Augmenting the Scope of Interactions with Implicit and Explicit Graphical Structures

Raphaël Hoarau
Université de Toulouse, ENAC, IRIT
7 av. Edouard Belin, Toulouse, France
raphael.hoarau@enac.fr

Stéphane Conversy
Université de Toulouse, ENAC, IRIT
7 av. Edouard Belin, Toulouse, France
stephane.conversy@enac.fr

ABSTRACT
When using interactive graphical tools, users often have to manage a structure, i.e. the arrangement of and relations between the parts or elements of the content. However, interaction with structures may be complex and not well integrated with interaction with the content. Based on contextual inquiries and past work, we have identified a number of requirements for the interaction with graphical structures. We have designed and explored two interactive tools that rely on implicit and explicit structures: ManySpector, an inspector for multiple objects that help visualize and interact with used values; and links that users can draw between object properties to provide a dependency. The interactions with the tools augment the scope of interactions to multiple objects. A study showed that users understood the interactions and could use them to perform complex graphical tasks.

Author Keywords
Graphical Interaction Design, Instrumental interaction, Exploratory Design.

ACM Classification Keywords
H5.2 [Information interfaces and presentation]: User Interfaces: Graphical user interfaces - Interaction Styles.

General Terms
Design, Human Factors.

INTRODUCTION

When using computerized tools such as real-time editors, presentation software, GUI builders, etc. users create and manipulate graphical objects on the screen. They can edit them individually, e.g. change their color or their stroke width. Users can also consider and interact with sets of objects as opposed to individual objects. To do so, they may be required to structure the scene, by relying on concepts such as groups, styles, or masters. According to the Oxford dictionary, a *structure* is "the arrangement of and relations between the parts or elements of something complex". Using a structure may have multiple assets, such as helping users conceptualize the scene they are creating ("the background of the slide includes this drawing and this text", "this set of slides is a subpart of the presentation" etc.), and think better about the problem at hand. Here, we are interested in structures as means to interact with the content: since structuring involves sets of objects, the actions done on an element of the structure may have an effect on several objects at once.

In current interactive systems, the use and the management of structures may be complex. Users have to create and maintain them. Depending on the kind of structure, some operations may be cumbersome or impossible to do, which prevents users to explore the design space of their particular problem. Furthermore, systems that provide structuring do not leverage off the structures fully to provide users with new ways of interacting with the content.

Interactions with structure and with multiple objects through a structure have not been studied extensively in the past. Of course, a number of past works have identified the problem [6], but few concepts or properties targeted it explicitly [2,12]. For example, what are the interactions that enable users to define sets of objects? What are the available means to augment the scope of interaction i.e. apply an interaction to several targets? What are the concepts that may guide the design of such interactions?

The work presented in this paper aims at improving the management of structures as means to augment the scope of interactions. Based on contextual inquires and related work, we present a number of requirements pertaining to the interactions with structures. We then present two interactive tools that aim at fulfilling those requirements. The first one is ManySpector, an inspector for multiple objects. ManySpector displays all used values for a property given a set of differing objects, whereas a traditional inspector displays no value. This reveals an *implicit* structure of graphics (the sets of objects that share a graphical property) and offers new interaction means. The second one is based on links that users can draw between object properties to provide a dependency. The resulting property delegation graph is a means for users to provide an *explicit* structure. We then report on a user study involving those tools.

CHI 2012, May 5-10, 2012, Austin, TX, USA.

CONTEXTUAL INQUIRIES AND SCENARIO

We have based our work on concrete and realistic case studies. We have conducted five contextual inquiries with "designers", the design activity being taken in its broadest sense: edition of graphics (Illustrator and OmniGraffle), courses schedule (iCal), architecture (Auto-CAD), or lecture presentation (PowerPoint). We have written a dozen scenarios that describe accurately the activities.

In order to introduce the problem, we present one of the scenarios. This scenario illustrates a number of requirements pertaining to interactions on several objects, with or without a structure. The scenario is real but adapted slightly for illustration purpose: some interactions that are deemed as impossible (e.g. with Inkscape) might be possible with other tools (e.g. with Illustrator and vice-versa). The steps are annotated in italic to characterize them. We detail the annotations later in this section.

