# Revisiting visual interface programming : creating GUI tools for designers and programmers

Stéphane Chatty, Stéphane Sire, Jean-Luc Vinot, Patrick Lecoanet, Alexandre Lemort, Christophe Mertz

# Revisiting Visual Interface Programming: Creating GUI Tools for Designers and Programmers

*Stéphane Chatty*[1] *Stéphane Sire*[1] *Jean-Luc Vinot*[2]
*Patrick Lecoanet*[2] *Alexandre Lemort*[1] *Christophe Mertz*[1,2]
[1]IntuiLab
Prologue 1, La Pyrénéenne
31372 Labège Cedex, France
{chatty,sire,lemort,mertz}@intuilab.com
[2]CENA
7 avenue Edouard Belin
31055 Toulouse Cedex, France
{lecoanet,vinot}@cena.fr

**ABSTRACT**
Involving graphic designers in the large-scale development of user interfaces requires tools that provide more graphical flexibility and support efficient software processes. These requirements were analysed and used in the design of the TkZinc graphical library and the IntuiKit interface design environment. More flexibility is obtained through a wider palette of visual techniques and support for iterative construction of images, composition and parametric displays. More efficient processes are obtained with the use of the SVG standard to import graphics, support for linking graphics and behaviour, and a unifying model-driven architecture. We describe the corresponding features of our tools, and show their use in the development of an application for airports. Benefits include a wider access to high quality visual interfaces for specialised applications, and shorter prototyping and development cycles for multidisciplinary teams.

**KEYWORDS:** visual design, vector graphics, SVG, software architecture, GUI tools, model-driven architecture

**CATEGORIES:** H5.2 [Information Interfaces and presentation] User Interfaces — GUI; D2.11 [Software engineering] Software Architectures.

**GENERAL TERMS:** design, human factors, languages

**INTRODUCTION**
The work of graphic designers in user interface projects was popularised with the Web and desktop interfaces [14]. With the growing understanding that this work can improve users' performance and acceptance of new products, it is now sought in the design of specialised user interfaces, from aircraft cockpits to plant supervision systems. For instance, Figure 1 illustrates a graphic designer's work for air traffic control.

This evolution raises an important engineering issue: how can companies design and produce such design-intensive software at reasonable costs? For standalone applications with no complex functional core, tools such as Flash and Director allow designers to produce high quality products. But for more complex software, programmers and designers still have to choose between cost and flexibility. On the one hand, pre-designed widgets offered by interface builders are simple to assemble. However programmers often produce poor layouts, and designers resent the low control over the look and interaction style. On the other hand, graphical libraries offer greater flexibility: programmers can reproduce designs provided by graphic designers within certain limits. But even then the limits are sometimes too strict for designers, and the cost of re-coding their work is too high to support both iterative design and a profitable industry.


Figure 1: Visual techniques, when used by graphic designers, can enrich the message conveyed.

Efficiently involving graphic designers in the production of interactive software calls for tools that meet the requirements of both graphic design and software engineering. It requires graphical capabilities that match the working methods of designers. It also challenges the architecture of common user interface tools and the notion of predefined widgets, which were introduced solely for programming purposes.

This article describes these two sets of issues and the solutions that we implemented respectively in TkZinc and IntuiKit. TkZinc is a graphical library provided as a replacement for the Canvas component in the Tk toolkit [15]. It is available at www.tkzinc.org. Co-designed with a graphic designer, its features are aimed at providing programmers with graphical capabilities and concepts that give them a common vocabulary with designers. IntuiKit is an interface design suite aimed at making the prototyping and development of multimodal user interfaces more accessible to industrial

companies. Its model-driven architecture [10] supports software engineering processes that involve iterative design and designers from multiple disciplines. In this paper, we focus on the features of IntuiKit that are aimed at graphic designers: the use of the Scalable Vector Graphics (SVG) standard as the graphical model of the environment, and an architecture that focuses on combining the graphical facet of interactive components with the other facets that are produced by programmers. We then illustrate the use of IntuiKit on a real application, and discuss implementation details, related work and the limitations and perspectives of our work. All illustrations in this paper except diagrams are screen shots of applications created with TkZinc and IntuiKit.

## MATCHING DESIGNERS' WORK

*Art does not reproduce the visible; rather, it makes visible.*
*Paul Klee, 1920*

Producing efficient visuals is not just a matter of good taste and having tools with a rich graphical model. Graphic designers are trained to use methods evolved by artists over centuries, whatever the media and tools. Understanding those methods helps selecting which features are most needed by designers and in what form. That is what guided the development of TkZinc: the goal was to make user interface design an experience similar to illustration design, notwithstanding the fact that user interfaces are dynamic programs.

Being implemented as a Tk widget, TkZinc benefits from standard GUI mechanisms: windowing, event management, etc. The originality of TkZinc lies in its graphical primitives, chosen and organised as a mix of GUI programming libraries and drawing tools such as Adobe Photoshop and Illustrator. All other features, such as the programming interface or distribution of events to graphical items, are modelled after the Tk Canvas. The main graphical characteristics of TkZinc are vector-oriented graphics and visual effects, a 2D scene tree where groups play an important role, and a graphical model that includes shape construction, gradients, clipping, transparency and shading. Some of these features are classical and available in standards such as SVG, whereas others are more radical or only available in 3D tools such as OpenGL. In this section we show how they were chosen, through the observation of visual designers.

