NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

CMU-CS-84-167

The Box, A Layout Abstraction For User Interface Toolkits

Joëlle Coutaz

Computer Science Department
Carnegie-Mellon University
Pittsburgh,Pa. 15213

13 December 1984

Abstract

The strict separation of the functionality of a system from the user interface is considered as a reasonable design principle. One promising way to enforce this separation is to supply the system designer with a "user interface toolkit", a set of integrated software tools for implementing the user interface. The difficulty resides in determining the "right" abstractions to be implemented by the toolkit. So far, there is no satisfactory answer to this problem, only propositions. Little has been proposed for object-oriented I/O as a toolkit facility. Yet, applications are currently bound to express I/O in terms of low level abstractions. As a result, they are in charge of tasks that are irrelevant to their functionality. In this paper, we propose the *Box* as a mechanism to permit applications to handle I/O in terms of their own abstractions and to be relieved from irrelevant tasks. The box mechanism has been implemented and is currently available on Perq/Accent.

This research was initially performed at the laboratoire of Informatique et Mathématiques Appliquées de Grenoble (IMAG), France and was sponsored by the Centre National d'Etude des Télécommunications (CNET). It has been continued as part of the C-MU Spice Project.

Keywords: Object-oriented I/O, Interactive screen layout, User interface.

Table of Contents

1 Introduction
2 Motivation
3 The Requirements
    3.1 The Requirements of the Applications
    3.2 The User's Requirements
4 Satisfying the Applications Requirements
    4.1 Definition of the Box
    4.2 The Spatial Attributes
    4.3 The Link Attribute
    4.4 Adornment Attributes
5 Satisfying the User's Requirements
    5.1 Multiple Screen-Views
    5.2 Screen Space Optimization
6 conclusion

List of Figures

Figure 1: The Box is a View with the Compositor and the Formatter as Agents. 3
Figure 2: An "If" Box for the "If" Statement. 7
Figure 3: Screen-view in a "wide" window. 7
Figure 4: Screen-view in a "narrow" window. 7
Figure 5: Behavioral features of a Menu. 9
Figure 6: A form defined with the root-box hierarchy facility. 10

1 Introduction

Scientific analysis of user interface design is a fairly new area of research but it is receiving increasingly widespread attention. As a result, useful design principles are progressively emerging. Most of these principles are attempts to provide the designer with guidelines for deriving a manner of interaction between the user and the computer. Other principles, which constitute a minority, are mainly concerned with structural relationships between the system and the user interface. In particular, there is a consensus among designers on the need for a rigorous separation between the functionality of a computer system and its user interface. The functionality of a system is defined by the set of tasks (or applications) that the system is able to perform. The user interface mediates between the applications and the user through physical I/O devices. Given the current situation, where guidelines for defining the interaction with the user are sometimes contradictory, the separation between the applications and the user interface is, in our opinion, a reasonable design principle: a proper modularity permits the applications and the user interface to evolve independently. In this paper, we accept this principle of separation and examine one feasible way to put it into practice.

As shown by previous experiments [9, 12], one promising way to enforce the separation between the applications and the user interface is the notion of a user interface toolkit. Broadly speaking, the toolkit defines a language that is usable in all of the applications to express the user interface. It can therefore increase consistency across applications; it can also reduce applications' code size and development time. It is easy to speculate about the fundamental purpose of the notion of user interface toolkit. It is difficult to define the appropriate set of abstractions to be integrated into a toolkit that would be *useful*. Because of the function of the toolkit, these abstractions should be satisfactory for both the application designer and the user.

With regard to the user interface, the designer's task consists of defining the dialogue with the user and managing the physical I/O devices which make the dialogue possible. Unfortunately, there are many different types of terminals and each user has his own particular tastes and aptitudes. This situation therefore makes it complex for application designers to implement a user interface. Since the strategy usually adopted for managing complexity is to use abstractions, the primary property of the toolkit abstractions will be to relieve the designer of the low level details of the interaction. The second property of these abstractions will be their ability to fit the level of abstractions handled by the applications. With regard to the user, one desirable requirement is that the toolkit abstractions should convey features for better user satisfaction. Unfortunately, there is no general theory about how to embody human factors knowledge properly in such abstractions. In addition, each user and each

task have specific requirements. Given our current lack of knowledge, a good approach is to develop some understanding in this area through experimentation. The Descartes and Cousin projects [10, 5], Apple with the Macintosh computer [12] and our own study [4] have made attempts in this direction.