Elodie is a designer tasked with creating the graphics of a custom software keyboard for a tablet computer. Using a graphical editor, she creates a first key. She draws a rounded rectangle with a solid white fill and a surrounding stroke. She adds a rectangle inside the previous one, with a blue gradient fill (no stroke). She selects both rectangles with a selection lasso (*designation*) and groups them with a command in a menu (*structuring*). She then adds a soft shadow effect on the group. She overlays a label with a text 'A' on the group of rectangles and centers the label and the group by invoking a 'center' command on a toolbox. She then forms another group with the label and the groups of rectangles, and names it "key" in the tree view of the graphical scene provided by the application (*structuring*). This first key serves as a model to create other keys: she duplicates the key, and applies a horizontal translation to the copy. She proceeds with this action several times in order to get a row of keys (Figure 1). She then modifies the text of each key one by one (Figure 2).

Figure 1. The user creates a key, and duplicates it.

Figure 2. The text of the 'I' key is not centered.

When she changes the letter 'A' for 'I', she realizes that the 'I' text is not centered with regards to the rectangles (Figure 2). The first object was specified incorrectly: if the three objects (label, gradient rectangle, rectangle) are correctly aligned, the text of the label is not centered. The problem was not noticeable with the first letters (AZERTYU) since their widths are similar. Each label being in a heterogeneous group (containing object types other than label), the system does not provide a text center command that can be applied to a selection of objects. She has to click multiple times on an object to reach the label and apply the 'text centered' command. Therefore, she estimates that it is more efficient to start over: she deletes all copies, ungroups the first key, centers the text, groups the objects again, copies and moves the copies, and modifies each letter one by one.

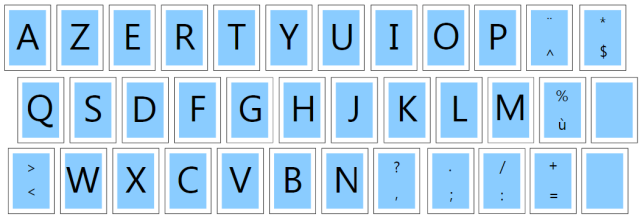

Figure 3. The entire keyboard with the double keys.

Elodie has finished the entire keyboard. Some of the keys are double keys that contain two smaller labels at the top and the bottom of the key (Figure 3). She wonders whether the double key labels are too small and she wants to explore new sizes (*exploratory design*). First she has to find each double key in her design (*searching*). To do so, she zooms out to make the keyboard entirely visible. This allows her to identify each double key. Again, she has to change the size of the labels one by one.

The scenario illustrates several requirements.

Structuring Elodie relied on the ability of the system to allow creation, modification, and management of sets. For example, she created a single group with two rectangles, then another group with the previous one and the label.

Designation Elodie designated objects, properties and actions. For example, she changed the "alignment" property of the label to "centered".

Scope of actions Elodie acted on multiple objects at once. For example, she grouped objects because she wanted to consider them as a single entity that keeps the relative positions between subparts, but also because she wanted to apply a single translation on three objects at once. Conversely, she was not able to apply the command 'set alignment' to several objects at once.

Seeking Elodie needed to retrieve objects: she had to search objects whose content is similar to other ones. The search action requires visually scanning the graphical objects and seeking candidate objects, at the risk of forgetting some of them. The more the objects, the more difficult it is to find out particular ones, especially if the features to search for are not pre-attentive [4]. As the number of keys increases, each modification gets more costly, not only because of the number of actions to repeat, but also because of the required visual search effort.

Exploratory Design Elodie explored parts of possible solutions, and modified existing parts of solutions. By combining action, visualization of intermediate results and thinking, she co-discovered the problem and the solution. In

doing so, she was pursuing an exploratory design activity. This phenomenon is important for activities in which the expected result is not known in advance: graphics edition activities, slides design, or class hierarchy design [8][24].

RELATED WORK

Past works have tackled the problems of managing structures, and interacting with multiple objects, either explicitly or implicitly. We present them along three axes: interactions for structuring the content provided by interactive systems, design and evaluation of interactions for structuring, and structuring in programming.

Structuring for users

Groups Traditional graphical editors allow users to create groups from a set of objects previously selected by the user, and to act on those groups. The only operation available for a group is 'ungroup', which removes the group entity and selects all objects that were part of the groups (no modification, addition, or subtraction). Selection can be seen as a transient group, with 'add' and 'remove' operations by holding the shift key and selecting several elements, or holding the ctrl key and clicking on individual elements. Some tools support heterogeneous settings, but with specific properties only *e.g.* translation, scale and rotation: all elements in the group are transformed accordingly. Conversely, some operations (e.g. 'set color') cannot be applied to groups, supposedly because some elements inside the group do not "understand" them. This forces the user to ungroup and apply the command on each object. In this case, interaction with the structure is not well integrated with interaction with the content.