When working on a display or a user interface, a designer takes several types of information into account:

- the characteristics of the media to be used, for instance the size and resolution of a screen;
- the data to be displayed, which is the main focus of software engineers;
- context information, provided by observation or more often by marketing or internal communication departments: what is the message to be conveyed to users (as opposed to mere data display);
- efficiency considerations, based on psychological laws or know-how.

Designing a display starts with a holistic approach: building a global picture that conveys the desired message. The goal is to *evoke* things known to the user and not to reproduce them, by tricking the spectator's eye if necessary. Once this global picture is defined in their mind, designers work on solutions to reproduce it, using the available range of visual elements and techniques. We now examine in more details the techniques used, and how TkZinc supports them.

### A palette of visual techniques

Artists and designers use many ways to convey the meanings they desire: contrasts, harmonies, light, etc. Combining these signals allows them to communicate data, but also context, culture or feelings, which contribute to the global interface usability and performance. The ideal designer's palette is made of shapes (including type), colours, textures, light and rhythm. In practice, they have always had to approximate it with available tools, whether physical or digital. For that reason, we observed that designers sometimes use graphical tools in unexpected ways for programmers. This has guided the design of many parts of TkZinc.

*Revisiting shapes* Although our understanding of the world is made of shapes, our perception of it is not. Usually, artists do not use "graphical objects" to produce shapes. Graphical designers sometimes do, when they want to produce high contrast and visual simplicity. But generally, they rather see them as "construction lines" used to apply visual effects such as color strokes, shadows or light which produce the desired perception. For this reason, TkZinc offers shapes that can be used both as graphical objects or as geometrical templates. Furthermore, with graphical models based on objects, designers often lack intuitive notions such as holes and clipping. For that purpose, TkZinc provides two complementary solutions. *Shape operations* make it possible to build shapes by assembling or subtracting other shapes, thus providing solutions for holes. *Clipping* is also available: a shape can be used as a clip mask for any graphical object or group.

Figure 2: Complex gradients used to create a rubber-like appearance (left) or to simulate a 3D design (right)

*Revisiting colour* Perception is based on contrast, and colour contrast is the easiest to create. However, designers often want to use gradients rather than mere colour juxtaposition. This is why TkZinc provides a *rich gradient model*, where gradients can be defined from shapes. That includes linear and radial gradients, but also conical gradients, and gradients computed from arbitrary paths. Figure 2 illustrates how gradients can be used to simulate shapes, perspective effects, and shadows. The button on the left is made of a circular black arc filled with a conical gradient, then cloned, translated and filled with a path gradient. Most shapes and shadows on the right are built with similar techniques.

*Rhythm* A complement to materials is visual rhythm, provided for instance by repeated pencil lines or strokes on a drawing. Rhythm can be provided by filters applied to images or by line textures (the richer version of dashed lines). It

can also be produced by the cloning and layout of visual objects along "networks". TkZinc provides *geometrical grids* that automatically clone and translate shapes and gradients. For instance, the top of figure 3 is made of one object built by mapping a square path on a grid, filled with a pixmap texture.

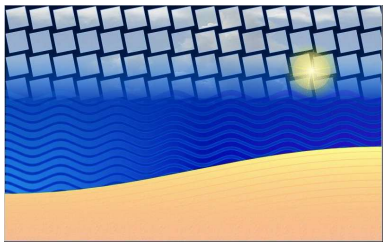

Figure 3: Rhythm can be obtained by repeating shapes along networks and combining them

*Light* The contrast between light and shade is an important way of signalling information as well as driving the order in which users read displays. It is also a way to structure the visual space, make it consistent and create 2-1/2D effects. The designer in our group likes to manage light as a "skin" added on top of shapes, colours and textures. For that purpose, TkZinc offers a combination of *groups and transparency*, as do recent versions of Adobe Illustrator. Objects can be hierarchically organised as groups, and a transparency can be associated to any group, making it behave as a semi-transparent layer. Figure 1 makes heavy use of groups and transparency.

**Image construction**

The above techniques are basic blocks for designers' work. However, some images cannot be obtained with a single operation. Oil painting, for instance, involves the progressive construction of pictures with several layers of paint. Designers are used to building images in several successive operations. Several features of TkZinc support this process, sometimes perceived as sub-optimal by programmers: rather than being pre-computed, images are recomputed during the interaction, which allows designers to manage interactive displays while keeping the appropriate graphical control.

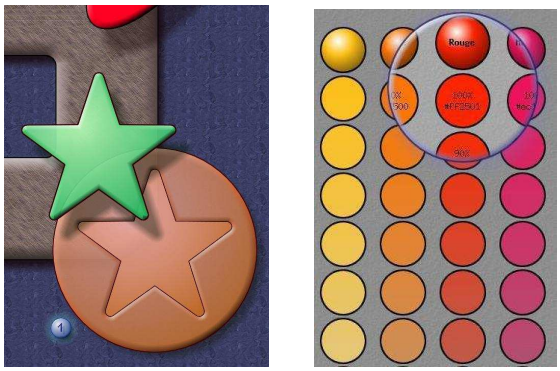


Figure 4: Hierarchical groups, simple transformations, clipping, shape operations and gradients help shape the scene light and provide great expressiveness

*Shape operations* Building some shapes is easier through the combination of simpler shapes. For that purpose, TkZinc offers a set of algebraic operations on shapes: addition, subtraction, intersection. Shapes are successively built, filled with textures or gradients, used to build other shapes, etc. Objects such as those shown on the left of Figure 4 can also be built by assembling triangles and shading the resulting tessellation.