The Descartes project uses principles of programming methodology and language design to elaborate a data base of abstractions which should offer a wide variety of styles of interaction. At the other extreme, the Macintosh "user interface toolbox" proposes one style of interaction: a graphically oriented approach based on the extensive use of the mouse. The Macintosh toolkit relies on the powerful notion of "resource" (or object) such as menu, icon. Resources are maintained outside of the application code. As a result, the application designer can elaborate new resources from already existing ones and can modify resources without recompilation of the application code.

In our study, we first analysed the basic needs of applications with regard to communication. From this observation, we identified two important issues that were not fully covered by previous work: dialogue independence and the expression of object-oriented I/O. A detailed description of how to achieve dialogue independence can be found in Cousin [5] or in a previous paper [4]. In the current paper, we are concerned with interactive display-oriented inputs and outputs.

Some works have already addressed this topic. Pic [7] provides an interesting language for typesetting graphics but is not an interactive tool. Pen [2] is an interactive display editor but is character oriented. Janus [3] and Mint [6] include an interactive layout facility which supports heterogeneous data but which is tailored to document preparation needs and is not available as a general purpose tool. Therefore, applications are generally bound to express information layouts in terms of low level abstractions. (When lucky, applications are provided with primitives from a window package). Besides, in order to permit full interactive use of the display (such as selection, scrolling), applications have to perform heavy duty bookkeeping (such as maintaining the location of the information or optimizing display outputs) that has little to do with their purpose.

In this article, we propose a solution that relieves applications from performing tasks that are irrelevant to their functionality. The solution is based on the concept of *Box* which permits applications to express display-oriented I/O in terms of their own objects. The box mechanism has been implemented and is currently available under Spice/Accent [1] for a Perq bitmap workstation. Experiments have also been made using it under Multics and VMS for CRT terminals.

Section 2 provides the motivation for this approach. Section 3 presents requirements that consider

both the application and the user. Sections 4 and 5 indicate how these requirements are satisfied by the box for the applications and the user respectively.

2 Motivation

Broadly speaking, applications reason in terms of specific abstractions (or objects), to perform a task. An object is made concrete through the existence of views which are representations of this object, tailored to the needs of specific agents. In particular, the *screen-view*, i.e. the view of an object appearing on a screen terminal has two agents: the user and the application that owns the object. Both of them perform actions on the same view. The ability of sharing a view between various agents is, in general, a powerful basis for communication. Unfortunately, a view that is good for one agent, may be bad for others. For example, in our case, the user may find it convenient to scroll a screen-view; conversely, the application may find it inconvenient to have to generate a new view and generally, does not desire to be bothered by user's actions that are semantically irrelevant. This observation lead us to the definition of an additional view, the *Box*, that would act as a filter between the application and the user. The box has two agents, the *compositor* and the *formatter*, that are related to the application and the user agents in a way shown in figure 1: the application provides the compositor with the description of the box. From this description, the compositor elaborates the box which is eventually processed by the formatter to produce the screen-view.

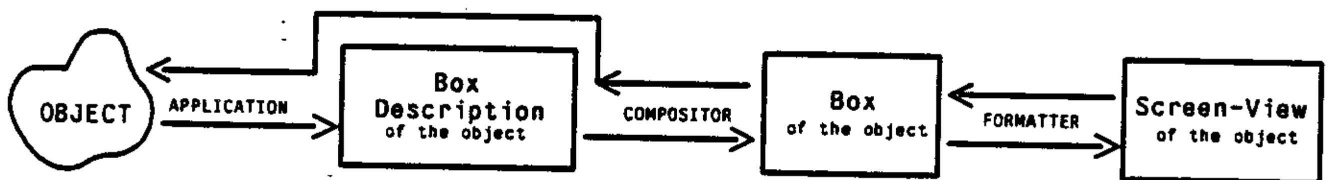

Figure 1: The Box is a View with the Compositor and the Formatter as Agents.

