A molecular architecture for creating advanced GUIs

Eric Lecolinet

GET / ENST and CNRS LTCI
INFRES Dept.
46 rue Barrault, 75013 Paris, France.
Eric.Lecolinet@enst.fr - www.enst.fr/~elc

ABSTRACT
This paper presents a new GUI architecture for creating advanced interfaces. This model is based on a limited set of general principles that improve flexibility and provide capabilities for implementing information visualization techniques such as magic lenses, transparent tools or semantic zooming. This architecture also makes it possible to create multiple views and application-sharing systems (by sharing views on multiple computer screens) in a simple and uniform way and to handle bi-manual interaction and multiple pointers. An experimental toolkit called *Ubit* was implemented to test the feasibility of this approach. It is based on a pseudo-declarative C++ API that tries to simplify GUI programming by providing a higher level of abstraction.

KEYWORDS: GUI toolkits, GUI architectures, multiple-views, multiple displays, declarative languages, transparent tools, ZUIs, bi-manual interaction, Ubit, brickgets.

INTRODUCTION
This paper presents a new architectural model that simplifies the programming of complex interfaces. Previous studies have shown that GUI programming is a difficult task that requires intensive work and extended knowledge [18]. As a consequence, sophisticated GUIs that implement new visualization or interaction techniques tend to be rare because such systems involve high development costs. Our model was implemented as an experimental GUI toolkit called *Ubit* (for “Ubiquitous Brick Interaction Toolkit”). This toolkit is based on a limited set of general principles that improve flexibility and provide capabilities for implementing information visualization techniques such as magic lenses, transparent tools or semantic zooming. The Ubit GUI architecture also makes it possible to create multiple views and application-sharing systems in a simple and uniform way. Bi-manual and multi-user interaction are also supported by this architecture. The next sections present the main principles of the Ubit model: the “molecular” architecture, pseudo-declarative programming in C++, object sharing and the visual replication of GUI components. The final sections are devoted to multiple display applications, information visualization techniques, related work and the conclusion

WIDGETS AND SCENE GRAPHS
Widget-based toolkits
Most 2D GUI toolkits rely on a *widget-based* architecture. In these systems, interactive objects and object containers are modeled by classes, called *widgets, controls* or *interactors* that implement a large set of features and behaviors. In most cases, these features are statically encoded in widget classes and can not easily be changed nor enriched. For instance, transforming a widget that can only display simple text into a widget that could contain composite text or hypertext elements is generally challenging, even if other widgets in the toolkit provide this capability. Hence, behaviors and other features are not seen as general services that could be used by any widget. Instead, classes tend to link widgets with toolkit features in a static way that cannot be easily changed.

This problem, that can be summarized as “the lack of flexibility of the class/inheritance model” has already been pointed out in previous studies (Amulet [20]). As a consequence, GUIs tend to be quite stereotyped and novel interaction and visualization techniques are rarely used. Originality often requires a lot of complex source code and higher programming costs. While GUIs should certainly respect certain guidelines in order to make them easier to understand, this lack of flexibility is an obstacle to creativity and makes it difficult to experiment new GUI techniques.

Scene graphs
Most 3D toolkits, and some recent advanced 2D toolkits (CPN2000 [2], Jazz [4]), are based on a *scene-graph* approach. 3D scenes are modeled by a graph of nodes

which represent shape objects, properties, cameras, grouping objects, etc. The main difference with the widget-based approach does not reside in the idea of using a graph (a widget tree can be seen as a kind of scene graph) but in the granularity of the nodes of the graph. Widgets are coarse objects embedding many features. Scene-graph nodes are fine grained objects that generally have a much more specific function. This low level of granularity enhances flexibility since many objects of different kinds can be composed together dynamically in various ways. However, scene-graphs may not be well-suited for creating 2D GUIs. Widgets provide a convenient encapsulation of usual interaction styles and a standardized appearance. They also enforce certain usability rules and reduce the number of objects that must be dealt with. This may facilitate programming traditional GUIs, especially interaction. In addition, the scene-graph approach is unfamiliar to most 2D GUI programmers.

MOLECULAR ARCHITECTURE

The Ubit model is a synthesis of the widget-based and scene-graph approaches. As with scene-graph approaches, Ubit GUIs are made up of small “atomic” objects called *bricks*. Hence, the appearance and behavior of the GUI components are not imposed statically by widget classes but result from the combination of inter-changeable light-weight components. However, this model also integrates the usual notion of a widget, although standard widget classes do not embed specific behaviors nor any other feature. Instead, Ubit “widgets” are just combinations of bricks that implement behaviors and other features. In that sense, they can be seen as “molecules” made up of “atomic” bricks. Because of this specific structure, we will call them *brickgets* in order to emphasize the difference with classical widgets. Depending on the programmer’s point of view, a Ubit GUI graph can either be seen as a traditional widget graph or as a scene-graph of atomic elements. A *brickget* can itself be seen as a sub-graph of the global scene-graph. Ubit graphs thus follow a model where scene graph nodes can embed other scene-graphs.

Figure 1 shows an example of a simple scene-graph: a tree that contains three brickget nodes (a check box, a button and a menu) and eight atomic brick leaves. Brickgets can contain an arbitrary combination of bricks that belong to five categories (Fig.2):

- *Behavioral bricks*. For instance, *UEdit* makes all the strings contained in the brickget editable (although it is a button).
- *Graphical properties*, such as *UFont* and *UColor* that specify the appearance of the brickget and its children.
- *Viewable elements*, such as text strings (*UStr*), images (*UIma*), pixmaps and graphical symbols.
- *Callback objects* (*UCall*) that are not illustrated here.
- *nested brickgets* (*UGroup subclasses*). For instance, the button contains a box that “contains” a menu. This latter kind of parent-child relationship means that the menu will be automatically opened when the box is pressed. This box serves as an invisible interactor: it is not discernible but can react to events and change the appearance of its children (as in Fig.!1 where the box is pressed and the arrow symbol highlighted).