*Hierarchical structure* The successive operations must often be applied to sets of objects rather than just one object. The combination of groups, transparency and clip masks that we described earlier plays an important role in this process. The picture on the right of Figure 4 is made of four shapes created from seven circles and a gradient each, which are grouped and cloned. The cloned group is translated and scaled, then clipped to produce the "magic lens" effect.

*Attenuation and blending* The iterative construction of images involves sequences such as making an operation then subtly attenuating it on some parts of the display, or making two shapes and merging them with a superimposed layer that blends the contours. Groups, gradients and transparency can be used for that purpose. Filters such as blurring are not yet available in TkZinc but would also be desirable.

**Parametric, interactive displays**

Many of the above features are available, though often in different forms, in drawing tools, graphical libraries, SVG players, and in some UI toolkits. However, one should distinguish between drawing and creating representations. What is needed to create high-quality user interfaces is a set of objects that can be computed from application data, modified at run-time, and used to capture users' action. Dynamicity is central, and TkZinc allows designers to create dynamic interactive objects, not just draw.

*Extreme vector graphics* There is a debate in the UI toolkit community about bitmap and vector graphics: the former provide pixel-precise control to designers, whereas the latter offer scalability. The design of TkZinc represents a radical position in this debate: TkZinc aims at being a vector-only toolkit (with a slight concession to imported textures and icons). Graphical objects are made of vectors of course, but gradients, clips, and grids are also based on geometrical vectors and can be changed at run-time. Similarly, all vertices of paths and curves can be accessed and changed individually. No new feature is added to TkZinc if it cannot be parameterised and related to the geometrical model. The reason for this choice is interactivity: all graphical constructs must be able to be bound to evolving data or interaction, and thus be recomputed at run-time.

*Object selection* Providing feedback often implies changing the visual attributes of several objects at the same time. It is not always possible to put those objects in one group, because they may be grouped with other objects for other purposes: clipping, blending, etc. Another way of addressing several objects must be provided. TkZinc provides a system of tags inspired from Tk [15]: an object can have several tags and it is possible to apply an operation to all the objects that share a given tag. We will see that in IntuiKit this system is extended to full XPath queries.

## GRAPHIC DESIGN AND SOFTWARE ENGINEERING

Augmenting the palette offered by GUI tools to designers is an important first step. However, in many contexts this is not enough. First, this leads to a linear production process: graphic designers do their work, then programmers take over. Second, this involves a duplication of effort: programmers have to reproduce part or all of the graphics. We observed that these two factors often prevent software development groups from involving graphic designers, because of the related cost and project management issues.

Wider acceptance of iterative design and graphic design requires solutions that preserve the ability to redesign graphics, and avoid duplication of effort. The direction we chose in the design of IntuiKit was to explore software engineering processes that give graphic designers a more central role in the production of software, while preserving the ability of programmers to structure their code appropriately. Designers become software producers. Their artwork becomes a new type of software component that can be managed in its own way then merged with other components, just like software components obtained from different source files have to be merged by compilers and link editors in traditional software engineering.

IntuiKit is a software suite aimed at designing and prototyping multimodal user interfaces, then transferring them to industrial production. It is not centered around a modality but rather organised as an execution environment for models of software components, which makes it closer in many respects to a language interpretor than to a GUI toolkit. With IntuiKit, a user interface is the result of the instantiation and combination of several models. These are all provided by extensions to the core software component model that is used both for structuring programs and merging models. The set of models used depends on the type of interface and the preferred modelling approach: for instance, models of graphical objects and behaviours for GUIs, or speech and grammar rules for speech interfaces. These models can be built and managed independently, each with the appropriate tools. They are then loaded by IntuiKit from XML files or instantiated through a programming interface, linked together with application code or non-modelled UI code, and "executed".

In this section, we describe and illustrate the parts of IntuiKit that are central to the involvement of designers in the software engineering process: the general architecture and software component model, the SVG graphical model, the discrete behaviour model, and how they are combined to produce applications.

### Architecture and components

The central structure of an IntuiKit application is a tree of Elements and Components. *Elements* are the nodes of the tree, and the basic blocks of all models: graphical objects, windows and behaviours are Elements. The execution of the application consists of a series of traversals of the tree, in which elements are activated. Initialisation, rendering and control flows triggered by events are example of such traversals. A special kind of Element is the Reference, which allows shared objects in the tree. Elements can be declared as *models*, making them insensitive to rendering traversals. Elements can be cloned, thus giving IntuiKit some features of prototype languages. Elements can be loaded from XML files. They can be parameterised through a *property* mechanism: they export the names of their properties, which can be set from Cascading Style Sheets (CSS) files [13]. References can be defined using XPath expressions [20] to select their target.

*Components* are Elements that contain children. They implement encapsulation and parameterisation through a namespace system. All element and property names exported by children are visible to their siblings. These names can in turn be exported (with an optional renaming) by the parent Component to its own siblings and parents. Properties can also be *merged* by the parent, which means that two sibling elements will share the merged property. Control in IntuiKit relies on a classical event model, in which any Element and Component can emit events. Programmers can implement their own Components in native code. The IntuiKit tree can be understood as an extended scene graph. We prefer to interpret it as the abstract tree of a language, because it is aimed at structuring software more than rendering graphics.