Having made these general observations about the primary purpose of the box, we are ready to provide a detailed description of the design. The box has been designed to satisfy some prerequisites that concern both the applications and the user. In section 3, we examine these requirements. The subsequent sections will then present the box and the solutions that are suited to each requirement.

3 The Requirements

This section identifies requirements that the box should reasonably satisfy. It is concerned first with the application, then with the user.

3.1 The Requirements of the Applications

The primary goal is to permit applications to *express high level inputs and outputs*. Applications manipulate objects. Thus, they may find it very convenient to ask the box mechanism to display this object, or to require it to determine which object has been designated by the user; they may not want to be concerned by how the layout of the object is performed nor by how the object is designated nor by where (in terms of window coordinates) the object is located. Sections 4.2 and 4.3 show how the box mechanism gives an answer to the problem.

The second requirement is concerned with the *expression of structural and spatial relationships between objects*. Some objects are built up from more primitive objects. For example, an "if" statement is made of the "condition" part, the "then" and the "else" parts. Section 4.1 explains how the box corresponding to the statement results from the composition of the boxes representing the components. In addition, some objects have internal semantic relationships that the application may want to express in a layout by a spatial relationship. For example, a scrollbar and a list of scrollable items have to appear contiguously on the screen. Section 4.2 and the figure 5 indicate how the boxes representing the scrollbar and the items can be related by spatial constraints.

Another important issue is to allow applications to *assign behavioral features to objects*. Behavioral features make it possible for applications to produce immediate feedback to a user's action on an object. They permit applications to show the user their current understanding of the action as well as the current state of the object. For example, an application may wish to associate the behavior "stand out" to a menu-item in order to confirm the selection to the user. Section 4.4 specifies how the box provides this facility.

Given the dynamic nature of interactive computing, applications must be able to *dynamically modify a box*. In particular, applications may modify structural and spatial relationships, or assign a new behavior or a new value to an object with an immediate and automatic feedback on the screen. This is made possible in a simple way: the application specifies the modification of the box to the compositor. The compositor modifies the box, then it transmits the request to the associated formatter which eventually updates the screen-view.

As our last requirement, applications may wish to be *unaware of events that are irrelevant to their task*. Such events, when happening to a box, are handled by the formatter of the box. As a result, applications are relieved from the burden of refreshing the screen when the size (or location) of a window is modified (or uncovered), or when the user performs scrolling operations.

So far, we have enumerated requirements with regard to applications. We now turn our attention to the user.

3.2 The User's Requirements

The user may desire *different screen-views for an object*, depending on screen space availability or the current focus of attention. Multiple screen representations can be handled in two ways. One approach is the use of ellipsis [8] as a vehicle for expanding or contracting the screen-view. For example, the user entering a document constructor, may initially display the document as a table of contents, then request the expansion of the chapter he is interested in. A second way to get multiple screen-views is simply to associate several screen-views with an object. For example, the text "HOUSE", the picture of the house, and the floor plan of the house are distinct visualizations of the object "house". Unfortunately, it is often the case that in non integrated environments, these views are generated by distinct tools without clear semantic links. As shown in section 5.1, multiple representations provided by the box rely on the use of ellipsis.

Facilities, besides window facilities, should also be provided to *overcome screen space limitations*. Ellipsis is one means. Scrolling is the usual trick. Nevertheless, scrolling is not suited to tasks which require distinct parts of an object to be simultaneously visible. For example, a programmer building the body of a procedure needs access to the declarative part. Frequent round-trips between various parts of an object compromise the productivity. In section 5.2 we show how sub-objects can be locally scrolled without need for the whole object to be scrolled. Another way to overcome screen size limitations is distribution. Distribution mentioned in section 5.2 means that the visual representation of an object is split across several windows.

This section identified the requirements for the benefit of both the applications and the user. The next section is devoted to the definition of the box and focusses attention on the way each application requirement is satisfied.

4 Satisfying the Applications Requirements

4.1 Definition of the Box

A box has to represent structural relationships between objects. Therefore, it is of prime importance that it be able to propagate structural properties. For this reason, a box is a tree-like structure which facilitates the definition of an inheritance mechanism.