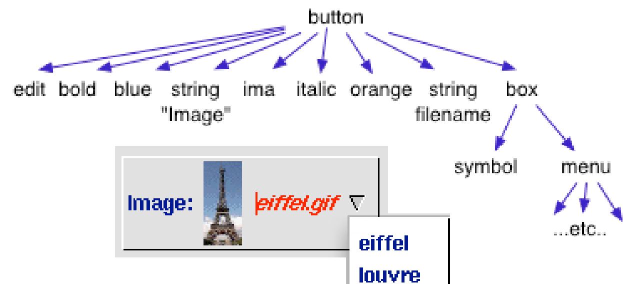

Figure 1: A simple example.

There are several advantages to this approach. First, embedded scene-graphs hide the complexity of traditional monolithic scene-graphs and are therefore easier to program. This may be less confusing for programmers that are accustomed to widget-based approaches (in fact, novice programmers may not even notice the architectural difference). However, this scheme also allows for a high level of flexibility. Brick combination makes it possible to create brickgets that combine any arbitrary set of features. The standard brickgets that are provided by the toolkit can be modified and enriched dynamically by adding bricks or removing bricks and there is no restriction on the type of content a brickget can display nor on the type of behavior it can implement. So, the richness of the toolkit does not reside in the capabilities of its interactive components but in its ability to combine many basic features and services in a generic and uniform way.

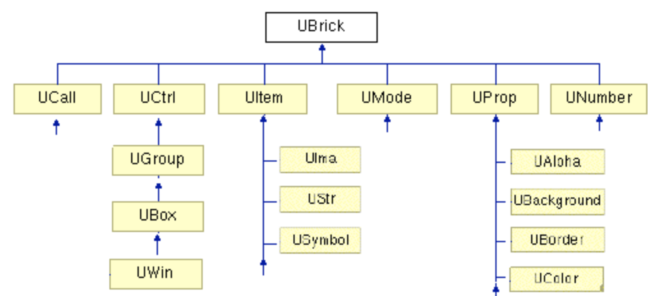

Figure 2: The main classes of the Ubit toolkit

USING C++ AS A DECLARATIVE LANGUAGE

GUI source code tends to be verbose and hard to read. Informative data is often hidden in a large amount of “syntactic sugar” that conveys little information but is necessary for proper compilation. This lack of conciseness tends to make programs harder to understand and to maintain. This fact has been a major argument in favor of

high-level programming languages that provide a higher level of abstraction. Mathematical formulas are another example: their ability to represent sophisticated concepts in a very concise way had a major impact on the development of mathematics.

Declarative languages may be a good solution to ease the design of GUIs. It is for instance surprising that even non specialists can create rather sophisticated HTML pages while most of them would be unable to use a GUI programming toolkit. However, declarative languages generally provide only limited interaction capabilities (with some exceptions such as UIML![27], an XML-based language). Scripting languages propose another approach (e.g. Tcl/Tk [21]). But they require that the GUI and the functional part of the application to be developed separately by using different languages. In both cases, this separation implies connection mechanisms that may be impractical for creating complex interactive programs requiring significant feedback.

Declarative C++. The Ubit toolkit solves this problem by providing a pseudo-declarative API that is compliant with the C++ language. As this API only makes use of the standard features of this programming language, no pre-processing stage is required and the C++ compiler ensures syntactical correctness. Thus, interface specifications can benefit both from the simplicity and compactness of a declarative language and the power of expression of an object-oriented programming language. For instance, the example shown in Fig.!1 could be created as follows:

```
UStr& !   filename = ustr( "eiffel.gif"!);
UColor& color    !  = ucolor( "orange"!);
UIma* !  ima;

UBox& !example1 != !ubutton(
     uedit(!)
     + UFont::bold !+! UColor::blue !+! "Image"
     + ( ima! =! &uima(!"eiffel.gif"!) )
     + UFont::italic !+! color! +! filename
     + ubox( USymbol::down + umenu( ..etc.. ) )
     );
```

String literals such as "Image" are implicitly converted into *UStr* bricks. Graphical properties apply to the children that follow them. They can be instances or predefined constants such as *UFont::bold* and *UColor::blue*. They can also be specified several times inside a single brickget in order to control its subparts independently. This feature can for instance be used to create multi-font multi-colored text, as illustrated in Fig.1. The notion of a graphical property is quite general. This category includes object decorators such as borders, background images and colors, alpha blending, scaling properties, layout managers, brickget locations, etc.

This syntax makes it possible to create an instance graph in a declarative way. It relies on a simple and uniform mechanism. First, expressions such as *ubutton(…)* are functions that call the constructor of the corresponding classes (*UButton* in this case). They are intended to make the code more legible and to give it a "declarative flavor". Brickget constructors have an *UArgs* argument that represents a list of arbitrary bricks. The + operator is overloaded in such a way that the "addition" of two bricks creates a temporary list. More precisely, the expression $a+b+c+d$ first creates a list containing the first two elements then adds the remaining elements to it. This list is then passed as an argument to the brickget constructor which adds these elements to the scene-graph. This mechanism also works when brickgets have only one child or no child at all. A single child is implicitly converted into a list in the first case (thanks to the C++ type conversion mechanism). An empty list is provided as a default argument in the second case. This addition mechanism allows for safe type checking at compile time since it relies mainly on polymorphism and does not involve any dangerous casting operation. The toolkit also provides methods for adding, removing and retrieving brickget children dynamically. For instance, the *addlist()* method makes it possible to add a list of bricks in a single call.