The core of IntuiKit only provides structuring and communication mechanisms. All interactive capabilities are provided by IntuiKit extensions; defining an extension consists of defining new Element types and their semantics when rendered. Element types can be taken from existing standards, as are the SVG elements for graphics or VoiceXML for speech. They can also be defined specifically, as are the finite state machines that are used below for defining the basic discrete behaviour of interactive components.


Figure 5: Henry-the-Frenchie

### SVG: turning artwork into software

SVG, a recommendation by the World Wide Web consortium, is an XML standard for exchanging vector graphics. Its object-oriented graphical model covers an important part of the requirements described earlier in this article, which makes it a good candidate for storing user interface graphics. SVG is also convenient as a graphical file format: it can be generated from graphic design tools such as Adobe Illustrator or Corel Draw. For these reasons, and despite some disparities with the features described earlier, the GUI module of IntuiKit implements the SVG model, using TkZinc as its graphical rendering engine. SVG elements are considered as IntuiKit Elements, and SVG groups as Components. IntuiKit offers two equivalent ways for manipulating graphics: by loading SVG files or through a programming interface. Using files, one can manage graphics as an independent part

of the software, produced for instance with Adobe Illustrator by saving the artwork in SVG format.

The two fragments of Perl code below show how the GUI module of IntuiKit is used. The first example uses the programming interface and opens a window that displays a text. It illustrates how the position of Elements in the tree influences the result just like in a scene graph: the window (Frame), font and text are added to the root Component in such an order that the text appears with the appropriate font in the window.

```
$root = new Component;
new Frame ( -parent => $root );
new Font ( -parent => $root, -family => 'Helvetica' );
new Text ( -parent => $root, -text => 'EAT!!!' );
$root->run;
```

The second example illustrates the use of an SVG file. It acts as a basic SVG player, by adding a window and an Element to a root Component. This is illustrated in Figure 5 with the SVG file from a very simple application that we will use in the rest of this section: a digital pet inspired from a popular electronic toy. The pet, named Henry-the-Frenchie, is totally useless and has a simple behaviour: it grumbles when hungry and can be fed by pressing a button.

```
$root = new Component;
new Frame ( -parent => $root );
$henry = load Element ( -parent => $root,
                -file => 'henry.svg' );
$root->run;
```

Considering structured graphics files as a software component allows designers to deliver successive versions and thus to work in parallel with programmers. However, conventions must be established. IntuiKit uses Element names for that purpose: just like function or component names must be determined in advance between programmers, SVG element names must be agreed upon by designers and programmers. As an example, Figure 6 shows an extract of the Adobe Illustrator palette that represents the artwork for Henry-the-Frenchie. These layers and groups are saved in SVG as a tree of groups with the names that were set in the palette. Consequently, the palette reflects the contract passed between the designer and the programmer.

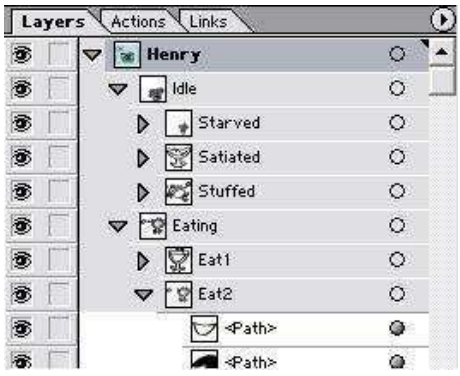


Figure 6: Structure of the SVG file for Henry

**Bringing graphics to life**

Programmers must be able to capture events on graphical objects and change their appearance upon user events or messages from the functional core. For instance, Figure 7 illustrates event flows in the Henry-the-Frenchie application. The "start eating" and "stop eating" events are generated when the user depresses and releases the "eat" button. A simplified functional core (FC) manages an internal numerical value that grows while the button is depressed, then decreases periodically. It notifies "increase" and "decrease" events that correspond to Henry's status changing from starved to satiated or stuffed and back.

Describing behaviour is a different facet than graphics. The core module of IntuiKit provides an event system based on a classical binding mechanism. A binding associates an action with an event source (clock, graphical object, etc) and a specification (button click, key press, etc); a binding can be dynamically enabled or disabled. The GUI module that implements the SVG model defines graphical objects as event sources like the Tk Canvas: when the user clicks, the first visible object under the mouse with an active binding has its binding triggered. Bindings on groups can be specified as "atomic" so that events are detected in all elements of a group. This binding system provides the foundation for programming behaviour as a set of callback functions, or for using more elaborate models.

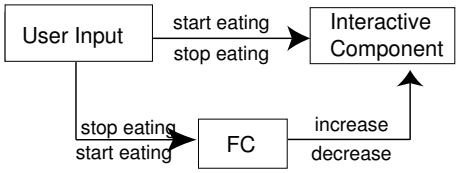


Figure 7: Event flows in Henry

The literature provides various models of discrete or continuous behaviour in user interfaces, such as StateCharts, finite state machines, Petri nets, data flows, constraints, etc. Any of these models could be offered as a set of Elements in IntuiKit, provided that the appropriate semantics are defined and implemented (some of these models may require improvements to the core of IntuiKit). For discrete behaviours, IntuiKit currently implements finite state machines (FSMs). Although its limits are well known, this model is rich enough to define basic interactive components, and it illustrates how such a model can be combined with graphical Elements. Continuous behaviours such as scaling or translation are also available; they rely on the ability of the GUI module to change the attributes of SVG elements once they are instantiated.