- Conceptually, *leaves* contain a piece of information and wrap an imaginary rectangle around this information. The size of the rectangle may be constrained by the application or determined by the volume of information itself. In practice, for sake of generality, leaves are gateways to some specialized tool for the production of the information. In the current implementation, gateways are available to give access to small chunks of text, pictures and text files. As a result, applications can mix information of various types in the same window (see figure 6).

- *Nodes* are compound boxes. A compound box is the result of a structural composition from daughter boxes; it wraps an imaginary rectangle around the resulting composition.

- Each node of the tree is decorated with *attributes*. Most of the attributes are inheritable. When the value of an inheritable attribute is left underspecified in a node, it is inherited from the ancestor node. As shown in the following paragraphs, attributes are a way of satisfying the requirements previously mentioned. They are comprised of the spatial attributes for the expression of spatial relations, the link attribute for object-oriented input (i.e. selection) and of various adornment attributes for the expression of behavioral features.

4.2 The Spatial Attributes

Spatial attributes are comprised of the formatting attribute and attributes that express indentation and spacing between boxes or constrain the size of the rectangle where the content of the box is to be placed. In this section, we emphasize the use of the formatting attribute.

The formatting attribute of a compound box determines the relative positions of the component boxes. It may have one of the following values : **H**, **V**, **HorV**, **HandV**.

- **H**, **V** concatenates the rectangles of the daughters *horizontally* along the upper side, *vertically* along the left side.

- **HorV** first tries to concatenate the daughters *horizontally*. If the resulting rectangle is too wide to fit the available width of the associated window, a *vertical* composition is applied automatically. The components all appear horizontally or all appear vertically.

- **HandV** concatenates the daughters *horizontally* with an automatic *vertical* folding when the horizontal composition overpasses the width of the associated window.

The **H** composition may result in a clipping of the displayed data. As explained in the example below, **HorV** and **HandV** permit applications to handle logically the size limitations of windows.

Figure 2 shows one possible tree of boxes for the object "If statement" of a syntax editor. The interpretation of this tree by a formatter may generate the screen-views shown in figure 3 or figure 4, depending on the effective width of the supporting window.

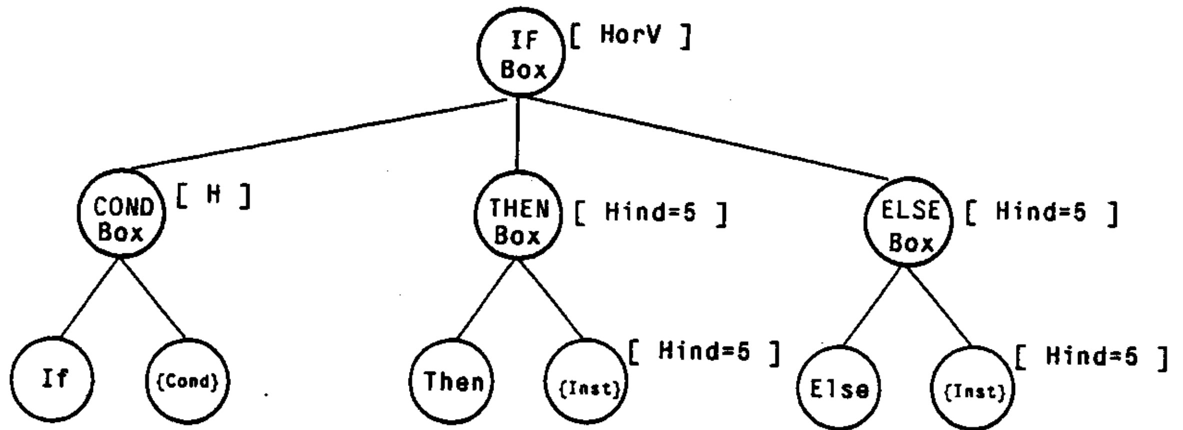

Figure 2: An "If" Box for the "If" Statement.

If the window is wide enough, all of the components of the **If** statement appear on the same line (cf **HorV** attribute of the **If** node).

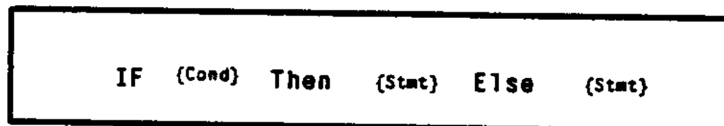

Figure 3: Screen-view in a "wide" window.