Variables and dependencies

Declarative specifications can be split into several parts by using intermediate variables. This can be useful to enhance the readability of the code source when too many objects are nested. Variables also provide a simple way to share objects and thus, to create a DAG (direct acyclic graph) or even an arbitrary graph. Two brickget parents could for instance share the *example1* object (Fig. 1) in this way:

```
UDialog& d1  =  udialog(!example1!);
UMenu& !m1 =  umenu(!example1!);
```

As will be explained later, *example1* will appear in both its parent brickgets: a menu and a dialog box in this case.

Implicit dependencies are automatically inferred from the scene-graph. This makes it possible to synchronize objects automatically. For instance, the following assignments automatically update *example1* because its depends on the *color* and *filename* variables:

```
color = UColor::green;   filename = "orsay.jpg";
```

This example also shows that brickget subparts can be controlled independently. For instance, the *example1* button contains two strings but the previous code sample only changes the color and contents of the second string. Explicit dependencies that do not rely on the scene-graph can also be specified:

```
filename.onChange( uset(!ima, filename) );
```

This specification means that the *ima* image will be automatically updated when *filename* is changed. More precisely, the *set* method of the image will be called and this method will load the image file given by *filename*. This is a general feature of the toolkit: all bricks can update

dependencies or execute callback functions when their value is changed.

Callbacks and conditional specifications

Dependencies provide an efficient way to reduce the number of callback functions and to avoid the "spaghetti of callbacks" problem [18]. However, callback functions are often useful to connect the GUI with the functional part of the application. Callback functions and *conditional specifications* can be specified in a pseudo-declarative way:

```
class Ex {
    void load(UStr* name) { ... }
} ex;

example1.addlist(                    // adds bricks to example1
    UOn::enter    /  ucall(!&filename, showName!)
    + UOn::action /  ucall(!&ex, &filename, &Ex::load!)
    );
```

The *UCall* brick represents a callback object. This object will call the function in its last argument with the arguments before it when the *UOn* condition is verified. Callback functions can either be non-member or member functions (i.e. methods). In the latter case (illustrated on the second line of the previous example), the instance pointer is provided as the first argument of the *ucall()* construct. Such specifications are based on C++ templates. They ensure that the arguments given to a given *ucall()* construct match the formal parameters of its callback function. Any function or method can be used as a callback function as long as is does not have more than two arguments.

Each *ucall()* specification creates a node in the instance graph. The / operator is overloaded. Its left hand side specifies the relationship between the enclosing brickget and the callback object. This can be a predefined condition, such as *UOn::action* (true when an object was activated, typically by clicking on it) or more general conditions, as explained in the next section. The same mechanism can be used for creating *conditional specifications* that activate (or deactivate) graphical properties or make viewable elements visible (or invisible). For instance, the following code specifies that the *color* object is changed when the mouse enters the brickget and that an image is shown when the mouse is pressed. A sub-tree of brickgets could also be made visible or invisible in this way.

```
example1.addlist(
     UOn::enter     /  uset(!&color, UBColor::red!)
     + UOn::mpress /  uima(!"working.gif"!)
);
```

BRICKGETS AND INHERITANCE IN THE SCENE-GRAPH

Brickgets are simple derivations of three base classes: *UGroup*, *UBox* and *UWin* described below.

Groups and markup tags

A *group brickget* is a node that defines a rendering context in the scene-graph. It acts as a generic container that "glues" together an arbitrary combination of children. Bricks inside a group are evaluated in left-to-right depth-first order. Graphical property bricks apply to the following children in the group (and, possibly, to their children). They do not have any effect on the parents of the group.

```
UBox& my_page = ubox(
     UFlowView::style
     + ugroup( UFont::bold + "Architect Pei's" )
     + " pyramid " + uima(!"pyramid.gif"!)
     + " marks the "
     + ulinkbutton(!"entrance"!) + "to the new museum"
     + example1
);
```

Figure 3: Markup tags and flow layout.

This example creates a box that will be laid out as an HTML page (the *UFlowView::style* brick specifies such a layout). Like an HTML page, it can contain any combination of text, images, links and other brickgets. The *UGroup* brickget applies a bold font to its remaining children but does not have any effect on their layout. Hence, a group behaves very much like an HTML markup tag. A "bold" brickget with the same semantics as the bold tag of the HTML language could be implemented as follows:

```
class UBold !:! public UGroup {
    UBold( const UArgs& a != !UArgs::none ) {
        addlist( UFont::bold + a );
    }
};

UBold&  ubold(!const UArgs&!a!)  {!return *new UBold(a);!}

my_page = ubox( ubold(!"Architect Pei's"!) + ... !);
```

The Ubit architecture is thus not limited to the management of widget-like components but can also model markup tags. The toolkit not only provides "black box" widgets that can read hypertext data but also makes it possible to manage document trees at a fine-grained level. There is no structural difference between a document tree and a GUI tree, all their elements are part of the same global scene-graph and dealt with in the same way. There is no equivalent to *groups* in traditional 2D GUI toolkits.

Boxes, windows and widget-like objects

UBoxes are similar to *groups* except that they can manage one or several *views* on the screen (Fig.!4). *Box views* impose a specific layout and define a clipping area. The difference between boxes and groups is very much the same as between widgets and markup tags. For instance, an HTML bold tag does not interfere with layout (its content

can be displayed on several lines) while a button widget corresponds to a specific area on the screen and imposes a given layout to its children. Widget-like brickgets such as buttons, text fields, list boxes thus derive from *UBox*.