*Finite state machines:* IntuiKit defines FSMs as a set of states and a set of transitions labelled with event specifications and actions. When in a state, a binding is activated for each outgoing transition. Actions include traditional callbacks and event emission. Event specifications can refer to all Element names that are visible from the FSM (the names exported by its siblings). This makes it possible to associate behaviours to events occurring on graphical Elements defined in the same Component.

IntuiKit FSMs are local to their parent Component. They can be used to store its state and to emit events or trigger callbacks when certain sequences of events occur. They can also be used for basic graphical feedback by triggering the display and hiding of graphical Elements that are associated to its states. This takes advantage of a method often used by graphical designers who use Adobe Photoshop to build behaviours in Web pages: they store the different states of their objects into different layers and manually simulate the transitions by turning the visibility flag of the different layers on and off. IntuiKit offers this possibility by using property merging and the Switch Element.

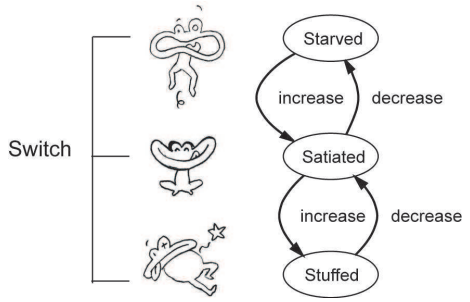


Figure 8: The FSM-Switch pair for Henry's stomach

*Switches:* In SVG, a Switch is a special group that renders only one of its children depending on the value of a variable of the run-time context, such as the preferred user language. IntuiKit extends the semantics of the Switch, by defining an implicit property named "branch" which controls which child of the Switch will be rendered. If the Switch is present in the same Component as a FSM, one can synchronise the Switch and the FSM by merging the "branch" property of the Switch and the "state" property of the FSM: when the FSM changes state, it triggers a partial traversal of the tree, and the Switch is rendered according to its new active branch.

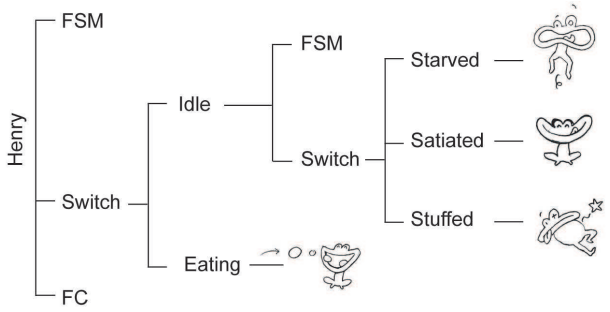


Figure 9: The complete IntuiKit tree for Henry

Figure 8 shows the FSM-Switch pair used for Henry's stomach. The FSM receives "increase" and "decrease" events from the FC. It is associated with a Switch that activates the group containing the appropriate picture for Henry. Figure 9 gives a more complete view of the tree, and shows another aspect of combining models: Switches can actually control any type of Elements, not only SVG ones; in the case of FSMs, the semantics of "being hidden" is to be disabled, which means that no event binding is activated. This provides control similar to that of hierarchical FSMs, and it is used at two levels of depth for Henry: Idle or Eating, then Starved, Satiated or Stuffed when Idle. The code fragment below illustrates the creation of the tree with XPath references, FSMs, including event specification, and the merging of properties which synchronises a FSM and a Switch.

```
$eat_model = $svg->find( -ref => './/Eating' );
$eat_model->clone( -parent => $Henry );
...
$switch = new Switch( -parent => $Henry,
                -branches => { 'eating' => '../Eating',
                    ...
$fsm = new FSM( -parent => $Henry,
        -states => [ 'eating', 'idle' ],
        -transitions => {
                {-from => 'idle', -to => 'eating',
                -on => [$eat_button, 'StartEating']},
               ...
$Henry->merge( -names => [$fsm->state, $switch->branch] );
```

### Adjusting parameters

User interface environments must support the management of parameters: colours, text labels, etc. In IntuiKit, this is obtained with CSS: all properties defined by Elements can be set from a style sheet file. For instance, let us assume that the Functional Core (FC) of Henry is implemented in Perl as a Component, and that the values of its "appetite" and "digestion" control its internal algorithms. The corresponding properties would then be defined as follows:

```
package FC;
use Component;
sub new {
    my $self = new Component;
    $self->define( -name => 'appetite', -default => '10');
    $self->define( -name => 'digestion', -default => '5');
    ...
```

Then, the functional component could be created as follows:

```
$root = new Component ( -name => 'Henry' );
$fc = new FC ( -parent => $root, -name => 'fc' );
```

And finally, the properties could be changed with the following style sheet file:

```
Henry/fc {
    appetite: 50;
    digestion: 44;
}
```

## IMPLEMENTATION

TkZinc is a free software graphical library distributed by CENA since 1998. Aimed at prototyping, it is built as a widget in the Tcl-Tk environment, and can be used from the Tcl, Python and Perl scripting languages. Its core structure is coded in C and originally based on the X Window System. Many advanced visual features of TkZinc require OpenGL. TkZinc runs on Windows, Linux and Mac OSX. Initially developed for safety-critical displays, TkZinc has good CPU performance and reliability: earlier versions of

TkZinc were used in operational air traffic control workstations from 1998 to 2003 for interactive displays composed of thousands of graphical objects. Benchmarking against the Java-based Batik SVG player revealed that TkZinc was more than 10 times faster when loading and displaying a figure made of 2600 curves amounting to 84000 vertices.