Trees Groups can be part of a surrounding group, turning them into trees or hierarchies. Support for management of such hierarchy ranges from no support at all, to navigation in the hierarchy of parents [18], and tree views in structured graphics editors (e.g. Inkscape or Illustrator). A tree view enables users to reparent elements with a drag and drop. However, there is no support for other operations, such as applying a color to a node in order to change all children.

Masters A Master is an element used as a "model" for other elements. For example, PowerPoint enables users to define in a master slide the appearance that other slides would inherit. Sketchpad introduced masters as shareable objects that could be used in multiple locations in the scene [22]. Changing a property of the master would modify all objects that depend on this master. This was a way to reduce the number of actions required from the user when something must be changed.

Properties Presto is a document management system that enables users to tag documents with properties, e.g. *year=2012* [5]. Properties provide a uniform mechanism for managing, coding, searching, retrieving and interacting with documents. For example, users can define directories (i.e. a set) of documents using properties: either by extension (by putting elements into the directory), or by intension (with a query such as *size >500k*). Conversely to purely hierarchical structures, properties enable objects to be part of several overlapping sets.

Graphical search Graphical Search & Replace [13] allows users to search for elements based on their graphical properties (*designation*) and change at once a particular property for all found objects (*multiple scopes*). Applications like Illustrator provide such a tool but through a dialog box, not by direct manipulation.

Surrogates Surrogates are specialized interactors that allow users to interact with the surrogate instead of the domain object [12]. Similarly to classical inspectors, surrogates expose attributes that are common to objects, by automatically narrowing the surrogate to the lowest common ancestor. This enables users to interact with those values and modify several objects at once.

User-defined macros and Programming by example User-defined macros allow for automation of repetitive tasks [15]. The user proceeds with an example of the task to repeat, and an algorithm abstracts the actions, so as to enable application on other objects.

Structuring for exploratory design Some structuring techniques have been designed to support exploratory design. The list of reversible actions is an implicit mechanism to help users not to fear possible damages [23]. Side Views display previews of interactive commands [25]. Parallel Paths support alternative exploration by relying on an arborescence of creations instead of a linear history, and on the simultaneous views of parallel results (*comparison*) [26]. Acting on a node of the creation path enables users to manipulate the subsequent designs at once (*scope*).

Structuring for designers

Interaction designers have already identified the need for many modifications with a low number of actions.

Cognitive dimensions In the cognitive dimensions of notation framework [8], the problem described in the software keyboard scenario is identified as "viscosity". It exhibits when the structure of the information contains a lot of dependencies between parts, which implies that a small change leads to numerous adjustments from the user. Viscosity is a hurdle to modification and exploratory design [9]. Since it may be costly to apply the changes, the user refrains from exploring alternatives. A solution to viscosity consists in creating an "abstraction", a "power command" that would act on several objects [9]. An abstraction is a class of entities, or a grouping of elements that users will handle as a single unit e.g. styles in a text document.

Abstraction can be costly. Learning, creating and modifying them require time and effort that should be balanced with investment in repeating a small sequence of actions to solve a small problem. Besides, abstractions can be a hurdle to exploratory design if they are required before any other

simple actions. Finally, abstraction may introduce hidden dependencies: some parts of the scene may depend on others in an invisible way, which makes it hard for the user to predict the effect of a change.

Instrumental interaction and design principles Direct [23] and instrumental [2] interaction techniques are efficient with a single object: they lower the number of required actions compared to other techniques, such as command lines, conversational dialogue, or modal interactions. Design principles related to instrumental interaction, such as reification (turning an object into a thing), polymorphism (applying the same change to different class of objects) and reuse (of past selection and interactions result) extend the scope of actions to multiple objects [2].

Cost of interaction techniques A particular technique is only better than another with respect to the task to accomplish: copy, modification, or problem solving (equivalent to exploratory design) [16]. CIS is a model that helps describe an interaction technique, analyze it, and predict its efficiency in the context of use [1]. CIS defines four properties for interaction techniques. Among them, Fusion is the ability of a technique to modify several work objects by defining multiple manipulations at once (*scope*), and Development corresponds to the ability offered to the user to create copies of tools with different attribute values.

Structuring for programmers

The problems raised so far can also occur during development activities. For example, refactoring tools in IDEs is an answer to the need for multiple scopes of action: if the user changes the name of a method, the system applies this change on each call of the method, possibly in many classes or files. Styles can be implemented in a style language (e.g. CSS), with a hierarchical structuring. Changing a parameter in an intermediate node has an effect on its children. Tags in the Tk toolkit allow the programmer to structure objects in overlapping sets [21]. Changes can be applied to graphical shapes or to a tag, and thus to the set of objects that hold this tag (*scope*). Tags can be defined by extension (with *designated* objects) or by intension (with a predicate e.g. all blue objects) [21].