Box views can be visually opaque, transparent or translucent. They can also be transparent to events in certain cases. Each view is controlled by a *view renderer* that controls its layout. Depending on the type of renderer, objects can be arranged in a line or in a column (horizontal and vertical boxes), in a continuous flow (such as HTML pages) or in two-dimensional tables. Objects can also be located on the borders of the brickgets (such as scrollbars that control a viewport) or at arbitrary positions.

Windows are a special case of *boxes*. The *UWin* container is a subclass of *UBox* that serves as a base class for menus, dialog boxes and main frames.

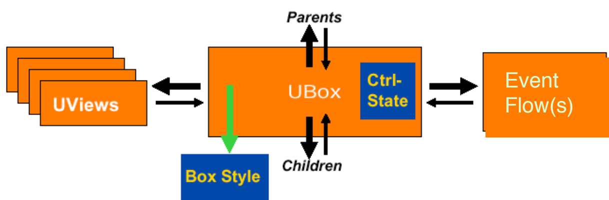

Figure 4: The architecture of a box brickget

Styles and inheritance in the scene-graph

Although they can mimic widgets, standard brickget classes do not have any property nor viewable elements of their own. For instance a *UButton* does not have a label field nor a background or foreground color. The appearance and the contents of brickgets depends on the bricks they contain, the properties inherited from their parents, and *styles*.

Styles provide a different flexibility than inheritance and make it possible to adapt the presentation to a specific platform (for instance a small sized handheld device or a large screen). Styles are compilations of atomic bricks that specify the default appearance of brickget class instances. A style object stores a complete set of graphical attributes for each possible state of the corresponding brickget instance (such as being pressed, activated, dragged, etc.) It can also specify the default content of a brickget and can be context-dependent. For instance, buttons have a specific appearance in menus and menu bars. The rendering algorithm manages style bricks in the same way as if they were actually added to the scene-graph. This algorithm collates style brick values, then overrides them with the values of inherited bricks and brickget children.

Property inheritance is specified in style objects. While this feature exists in document specification languages such as HTML, it has rarely been used in GUI toolkits. The combination of styles and scene-graph inheritance make it possible to control the appearance of a GUI by very few parameters. Implicit dependencies make this feature even more powerful. A single brick can not only control the appearance of the brickgets that depend on it but also those of all their children and descendants. All these objects are automatically updated when the value of such a brick is changed.

Behaviors

Behaviors are not embedded in specific widget classes. They are either common to all brickget classes or implemented as behavioral bricks that can be added to any brickget. Basic behaviors (such as highlighting, activation, selection, selection in a group, a list or a menu, text editing, drag and drop) are controlled by programmable controllers. Such behaviors can be dynamically activated or deactivated by brickget instances (typically in the object constructors). Since all brickgets share the same basic behaviors, any standard interactor can be dynamically transformed into any other. For instance, the following specifications are equivalent from a functional point of view:

```
ubutton(a) = ubox(!UMode::canHighlight!+!UMode::canArm!+!a!)
ucheckbox(a) = ubutton( UMode::canSelect + a )
utextfield(a) = ubox( uedit(!) + a )
```

The *UMode::canArm* brick tells the interaction controller of this brickget that it can be armed when the user presses the mouse on it while the *UMode::canSelect* brick makes it selectable. A brickget controller is equivalent to a finite state machine. The state of each brickget controls its appearance. There is a direct correspondence between states, *UOn* conditions and style specifications.

More sophisticated behaviors (such a text editing, text selection, object selection in lists, various implicit behaviors) are handled by separate brick instances that cooperate with the brickget controllers. These behavioral bricks act as sub-controllers. As seen in the previous examples, these “chunks of behaviors” can be combined together. For instance, the *UEdit* brick makes it possible to edit text in any brickget. Several independent text strings can be contained in the same brickget, even if they are separated by other children. These strings will be seen as a continuous character string from the user point of the view. For instance, the caret will move automatically from the end of a string to the beginning of the following one. Text selection works in a similar way. Text can be selected in all brickgets, even if it is included in their children and descendants.

The following examples (Fig. 5) illustrate the generality of this model. In the first example, menu items contain various interactors, such as editable text fields or buttons included in other buttons. It is thus possible to create generalized menus that contain any kind of interactor (for instance, a file selection box, an active overview of the GUI, etc.) The second example shows an icon box that automatically

adjusts the number of icons in each line according to the width of the icon box. Two features are combined here: a behavior brick that ensures exclusive selection among brickget children and a “flow” layout brick that arranges children in a left-to-right flow. Any brickget can thus behave as a “list” widget, and its items can be of any type. Any brickget can also enforce a flow layout. Its text string children are then automatically wrapped if they cannot fit on a single line (as illustrated in Fig.!3). There is no need for specific text components, as in Swing [9] and other GUI toolkits. For the sake of simplicity, text editing brickgets are provided in the toolkit, but they are just trivial derivations of the *UBox* class.

```
UColor color, red, blue green;
UString font_str!;

UMenu& optionMenu = umenu(
   ubutton( UPix!::colors + "Colors:"
             + ubutton( ubgcolor(!red!)    + " "  + uset(!&color, red))
             + ubutton( ubgcolor(!green!) + " "  + uset(!&color, green))
             + ubutton( ubgcolor(!blue!)   + " "  + uset(!&color, blue))
            )
   + ubutton( UPix!::edit + "Font:"
               + utextfield( font_str + ucall(this, &font_str, setFont!))
               )
   + ubutton( UPix!::book + "Current Page" + umenu( ... )
);

UMenubar& mbar = umenubar(
    ubutton( "File"                + file_menu!)
    + ubutton( "Views"             + viewMenu!)
    + ubutton( "Options"           + optionMenu!)
    + ubutton( UPix!::question + helpMenu!)
 );
```