IntuiKit is developed and distributed by IntuiLab since 2003. It offers a prototyping environment programmed and accessible in Perl, that has been used to develop car displays, touch-screen-based workstations for various domains, as well as prototype multimodal applications. It runs on Linux and Windows. A C port accessible in C++ and Java and aimed at small devices and production code is under development.

## EXAMPLE APPLICATION

IntuiKit is used to develop prototypes or pre-operational products in various domains: automotive, aerospace, manufacturing, telecommunications and defence. This has given us the opportunity to test the concepts described in this article on real applications. Four graphic designers were involved in the projects, one at a time. Some had no previous experience of working with programmers. In all cases the collaboration was remote, meetings being reserved for participatory design sessions. These experiences allowed us to refine the proposed process and communication conventions between project managers, graphic designers and programmers. They also provided data to assess the gains brought by the chosen architecture in terms of schedules and effort.

We now describe the development of a departure manager for airports, which was designed and developed over a period of a few weeks in late 2003. It is a good example of how professional applications can benefit from giving designers more expressive power, and of the work process supported by our choice of software architecture. A company had developed algorithms to optimise the sequence of departures and coordinate controllers who guide taxiing aircraft. Air traffic controllers are known to be very demanding professional users, and the company wanted to embed their algorithms in touch-screen workstations that would both provide excellent usability and seduce users and decision makers. The application was meant for pre-operational tests, and consequently had to offer full functionality. The company also had a very tight schedule because they wanted to exhibit the product at a professional convention so as to gain customers. This strict deadline obviously played a key role in the organisation of the project.

The project team was composed of two programmers (one being the lead interface designer and the other a domain expert), and a graphic designer. The project started with a discount participatory design session that produced a paper prototype for every interface to be built in the project. Figure 10 shows one of the prototypes. This prototype served as the reference for all further developments on the interface, in several ways. First, the layout of the different parts of the interface was the result of collective work and served as the basis for future composition work by the designer. Second, all parts of the prototype were given a name: static parts (“printer”, “column”, “timeline”, etc) as well as a template name for dynamic parts (“strip”, “plan”). The names served


Figure 10: The paper prototype that served as a reference for group work

as a basis for informal communication between participants, who never met again until the end of the project. Third, immediately after the session the lead interface designer decomposed the prototype into a tree of components, starting with the top-level parts. This tree represented the architecture of the application, both in terms of software components and graphical components. She used the tree as a collective contract and interface between project actors, especially between UI programmers and the graphic designer.

From then on, the programmers and the designer worked independently, contacts being limited to clarification questions or visual design proposals and feedback. The designer started working on the general impression he wanted to convey, and tested ambiances, colours, harmonies, textures, etc. Figure 11 gives a sample of his work.

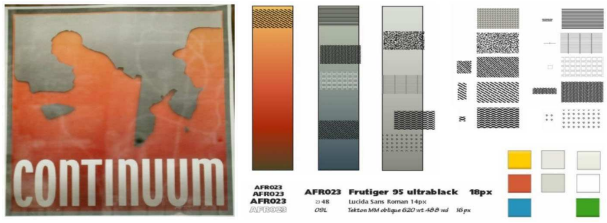


Figure 11: The designer’s work on “the global picture”

Meanwhile, programmers used the reference component tree to code the application. For each component they defined the behaviours, the computations, and the connection to the functional core. They also created basic graphics, which were needed for testing the code. They used a professional drawing tool to produce some of the graphics, others being coded with the IntuiKit Perl API. The result of their work is shown in Figure 12. The application was of very limited visual quality, but sufficient to test its usability as well as the connection to the functional core, implemented as a server.

Three to four weeks before the deadline, the designer had finished maturing the design and preparing his elements: fonts,

Figure 12: The application, with programmer-made graphics

patterns, etc. He was then ready to produce graphical elements to replace the placeholders made by programmers. Having a computer equipped with IntuiKit and being regularly sent new versions of the application, he was able to test the elements in context himself: for that, he just had to drop SVG files in the appropriate places. When satisfied with the result, he sent the SVG files to the programmers by email. Figure 13 shows two SVG designs that are perfectly equivalent from the application's point of view, though not exactly equivalent for the user.

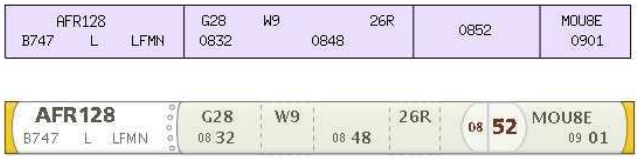


Figure 13: Detail: two skins for the "strips"

Finally, after a period of design, test, feedback, redesign, and integration, the final graphics were produced a few days before the deadline. The very low cost of integrating graphics as well as the ability to work in parallel allowed for these very late iterations while programmers were busy testing and debugging the application. This process allowed every actor of the project to manage time in their own way to work until the last limit, without excessive tension. The final result is shown in Figure 14. As a final note, the product was a great success with potential users and customers at the exhibition, and its interface was perceived by all as an important competitive advantage. This is one more empirical proof of the potential of graphic design for professional systems and the importance of supporting it.