Prototype-based languages offer an alternative to class-based languages for object-oriented programming [14][20]. They offer a flexible creation model that allows sharing of properties and behaviors. Such mechanisms allow users to structure a hierarchy of prototypes and to act on several clones by manipulating a prototype in the delegation hierarchy. Morphic reifies prototypes and clones into graphic objects (called Morphs), and allows for their construction and edition with direct manipulation [18]. Tools have been designed to help structure a prototype hierarchy. For example, Guru is an algorithm that automatically creates a well-organized graph of prototypes, by factoring shared properties into new prototypes [19].

REQUIREMENTS

In this section, we synthesize the requirements for the manipulation of objects through structures (Table 1). The synthesis is derived from the contextual inquiries we ran, and our analysis of the related work. Notably, the requirements are related to the set of tasks identified in [6] that are known to be difficult to perform with direct manipulation techniques. We have expanded and refined them in this section. We present 3 subsets of requirements: *managing sets of objects* (R1), *managing actions* (R2), *fostering exploratory design* (R3).

Manage sets of objects (R1)	*Search* (R1.1)
	Designate (R1.2)
	Modify (R1.3)
	Identify sets (R1.4)
Manage actions (R2)	*Specify their nature* (R2.1)
	Specify their parameters (R2.2)
	Specify the scope (R2.3)
	Perceive consequences (R2.4)
Foster exploratory design (R3)	*Try* (R3.1)
	Evaluate (R3.2)
	Short-term exploration (R3.3)
	Compare versions (R3.4)
	A posteriori structuring (R3.5)

Table 1: Requirements

Managing sets consists in *searching* (R1.1), and *designating* (R1.2) the objects that are part of a set. It is also necessary to *modify* (R1.3) the sets (add, remove elements). Finally, users must be able to *identify* (R1.4) the objects that belong to a particular set, or determine the sets a particular object belongs to.

Managing actions consists in *specifying their nature* (e.g. by clicking on an ‘alignment” icon, or a menu) (R2.1), *their parameters* (“vertical” or “horizontal”) (R2.2) and their *scope* (R2.3). *Perceiving their consequences* (R2.4) with appropriate feedback enables the user to realize the effects of its action after, and even before it is triggered [23].

In order to support exploratory design, it is important to provide users with tools that enable them to *try* (R3.1) and *evaluate* (R3.2) solutions during short-term exploration (R3.3), and *compare different versions* during middle-term exploration (R3.4) [24]. When satisfied with the results, users must be able to extend the modifications to other objects. If the system does not support this task efficiently, users will have to repeat the same actions to propagate changes (viscosity). Finally, if structuring is a solution to the viscosity problem, it is a hurdle to exploration if required a priori. Therefore, structuring should be made *a posteriori* (R3.5) i.e. when actions have already been done.

INTERACTIVE TOOLS

We have explored a number of interaction techniques to offer new ways of interacting with multiple objects through structures. To design them, we involved the users we interviewed in a participatory design process, with 2 brainstorming and sketching sessions, and 5 evaluation sessions, as demonstrated in [17]. In the following, we cite the requirements that each feature is supposed to address. Requirements serve both as rationale to explain the design, and to help readers determine whether they are satisfied by our claims that the design fulfills the requirements.

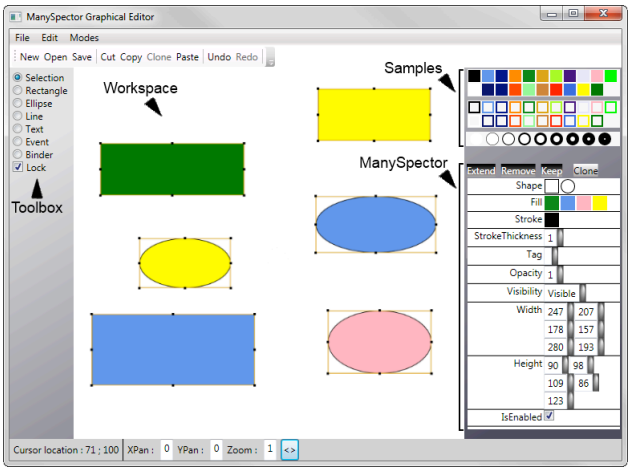

Figure 4. Overview of the application. Center: workspace, top-right: samples; bottom right: inspector.