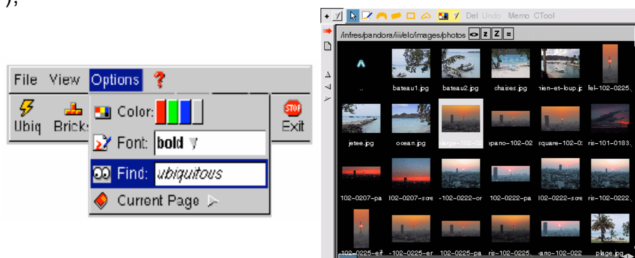

Figure 5: generalized menus and list boxes.

Multiple event-flows and bi-manual interaction

Events are sent to brickget controllers by one or several *event flows* (Fig!4). Event flows receive raw events from a given *event source* (such as a pointer or a keyboard) and perform object picking. There is usually only one native event source for a given computer. However the toolkit can also support alternate event sources. A separate event flow is created for each event source so that events coming from the different sources can be distinguished and controlled in an independent way. Multiple actions controlled by different input devices can thus be performed simultaneously. Hence, the toolkit supports bi-manual interaction and multiple user interaction (each user controlling its own pointer). A multiple mouse server that can manage several cursors on the same screen has been developed for this purpose. This server also makes it possible to control applications from a remote machine or to manage pointers continuously on a set of screen controlled by different machines (as in [6]).

THE EFFECT OF SHARING OBJECTS

A general principle of the Ubit architecture is that all bricks (including brickgets) can be shared by several parents. While some systems support sharing, sharing sophisticated interactive components is a rather unusual feature. The molecular architecture of the toolkit favors the sharing of objects. In traditional widget-based toolkits, features can not be shared because they are embedded in the widget instances. In some cases, sharing is limited to specific cases, such as the “models” of the Swing toolkit [9]. Sharing is used in a more uniform way in 3D toolkits (Interviews [15]) or in some 2D toolkits (Fresco [16], OpenInventor [11], Jazz [4]) but mainly concerns graphical objects and properties, not interactive objects. The generalization of object sharing in Ubit has many advantages:

- multiple views are automatically synchronized. For instance, brickgets that share the same textual elements are automatically updated when the user enters text.
- sharing avoids object duplication and thus reduces the amount of run-time memory. This point is not negligible as GUIs may contain many memory consuming objects (such as images, complex composite widgets, etc.)
- sharing enhances configurability, especially when combined with dynamic inheritance in the scene-graph. A very small number of objects is then sufficient to parameterize the whole interface.

Recursive replication

The Ubit toolkit makes it possible to share brickget sub-trees without any restriction. The semantics of sharing depends on the type of brickgets, as described below.

The viewable children of *UGroup brickgets* are visually replicated in their parents. Since groups are just intermediate nodes in the scene-graph, group children are laid out according to the policy of each parent (or grand-parent if the parent is itself a group).

As for group brickgets, the content of *UBox brickgets* is visually replicated in all parents, but each replication is controlled by a separate view that imposes its own layout. This replication property is recursive. This means that when a brickget tree is shared by N parents, each brickget will then control N separate, but synchronized, views. Besides, some parents (or some children, in the case of a brickget sub-graph) can themselves be shared. Each brickget will thus control as many views as there are possible paths from the root object of the scene-graph (Fig. 6). More precisely, a view corresponding to a given path is only created if all

parents are visible along this path. In any case replication only takes place on brickget views and not on the brickgets themselves.

In contrast with group and box brickgets, *UWin brickgets* (i.e. windows) are not replicated on the screen. This is because it would be meaningless to make the same dialog box appear several times on the same screen. When several parents share windows, the parent/child relationship is used to make them appear automatically at appropriate locations. Structural relationships are thus used to derive implicit behaviors. Boxes can be used instead of windows when visual replication is useful (for instance for creating "internal windows"). The main difference between these two categories of brickgets is their replication semantics.

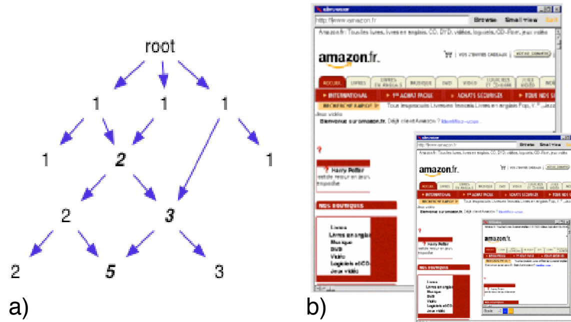

Figure 6: a) recursive replication. The number of views controlled by each brickget is shown on the figure. b) two replicated views (at different scales) of a Web page markup tree.