## DISCUSSION AND PERSPECTIVES

Practical experience with TkZinc and IntuiKit such as the one we just described has evidenced both benefits and remaining limitations. The most obvious benefit of using rich graphical models is the visual quality of the resulting user interfaces, when designed by professionals. But we consider the efficiency gains as equally important. The project described above has allowed us to compare figures with a project of similar size and complexity executed by the same team a few months earlier without Intuikit. The earlier project involved the same effort by the graphic designer (15 days) and the same amount of non-graphical UI code (15 kloc). It was built with a more traditional process: the designer produced visual elements that the programmers reproduced with their programming language (Perl, used in object-oriented style). We have observed three major improvements in the execution of the second project with IntuiKit:

- a 20-30% reduction in programming effort, obtained by avoiding the re-coding of graphics;
- a 50-70% reduction in overall project length, obtained with the parallel production process, even though a pipe-line process was set up in the earlier project to incorporate visual elements as they were produced;
- a reduction in coordination costs that we did not measure but that we estimate at 50-70% in terms of number and duration of telephone calls.

We also have identified several limitations and possible improvements:

- Freeing graphic designers from certain constraints has a major impact on performance: they create much more complex visual representations. Between the two projects mentioned above, memory use has more than doubled (from 30 Mb to 70 Mb with IntuiKit). Similarly, graphical performance becomes an issue again, even with recent GPUs. Animations in the application illustrated in Figure 14 are not smooth enough with low-end GPUs.
- The standard SVG format does not support all types of blending between layers, neither does TkZinc. Some designers consider that as a constraint, since they use blending in their iterative construction of visual effects. An option might be to support SVG extensions provided by some graphical tools.
- IntuiKit does not yet support SVG filters, which could be useful to designers especially in terms of rhythm and light. However, performance issues would have to be monitored.
- Some of the features of TkZinc are lost when using IntuiKit and SVG. That forces a trade off between development cost and visual richness. Evolutions of the SVG standard could help avoid that trade off.
- The use of OpenGL in TkZinc provides expressiveness and performance. However, extra care had to be taken when implementing features with OpenGL and there still are a few conceptual flaws caused by it. OpenGL is not primarily aimed at 2D graphics design, and some operations lead to inappropriate results as far as visual designers are concerned: lack of pixel-precise control in certain cases, and unnatural results for certain colour combinations.

IntuiKit was tested successfully on projects where the initial iterative design was done with low-fidelity prototypes, and production started once the general structure of the interface was decided upon. It has not been tested for early design phases. Regarding the response of teams, experience showed that many programmers are not yet used to programming user interfaces as components communicating through


Figure 14: The final application, after full integration

events, and require some training. Once familiar with this notion, programmers exposed to IntuiKit seem to have no problem with the manipulation of FSMs. Designers enjoyed the process, and some wanted to go further. The supported process confines them to the role of providing visual elements, when they could define part of the behaviour of visual elements: visual styles that would define what a progress bar would look like for the whole range of values, for instance. We are considering the use of rules or constraints [18], as well as data-flow techniques inspired from [4], [9] or [5].

Future work also includes the implementation of new models. For GUIs, that would include layout, non-linear geometric transformations, and animation. Finally, access for non-programmers would require tools for building models, when no commercial tools are already available. This includes a graphical editor for building behaviours and associating them to graphics. This could also include other techniques from the literature to build or infer models, such as programming by demonstration.

## RELATED WORK

Although mostly aimed at 3D rendering, the wide availability of OpenGL-capable graphics cards stimulates the introduction of richer 2D tools. For instance, the new Java 1.5 adds OpenGL-based rendering for Java2D. Similarly, the scene graph of 3D toolkits such as Open Inventor [21] has also inspired the design of toolkits for programming post-WIMP user interfaces [2, 1, 11]. The focus of these toolkits is on providing extendable architectures for developing novel interaction and visualisation techniques. Ubit for instance is based on a scene graph model for combining visual elements with behavioral elements and layout elements [11]. These elements are defined in a declarative C++ style, and thus are not accessible to graphic designers. CPN2000 is a graphical Petri net editor [1]. It has been built with a programming toolkit that supports transparency and non-rectangular windows rendered with OpenGL, and uses SVG to display icons. Many SVG renderers such as the Batik toolkit [19] are available, as well as SVG extensions to existing toolkits. Most of them are limited to displaying graphics and do not provide programmers with access to graphical objects so as to manage interaction. Similarly, little consideration is given to work processes and the role of graphic designers.

Previous work on involving graphic designers in the production of user interfaces can be analysed according to the level of flexibility given to designers. At one end are designer-made widgets that can be reused by programmers. Then some tools incorporate designers' know-how for producing semi-automated designs: the Kandinsky system [6] provides programmers with templates they can transform for presenting data; the Kinetic typography engine [12] encapsulates animation techniques from cartoons. At the other end, Hudson et al [7, 8] propose solutions for splitting widgets into several pixmap zones, which allows programmers to resize them at will while designers can still propose interchangeable designs. By using vector graphics, the IntuiKit GUI module avoids the problem of resizing. By not relying on a predefined set of widgets, it gives the designers total con-

trol on the look and behaviour of interactors, but does not support the creation of themes. With IntuiKit, designers produce the graphical part of the interface, rather than providing styles for graphical objects assembled by programmers.