If the window is narrow, all of the components are displayed vertically; and because of the **Hind** attribute of the **then** and **else** nodes, the **then** and **else** parts are indented horizontally.

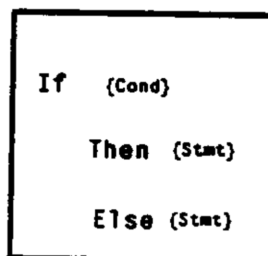

Figure 4: Screen-view in a "narrow" window.

To conclude this paragraph, in order to "write" an object, an application provides the compositor with a box description. For example, the description of the "If" box includes the description of the component boxes along with the specifications of the attributes whose values are different from the standard values. We now show how the link attribute permits applications to perform object-oriented "inputs".

4.3 The Link Attribute

An application program establishes a correspondence between an object of its own and a box by setting the *link* attribute of that box. When the user performs a "select" operation, the leaf box that is currently visited by the cursor is first determined. Then, its *link* attribute is evaluated according to the inheritance mechanism; the tree is climbed until a node with a value set for that attribute is encountered. This node is the root of a subtree which constitutes the box representation of the object that the user has actually selected. The mechanism shows how application programs have full control over the granularity of selectable information. For example, the syntax editor may have decided that the selection of any character of the "if" statement would result in the designation of the whole statement. To achieve this goal, the syntax editor would "link" the compound "if box" to the internal representation of the statement, leaving the link attribute of the other boxes unspecified.

So far, we have seen how an application can perform object-oriented I/O, and how it expresses structural and spatial relations between objects. The last requirement to be studied is the assignment of behavioral features to objects using adornment attributes.

4.4 Adornment Attributes

Adornment attributes, such as reverse video, blinking, colors, frames, fonts, cursor shapes, etc ... belong to the category of tricks which, when judiciously used, can significantly improve the quality of a user interface. To illustrate this assertion, we take the example of a menu of the application "Help" that we are currently developing (figure 5).The left side of the figure shows the tree of boxes with the attribute assignments specified by the application. The right side shows the screen-view of the menu with the shape and localization of the possible cursors. When the user moves the mouse cursor in the upper part of the scrollbar, or when he moves the mouse cursor in the left side of a menu-item, the cursor automatically changes to the shape "↑", "<u>details</u>" respectively. As a result, the user is immediately aware that clicking the upper part of the scrollbar will perform a scrollup and that clicking the left side of an item will provide him with detailed information about the item. There is no need for the application to prompt the user nor to install the "details" or "abstract" modes, avoiding this way the mode pitfalls described by Larry Tesler [11]. Here is an example of a human factor principle

embedded in the tool: (1) the change of the cursor shape is fully handled by the box mechanism, (2) the application directly receives the identity of the clicked object and a location relative to that object (e.g. left, right), (3) the user is notified in a clear and concise way of the facilities provided by the application and (4) the user is able to switch from one facility to another without waste of time.

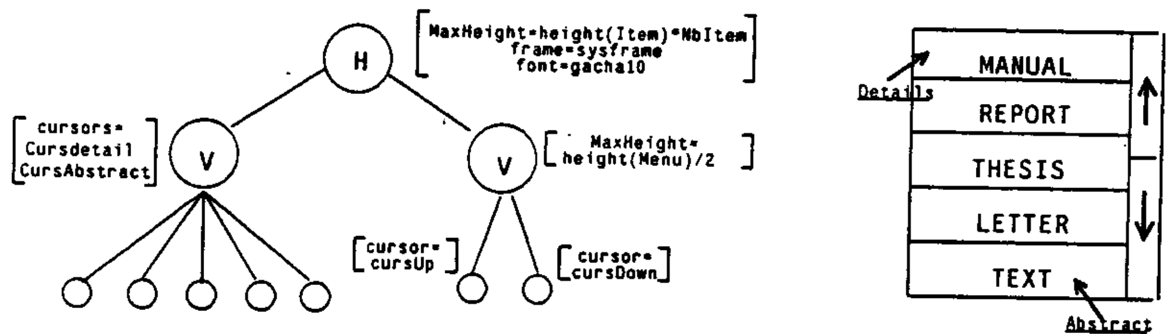

Figure 5: Behavioral features of a Menu.