Parameterized replication

The views controlled by a given widget are not necessarily identical nor even similar. For instance, they will have different layouts if their parents have been resized or if they are managed by different view renderers. Dynamic inheritance in the instance graph also provides a powerful means to parameterize replicated views. Each view can for instance inherit different fonts, colors or scaling factors from each parent. In addition, specifications can be parameterized by *conditional flags* that are inherited in the scene-graph. This makes it possible to generate completely different views from a single specification. For instance, the icons shown in Fig.!5 could be created as follows:

```
UFlag IconBgcolor, DontShowImage, DontShowName;

UBox& icon = ubox(
        + UOrient::vertical + uhcenter(!)      // vertical & centered layout
        + upropval( IconBgcolor )
        + ! DontShowImage!!/  icon_image
        + ! DontShowName!!/  icon_name
        );
```

The *icon_image* and the *icon_name* of each icon will be displayed except if the conditional flags *DontShowImage* or *DontShowName* are inherited. *upropval*!*(IconBgcolor)* represents an optional property brick that can be specified by parents (and will be ignored otherwise). It is thus possible to inherit "named properties" that are identified by conditional flags. For instance, an icon box could specify that its icons should have a black background color and should not display images in the following way:

```
UBox&! iconbox =! listbox(
        UFlowView::style
        + !uflagdef( DontShowImage!)
        + !upropdef( IconBgcolor, UBgcolor!::black!)
       )!;
for ( icon = ... )  iconbox.add(!icon!))!;
```

Conditional specifications make it possible to specify non-local relationships between interactors. This feature shares some similarities with lexical binding in the Lisp language but it imposes stronger syntactical constraints. It should obviously be used with care in order to preserve understandability in the code. For instance, flags should be part of the public API of the brickget classes that are sensitive to them. However, conditional specifications provide a simple parameterization mechanism that is coherent with the scene-graph structure of Ubit GUIs. They make it possible to communalize specifications, and thus, to avoid the duplication of code (with many subtle variants).

Multiple displays

The Ubit toolkit is able to open and manage GUIs on several displays controlled by different computers. Object sharing makes this feature quite powerful as objects can be shared among several screens. This approach does not impose restrictions on the degree of sharing: it is possible to share atomic bricks, brickgets or any subpart of the scene-graph. It is thus possible to show the same windows on several displays, to share some windows but not others, to share subparts of certain windows or just to share constitutive elements such as strings and data structures. As seen above, *conditional specifications* can be used to parameterize the views according to the displays where they are shown. For instance, the views displayed on a small PDA could be displayed with smaller fonts, some unnecessary objects could be omitted on these views, etc.

The main advantage of this centralized architecture is that collaborative applications can be created almost in the same way as single-display applications. As pointed out in [8] very good performance can be obtained under certain conditions (typically, a high bandwidth network, such as a local network, and a limited number of remote displays).

This mechanism (and the Ubit toolkit) is currently implemented on top of the X Window system, but other protocols could also be used. In contrast with X, Ubit property bricks have a level of abstraction that makes them independent from the characteristics of a given display. For instance, a *UColor* brick can work with several displays that do not have the same number of bits per pixel.

ADVANCED FEATURES

Novel interaction and visualization techniques such as semantic zooming, magic lenses, transparent tools, Control menus, bi-manual interaction can easily be implemented by combining the features described so far.

Translucent widgets and Control menus: boxes and windows can be made transparent or translucent by adding a *UAlpha* brick to them. The background of the widget is then alpha-blended with the part of the GUI that is located beneath. Multiple separate transparent layers can also be created in this way. This makes it simple to create transparent dialog boxes, transparent scrollbars (Fig.!7) or Control menus [23], an extension of Pie menus![22] that makes it possible to select an operation and to control it interactively with a single gesture (Fig.!11).

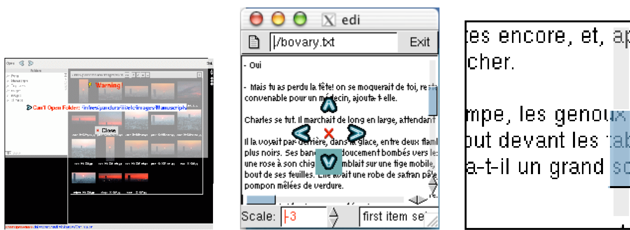

Figure 7: transparent dialogs, menus and scrollbars.

Zoomable interfaces. Widgets can be zoomed dynamically by adding a *UScale* brick to them. This scaling attribute also applies to widget children. *UScale* bricks can appear at several levels of the instance graph. They are always relative to the current level of scaling of the parents. Scaling can thus take place on the entire interface or any of its subparts. This feature can also be used to display several views with different scales:

```
UScale small, large;
UBox& overview = !ubox( small + my_page );
UBox& large1     = !uscrollpane( large + my_page );
UBox& large2     = !uscrollpane( large + my_page );
```

The *my_page* brickget and its children will be displayed three times: in a box that shows a small-size overview and in two independent scroll panes. The scaling of both scroll panes can be dynamically changed and will be synchronized as they share the same scaling brick.

Figure 8: Two multiple-scale interfaces

Semantic zooming can be obtained by using conditional specifications that specify scale ranges:

```
usrange(-10,!-4) !/! version1
+ !usrange(-3,!3) / !version2
+ !usrange(4,!10) / !version3
```

Depending on the current level of scaling one of the three versions of the widget will be displayed. This mechanism is recursive: each version can contain sub-versions that depend on other scale ranges and so on. The toolkit updates the display automatically when the level of scaling of a subtree is changed. This mechanism is quite efficient for creating zoomable interfaces. This illustrates a motivation of this work: the Ubit toolkit is not dedicated to a specific visualization technique (some specialized toolkits such as Jazz [4] or CPN2000 [2] provide more advanced features for creating ZUIs) but its architecture makes it possible to create advanced interfaces by combining the standard features of the model.