Regarding software architecture, IntuiKit can be compared to the model-driven tools that use the XML format [17]. The closest are the XUL [3] and XAML widget sets, which allow vendors to provide variants of widgets as long as they implement the specification. IntuiKit is more oriented towards post-WIMP interfaces and allows one to freely manipulate graphical objects and behaviours, whereas XAML widgets are provided as compiled C# code. Current efforts by the W3C are aimed at extending SVG with other markup languages such as XForms [16] for describing behaviour, but put less stress than IntuiKit on separating concerns.

## CONCLUSION

We have described the design and features of the IntuiKit interface design suite and its graphical engine TkZinc aimed at supporting designers' needs when developing user interfaces. The features of TkZinc allow programmers to better match the work of designers. The architecture of IntuiKit, based on the combination of models, allows teams to manage graphics separately from behaviour and other software concerns then merge them in a structured tree of components. It organises the collaborative work of designers and programmers. The SVG format, though sometimes restrictive, is key to this process because it allows designers to use their own tools to produce parts of the interface. The architecture of IntuiKit helps bridge the gap between iterative design and software engineering cycles, and is ready for integrating new models and making them available to development groups.

## ACKNOWLEDGEMENTS

Frédéric Lepied (now with Mandrakesoft), Dominique Ruiz and Stéphane Valès contributed to the implementation of TkZinc and IntuiKit. Yves Rinato (Intactile Design) designed the departure manager, which is shown with the kind permission of Sofréavia. Michel Beaudouin-Lafon was of great help in improving earlier versions of this paper.

## REFERENCES

1. M. Beaudouin-Lafon and H. M. Lassen. The architecture and implementation of CPN2000, a post-WIMP graphical application. In *Proceedings of the ACM UIST*, pages 181–190. ACM Press, 2000.

2. B. Bederson, J. Meyer, and L. Good. Jazz: an extensible zoomable user interface graphics toolkit in Java. In *Proceedings of the ACM UIST*, pages 171–180, 2000.

3. D. Boswell, B. King, I. Oeschger, P. Collins, and E. Murphy. *Creating Applications with Mozilla*. O'Reilly, Sept. 2002.

4. S. Chatty. Extending a graphical toolkit for two-handed interaction. In *Proceedings of the ACM UIST*, pages 195–204. Addison-Wesley, Nov. 1994.

5. P. Dragicevic and J.-D. Fekete. Input device selection and interaction configuration with icon. In *Proceedings of HCI-IHM 2001*, pages 543–448. Springer Verlag, Sept. 2001.

6. J. Fogarty, J. Forlizzi, and S. E. Hudson. Aesthetic information collages: generating decorative displays that contain information. In *Proceedings of the ACM UIST*, pages 141–150, 2001.

7. S. E. Hudson and I. Smith. Supporting dynamic downloadable appearances in an extensible user interface toolkit. In *Proceedings of the ACM UIST*, pages 159–168, 1997.

8. S. E. Hudson and K. Tanaka. Providing visually rich resizable images for user interface components. In *Proceedings of the ACM UIST*, pages 227–235, 2000.

9. R. Jacob, L. Deligiannidis, and S. Morrison. A software model and specification language for non-WIMP user interfaces. *ACM Transactions on Computer-Human Interaction*, 6(1):1–46, 1999.

10. J. W. Jespersen and J. Linvald. Investigating user interface engineering in the model driven architecture. In *Proceedings of the Interact 2003 Workshop on Software Engineering and HCI*. IFIP Press, Sept. 2003.

11. E. Lecolinet. A molecular architecture for creating advanced GUIs. In *Proceedings of the ACM UIST*, pages 135–144, 2003.

12. J. C. Lee, J. Forlizzi, and S. E. Hudson. The kinetic typography engine: an extensible system for animating expressive text. In *Proceedings of the ACM UIST*, pages 81–90, Oct. 2002.

13. H. W. Lie and B. Bos. *Cascading style sheets, designing for the Web*. Addison-Wesley, 1999.

14. K. Mullet and D. Sano. *Designing Visual Interfaces*. Prentice Hall, 1995.

15. J. K. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley, 1994.

16. A. Quint. SVG and XForms: A primer. *IBM developerWorks*, Nov. 2003.

17. N. Souchon and J. Vanderdonckt. A review of XML-compliant user interface description languages. In *Proceedings of DSV-IS 2003*, pages 377–391. Springer-Verlag, 2003.

18. P. Szekely and B. Myers. A user interface toolkit based on graphical objects and constraints. In *Proceedings of OOPSLA*, pages 36–45. ACM Press, 1988.

19. The Apache XML project. *Batik SVG Toolkit*. http://xml.apache.org/batik/, 2004.

20. W3C Recommendation 16 November 1999. *XML Path Language (XPath) Version 1.0*. http://www.w3.org/TR/xpath, 1999.

21. J. Wernecke. *The Inventor Mentor, programming object-oriented 3D graphics with Open Inventor, Release 2*. Addison-Wesley, 1994.