5 Satisfying the User's Requirements

With regard to layout manipulation, the user's requirements are essentially the following: obtaining multiple screen-views and optimizing the screen space.

5.1 Multiple Screen-Views

Multiple screen-views, provided by the box, are based on the elision principle. This principle permits the generation of distinct screen-views from the same tree of boxes by ignoring subtrees that do not satisfy some specified conditions. Two attributes achieve the multiple representation: the *invisible* and the *finesse* attributes.

1. *Invisible* forces the box to be invisible.

2. *Finesse* is an extension of the invisible attribute. Roughly, it is an integer function with the position of the box in the tree as the main parameter. The box is ignored by a formatter (and therefore is invisible) if its finesse does not satisfy a constraint associated with the formatter. The finesse attribute permits the static definition of a set of possible screen-views for a given tree of boxes. By modifying the constraint of a formatter (with a window editor for example), the user can dynamically obtain distinct views of the same object.

An invisible Box may not be invisible for the user: if the application has provided an alternative description for the Box such as, the sequence ". . ." for the body of a procedure, the formatter considers this alternative. The whole body of the procedure may, in this case, be replaced by the sequence ". . .".

5.2 Screen Space Optimization

Scrolling, distribution and root-box hierarchy permit the user to overcome screen space limitations. We have mentioned in 3.1 how the box mechanism handled *scroll* operations (up, down, left, right). To avoid frequent scrolls, the user may prefer to use the distribution facility. *Distribution* is achieved through tree manipulations along with assignments of subtrees to distinct windows. A subtree may therefore belong to several windows giving birth to distinct screen-views. The box mechanism automatically insures consistency between the multiple screen-views. This feature may help the user to remember the relationships between the various visual representations of an object.

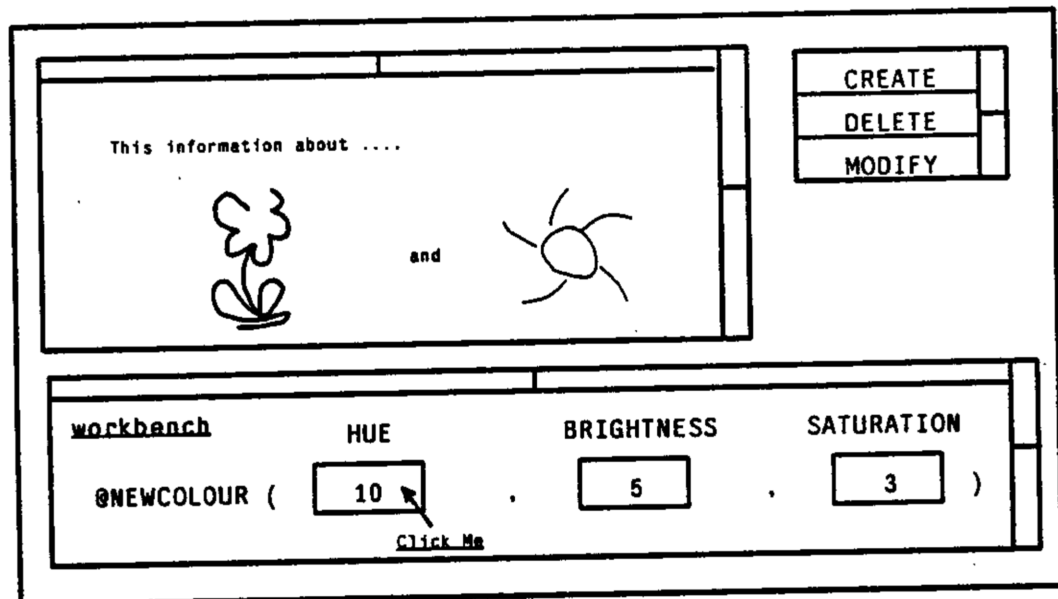

Figure 6: A form defined with the root-box hierarchy facility.