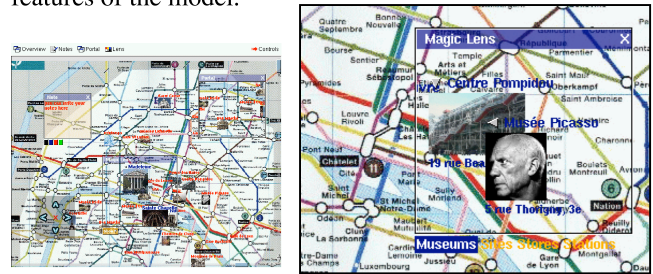

Figure 9: Magic lenses on a sensitive map.

Magic lenses. Magic lenses are visual filters that change the appearance (or the behavior) of objects that are located beneath them. They are typically used to provide alternate representations or complementary information. They can be active or passive: passive lenses just reveal hidden data while active lenses completely change the representation. Sharing and conditional specifications make it easy to create both types of lenses. The lens shown at Fig. 9 could for instance be created as follows:

```
UFlag stores, museums, stations;
UBox& scene = !ubox( uima(!"paris-map.jpg"!)
                     + !stores     /  ugroup(!....!)
                     + !museums /  ugroup(!....!)
                     + !stations   /  ugroup(!....!)
                   );
stores_lens != !upane( uflagdef(!stores!) + scene!);
uscrollpane(!scene + store_lens!);
```

This will create two superimposed views: a view of the *scene* that only displays an image and a view of the *store_lens* that also contains all the objects conditioned by the *stores* flag. The *store_lens* brickget is a viewport (*UPane*) that only displays a subpart of the scene and that can be moved interactively over the image. A magic lens effect can be obtained by synchronizing the location of *store_lens* with the area shown in its viewport.

Transparent tools. Transparent tools are used to perform alternate operations on the interactors that they cover [5] [25]. The toolkit provides two ways for implementing such tools. Active tools retrieve the object that is underneath and performs an operation on it. Passive tools are transparent to events but modify them in an appropriate way. They do not perform any operation by themselves but define a protocol that modifies the actions of other interactors. It is up to these interactors to react in a specific way if they are aware of this protocol. From an architectural point of view, this scheme avoids complex interactions between objects and makes it simple to create such tools.

Figure 10 shows an example of a transparent tool and a transparent note editor. These objects are created by combining standard brickgets (that are made translucent by adding *ualpha*() bricks to them). The transparent *Cross tool* modifies events in two different ways: it adds a *flag* to the event and changes its location so that the object located beneath the cross will get the event that is produced when a button of the tool is clicked.

Figure 10: An annotating tool using transparent tools

RELATED WORK

One of the main goals of the molecular architecture is to enhance the flexibility of the toolkit. This was also one of the goals of the Amulet toolkit [20]. Ubit is quite different as it is not based on a prototype/instance system. One important advantage of the Ubit architecture is that it only relies on standard features of a general-purpose programming language and thus allows safe type-checking at compile time. The fact that no interpretation takes place at run-time may also improve performance.

In contrast with Amulet or SubArtic [26], the Ubit architecture does not provide constraint solvers but implements a simplified form of constraints called dependencies. Dependencies are less general but they provide an efficient means to control and synchronize multiple objects. The combination of this feature with scene-graph inheritance makes it even more powerful as trees or sub-graphs of related objects can be automatically synchronized.

Procedural programming languages have not been used for expressing GUIs in a declarative way except in our former XXL system [12]. However, this idea has been used in other domains, such as in the QOCA constraint-solving toolkit [24]. The Ubit syntax also resembles a formal specification. Its conciseness and the higher level of abstraction it provides may offer better understandability and help the programming of complex interfaces. A preliminary version of this approach was presented in [13].

The sharing of objects is supported by previous systems such as Interviews [15] and Fresco [16] and is rather common in 3D toolkits such as OpenInventor [11]. But the "molecular" architecture generalizes this principle since all GUI objects can be shared, including interactors. This point has a major impact on the way applications are designed, especially for the synchronization and the configuration of GUI components. This feature offers very interesting properties, such as the recursive generation of multiple views that are synchronized – although not necessarily similar. It also serves as a basis for novel interaction and visualization techniques. Brickget "molecules" can be seen as "abstract" representations in the sense that they do not control the screen area directly. As in the MVC model, the views are completely separated from the interaction controller. This architecture can be seen as an extension of MVC where each brick would be a separate "model".

Several GUI toolkits that provide advanced interaction and visualization techniques have been developed in recent years. Jazz [4] and CPN2000 [2] use a scene graph for implementing 2D GUIs. The molecular architecture is an intermediate approach between classical widget-based and scene-graph toolkits. A complex sub-hierarchy can be seen as a single element in the global instance graph of the GUI. Hence, the Ubit DAG follows a recursive model where scene graphs can be embedded. This feature may ease programming (by avoiding handling large scene graphs) and may be less confusing for programmers that are accustomed to classical 2D toolkits. Moreover, Ubit is a general-purpose toolkit that proposes a new architecture for designing interactors, while Jazz relies on Swing widgets and an embedding mechanism. CPN2000 is based on the notion of "instrumental interaction" [1]. The reification of behaviors in the Ubit toolkit is a similar (but currently less powerful) approach.

IMPLEMENTATION

The toolkit has been implemented on top of the Xlib, but it follows a layered architecture so that most of the code does not depend on a specific platform. Optionally, the Open GL graphical engine can be used instead of X. This latter implementation is not optimized but it demonstrates the feasibility of the approach. Alpha blending and double buffering are performed internally by the toolkit if these services are not provided by the windowing system. Windows can either be rendered as *hard windows* that use the windows of the underlying windowing system or as *soft windows* that are drawn directly on the main window. This feature is useful for Open GL rendition and for improving alpha blending when the Xlib is used. The toolkit has been

tested on a variety of Unix platforms (Linux, SunOS, Mac) including an embedded version of Unix for the Ipaq PDA.

