and very little to memorize.

The second level supports two selections, requires two hands to operate and has several function-key shortcuts. (You may have trouble with this when coming back from vacations.)

11.1 Vocabulary

A "position" is an integer denoting the location between two characters; position 0 is before the first character of a text.

A "selection" is determined by two positions n,m (n<=m).

An "empty selection" is a selection with n=m.

Text areas support (and highlight) two selections, called the "primary selection" and the "secondary selection". There are only at most one primary and one secondary selection in the universe, at any given time.

The text area with the primary selection (if any) is called the "primary area", the one with the secondary selection (if any) is called the "secondary area". Primary and secondary areas may coincide.

The primary area is said to have "input focus"; characters typed on the keyboard go to the primary area, i.e. to the area with input focus.

To set the focus of an area "on" means to make it the primary area, i.e. to create a primary selection in it. To set the focus of an area "off" means to move the primary selection to a different area, or to go in a state with no primary selection.

"Press" means to push a mouse button down and keep it there.
"Release" means to pop a mouse button up from the press state.
"Drag" means moving the mouse with a button pressed.
"Click" is a quick Press-Release action on the same button.
"Click-Press" is a quick Press-Release-Press action on the same button.
"Double-Click" is a quick Click-Click action on the same button.

11.2 User Interface

11.2.1 Level A (Naive users... single selection, free left hand)

Generalities:
Left Button is for positioning and making selections.
Middle Button is for extending selections.
Right Button is for the cut&paste pop-up menu.

Left Button:
Press: begin selection at the nearest character boundary.
Drag: extend selection by characters.
Release: end selection at the nearest character boundary.

Click-press: select word; begin selection at the nearest word boundary.
Drag: extend selection by words
Release: end selection by words at the nearest word boundary.

Middle Button:
Press: begin adjust selection for the closest selection boundary.
Drag: continue adjust selection by characters.
Release: end adjust selection.

Click-press: begin adjust selection for the closest selection boundary.
Drag: continue adjust selection by words.
Release: end adjust selection.

Right Button:
Menu:

Cut: cut selection to cut-buffer	[F3+F4]
Paste: paste cut-buffer to selection	[F5]
Copy: copy selection to cut-buffer	[F4]
Erase: delete selection	[F3]

Type-in:
Printable characters: Delete selection, insert character.
<X]: delete character backwards if selection is empty;
delete selection otherwise.
Option-<X]: delete character forward if selection is empty;
delete selection otherwise.

11.2.2 Level B (advanced users... two selections, two hands)

Generalities:
Secondary selections are enabled by holding the shift key down,
and disappear when the shift key is released. While the shift
key is down one can (in any order and repeatedly) make secondary
selections and select actions off the menu. When the shift key
is released, the latest action has effect on the latest selection.
Shifted Left Button: for positioning and making secondary selections
Shifted Middle Button: for extending secondary selections
Shifted Right Button: for the move&stuff pop-up menu

Left Button:
(like Level A)
with Shift key: like Level A, operating on secondary selection.

Middle Button:
(like Level A)
with Shift key: like Level A, operating on secondary selection.

Right Button:
Menu (with Shift key):
(here 1 and 2 are the primary and secondary selections):

Move: replace 1 by a copy of 2, and delete 2	[F3+F4]
Stuff: replace 1 by a copy of 2	[F4]
Swap: swap 1 and 2, if disjoint	[F5]
Wipe: delete 2	[F3]
(cancel) cancel operation	

Type-in:
(like Level 1)

11.2.3 Shortcuts

Modifiers:
Holding down the escape key is another way of making word and word-boundary selections, equivalent to double clicking.

Function keys:
As described in the menus. The actions happen on up-transitions. While holding a function key down, the secondary selection menu is available on the right button. Performing a selection from the menu overrides the function key.

Instant action:
The following actions:
"with shift key pressed perform secondary selection (by left and middle buttons) and select action marked Fn in the menu, then release shift key"
may be abbreviated by:
"with function key Fn pressed perform secondary selection (by left and middle buttons), then release Fn"
An instant action may be cancelled by releasing the function key before releasing the mouse button, or by selecting "(cancel)" from the menu.

11.2.4 Search (experimental feature)

Hold down the left control key and type: the string you are typing will be forward-searched incrementally as you type. Backspace allows you to backtrack on the search string, but does not backtrack the search. Release the left control key when done.

Click (press-release) the left control key. If the primary selection is empty, the last string which was searched for will be forward-searched. If instead the primary selection is non-empty, the contents of the selection will be forward-searched.

Backward search is done the same way, but with the right control key. All searches are case-insensitive.

11.3 Notes

This interface works best when a different finger is used for each mouse button. An alternative (slightly slower) is to use one finger for the left and middle button, and another finger for the right button. Using the same finger for the three buttons in not advised.

Menus display the name of the function key that is equivalent to a menu command.

Left click followed by middle click can be used to select in two steps, instead of draw-through. Left click followed by middle click in the same position selects a single character.

The cut buffer contains the last text that was either "cut" or selected as a secondary selection involved in some action. For example: after a "stuff" or a "move" the cut buffer contains what was the (now empty) secondary selection; repeated "paste" will keep pasting it into new primary selections.

"paste", "move" and "stuff" change the primary selection into an insertion point at the right edge of the selection.

"swap" of two overlapping selections is a no-op.

Button clicks set the input focus if they set the primary selection.