Nevertheless, distribution forces a possibly undesirable scattering of logically connected information. To avoid an uncomfortable scattering, imagine scrolling sub-objects locally, such as the body part of the procedure only, without need for the whole object to be scrolled. By doing so, the declaration part would stay in place, for example, at the top of the window. The notion of *root-box hierarchy* provides this facility. A root-box is a root of a tree and has a formatter assigned to it. This formatter generates and maintains the associated screen-view in the rectangle which corresponds to the tree. (The rectangle of a main root-box is a window). On the request of the application, any node of a tree may be declared as a root-box. As a result, a tree may be processed by a hierarchy of formatters. Each of these formatters have the properties previously described and work independently from each other. The advantage of the root-box hierarchy is shown in figure 6. In this example, the main root-box, which describes a form, is comprised of three root-boxes: one (upper-

left) contains the information resulting from the composition of text and graphics data; the second one (upper-right) corresponds to a list of menu items; the last one (bottom) represents a workbench. The workbench includes three numerical slots. Each slot is a root-box with the V formatting attribute, a height constraint (in order to have only one value visible at a time), the "ClickMe" cursor shape and a set of leaves that contain the numerical values. At any time, and in any order, the user is able to scroll the whole form, scroll any of the three areas, click a slot (to get the next value) or even change the size of the rectangles of the root-boxes (since characteristics of a box may be modified dynamically).

6 conclusion

In this paper, we have described the box, a layout abstraction for user interface toolkits. This abstraction provides applications with an exchange mechanism. For input, it returns the identification of an object. For output, it produces and automatically formats the screen-view of an object. The Box can therefore be considered as a high level input and output logical device with application objects as units of exchange. This property results in two primary advantages. First, applications can pursue their processing in terms of objects even for I/O. In addition, these objects may be associated with data of various types (such as bunch of text, picture, file) and can usefully inform the user with application semantics. Objects may also be manipulated by the user in autonomous areas that behave just like the whole screen-view and that permit him, for example, to scroll a sub-object without need for scrolling the whole parent object. A second general benefit of the Box is that applications are protected from low level I/O bookkeeping and from semantically alien events, such as scrolling, window modifications.

Our recent experience leads to the following concluding remarks: the box is an appropriate tool to build a data base of parameterized interactive objects (such as menus, slots, forms); it significantly facilitates the writing of a user interface indroducing some "nice" features for the user. Nevertheless, it is an experimental tool, only usable on powerful (i.e. 1 Mips, 1Mb) personal computers.

References

1. *The Spice User's Manual.* Carnegie-Mellon University edition, Pittsburgh, Pa. 15213, 1984.

2. D. Barach; D. Taenzer; R. Wells. "Design of the PEN Video Editor Dispaly Module." *Proceedings of the ACM Symposium on text manipulation, Portland, Oregon 16*, 6 (june 1981), 130-136.

3. D. Chamberlin; J. King; D. Slutz; S. Todd; Wade, B. "Janus: An Interactive System for Document & Composition." *Proceedings of the ACM Sigplan 16*, 6 (June 1981), 82-91.

4. J. Coutaz. A Paradigm For User Interface Architecture. C-MU, Computer Science, May, 1984.

5. P. J. Hayes; P. A. Szekely. Graceful Interaction Through the Cousin Command Interface. Carnegie-Mellon University, CMU-CS-83-102, January, 1983.

6. P. Hibbard. *Mint Reference Manual.* Carnegie-Mellon University, Pittsburgh, pa. 15213, 1984.

7. B. Kernighan. "PIC- A LAnguage for Typesetting Graphics." *Proceedings of the ACM Symposium on text manipulation, Portland, Oregon 16*, 6 (june 1981), 92-98.

8. M. Mikelsons. "Prettyprinting in an Interactive Programming Environment." *Proceedings of the ACM Sigplan Sigoa Symposium on Text Manipulation 16*, 6 (June 1981).

9. M. Satyanarayanan. The ITC Project: An Experiment in Large-Scale Distributed Personal Computing. Carnegie-Mellon University, CMU-ITC-84-, october, 1984.

10. M. Shaw; E. Borison; M. Horowitz; T. Lane; D. Nichols; R. Pausch. "Descartes: A Programming Approach to Interactive Display Interfaces." *Proceedings Sigplan 83* (June 1983), 100-111. Symposium on Programming Language Issues in Software Systems

11. L. Tesler. "The Smalltalk Environment." *Byte 6*, 8 (Aug. 1981), 90-147.

12. G. Williams. "The Apple Macintosh Computer." *Byte* (february 1984), 30-54.