Our informal tests show that the performance of the X implementation compares well with other C or C++ GUI toolkits. For instance, a refreshing rate of 65 fps (30 fps with internal double buffering) was obtained on an Ultra Sparc-60 with a 450 MHz processor by forcing the icon box shown on Fig.!5 (with thirty 60x60 images) to redisplay itself continuously (normally, views are only repainted when they are damaged). The size of the toolkit is relatively small (about 25 000 lines of C++ code including header files). The size of its dynamic library is less than 2 Mo on a Unix system (as an example, the libraries of Qt and Gtk, two popular Unix GUI toolkits, are five times larger).

CONCLUSION

A new GUI toolkit, based on a "molecular architecture" is presented in this paper. It enhances flexibility by means of a very dynamic model and offers a higher level of abstraction for GUI specifications. This model also makes it possible to program advanced interaction and visualization techniques in a simple way.

The Ubit toolkit has been used for developing various students' projects at our institute, such as tools for editing annotations on hypermedia documents [14] or the GUI of the *VReng* [28] virtual reality engine (Fig.!11). The (superficial) similarity with classical widget-based toolkit seems to help novice programmers. They also seem to be comfortable with the pseudo-declarative API and to appreciate this feature. The source code of the toolkit, some examples, demonstration programs and videos are available from: www.enst.fr/~elc/ubit.

Figure 11: The Vreng GUI (with a Control menu![23])

ACKNOWLEDGEMENTS

The author would like to thank Michel Beaudouin-Lafon, Scott Hudson and the anonymous reviewers for their helpful comments.

REFERENCES

1. Beaudouin-Lafon M. Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. *Proc CHI* 2000, ACM, 446- 453.
2. Beaudouin-Lafon M. The Architecture and Implementation of CPN2000, a Post-WIMP Graphical Application, *Proc. UIST* 2000. ACM, 181-190.
3. Bederson B.B., Hollan J.D., Perlin K., Meyer J., Bacon D., Furnas G.W. *Pad++:* A Zoomable Graphical Sketchpad for Exploring Alternate Interface Physics. *Journal of Visual Languages and Computing*, 1996, 7(1), 3-31.
4. Bederson B., Meyer J., Good L., Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java, *Proc UIST* 2000.
5. Bier E.A., Stone M.C., Pier K., Buxton W., DeRose T.D. Toolglass and Magic Lenses: The See-Through Interface, *Proc. SIGGRAPH*, ACM, 1993, 73-80.
6. Booth K.S. et al. The "Mighty Mouse" Multi-Screen Collaboration Tool, *Proc UIST* 2002, 209-212.
7. Card S.K., Mackinlay J.D., Shneiderman B. *Readings in Information Visualization "Using Vision to Think"*. Morgan Kaufman, 1999.
8. Chung G., Dewan P. Flexible Support for Application-Sharing Architecture. *Proc ESCSW*, 2001, Kluwer, 99-118.
9. Fowler!A. *A Swing Architecture Overview.* http://www.javasoft.com/products/jfc/tsc
10. Gamma E. et al. *Design Patterns*, Addison-Wesley, 1995.
11. Inventor. http://www.sgi.com/Technology/Inventor/
12. Lecolinet E. XXL. A Dual Approach for Building User Interfaces. *Proc. UIST*, ACM, 1996, 99-108.
13. Lecolinet E., A Brick Construction Game Model for Creating Graphical User Interfaces. *Proc. INTERACT* 1999. 510-518.
14. Lecolinet E. Robert L., Role F. Text-image coupling for editing literary sources, *Computers and the Humanities Journal*, 2002. Kluwer. 36(1), 43-73.
15. Linton M., Vlissides J.M., Calder P.R. Composing User Interfaces with InterViews. *Trans. IEEE Computer*, 1989, 22(6), 8-22.
16. Linton M., Tang S., Churchill S. Redisplay in Fresco. *The X Resource,* 1994, (9), 63-69.
17. Maloney J.H., Smith R.B. Directness and Liveness in the Morphic User Interface Construction Environment. *Proc. UIST*, 1995, ACM, 21-28.
18. Myers B.A. Challenges of HCI Design and Implementation, *ACM Interactions*, 1994, 1(1), 73-83.
19. Myers B.A., Hudson S.E., Paush R., Past present and future of user interface software tools,!*ACM ToCHI,*!2000,!43(3),!82-89
20. Myers B.A et al. The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE trans. on Software Engeenering*, 1997, 23(6), 347-365.
21. Ousterhout!J., *Tcl!and!the!Tk!Toolkit*. Addison Wesley, 1994.
22. Pie menus. http://www;piemenus.com
23. Pook S., Lecolinet E., Vaysseix G., Barillot E. *Control* Menus: Execution and Control in a Single Interactor. *Proc. CHI* 2000, ACM, 263-264.
24. QOCA: http://www.csse.monash.edu.au/projects/qoca
25. Hudson S.E., Rodenstein R., Smith I. Debugging lenses: a new class of transparent tools for user interface debugging, *Proc. UIST* 97, 179 – 187.
26. Hudson S.E., Smith I. Ultra-Lightweight Constraints. *Proc UIST 96*, 147-155. http://www.cc.gatech.edu/gvu/ui/sub_artic.
27. UIML. http;//www.uiml.org
28. Vreng: http:/www.enst.fr/~dax/vreng