Overview

To illustrate the interactive tools, we have designed a graphical drawing application. There are four parts: a tool palette on the left side, a workspace in the middle, a sample panel on the top right corner, and an inspector on the bottom right corner (see Figure 4). The workspace is the main view, where users can create a new object by clicking and resizing. Selection is performed by clicking on an object or by drawing a rubber rectangle to encompass several items, as implemented in usual graphics editors. A bounding box with handles surrounds selected items.

The samples panel contains a set of values for shape (square, oval, T for text), fill color (represented by a colored square), stroke color (stroked-only colored square) and stroke thickness (stroked-only circle). In order to modify a property of an object in the main view, users can drag a sample and drop it onto the object. Feedback is shown as soon as the sample hovers over the object, in order for the user to understand the action and to assess the change before effectively applying it by releasing the mouse button. This enables the user to cancel the action, by releasing the button outside of any object (*R3.1 try, R3.2 evaluate, R3.3 short term, R3.4 compare, R2.4 perceiving consequences*). Drag and drop of samples also applies to a selection of objects. The interactions described so far are not entirely novel. The next sections present two tools with novel interactions.

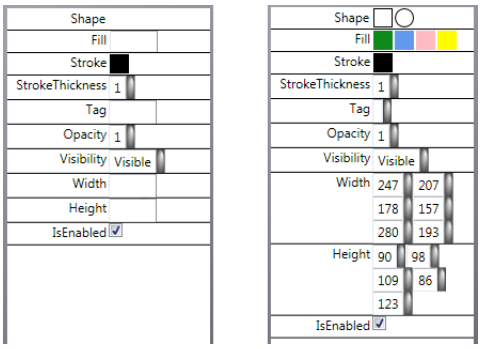

Figure 5. The user's selection contains objects with varying shapes, fill colors, width, and height. A classical inspector (left) displays a blank fill for those properties, whereas ManySpector (at right) displays all different values.

Implicit structure: ManySpector, an enhanced inspector

An inspector (or property sheet [11]) is a window containing a vertical list of pairs of property name and value (e.g. shape: rectangle, color: green, thickness: 3). An inspector offers two services to the user: visualizing values with progressive disclosure and modifying them [11]. If multiple objects are selected, a classical inspector only displays values shared by all selected objects (e.g. stroke color in Figure 5, left). Users can change such a value, and the system reflects the change to all selected objects. The inspector does not display any value for properties for which there are multiples values (e.g. fill color in Figure 5, left). Users are thus not informed about those values, and sometimes cannot modify them through the inspector.

We have designed ManySpector, an inspector that displays all used values for a property given a set of differing objects. For example, in Figure 5-right, the Fill property displays all colors used by objects in the selection. Used values reveal an *implicit* structure of graphics, the sets of objects that share a value for a given property. Though not explicitly defined by the user, we think that such sets may be useful, since users sometimes think about objects with a graphical predicate ("all red objects"). We relied on the display of used values to design a set of interactions that offer new services for exploratory design and structure-based interaction: query and selection of objects with graphic examples, selection refinement, and properties modification on multiple objects.

The representation of a shared value in ManySpector actually reifies [3] both the value per se, and the set of selected objects that exhibits this property value. As a value per se, and similarly to the interaction with the sample panel, users can drag the shared value (considered as a value) from ManySpector onto (a selection of) objects in the main view to modify a property. If the shared value is numerical, users can hover over it and rotate the mouse wheel to increment or decrement it (*scope* and *specify actions*). Together with immediate feedback, this enables both exploration and precise adjustment of properties, thus reducing temporal offset [2] between action and feedback.

ManySpector limits the number of used values to half a dozen. If the number of used values is larger, a scrollbar enables the user to browse through all values. When the cursor hovers over a property placeholder, an animation enlarges it smoothly to reveal other used values.

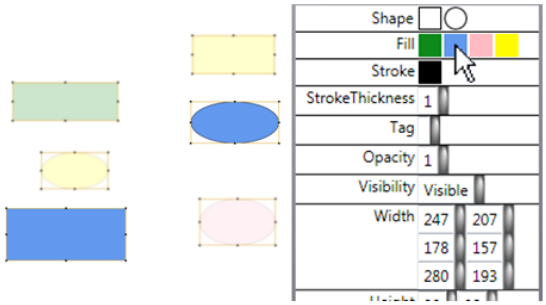

Figure 6. The cursor is over the blue shared value of the fill property. Because they don't have this shared value, the green rectangle, the pink circle and the two yellow shapes are dim.

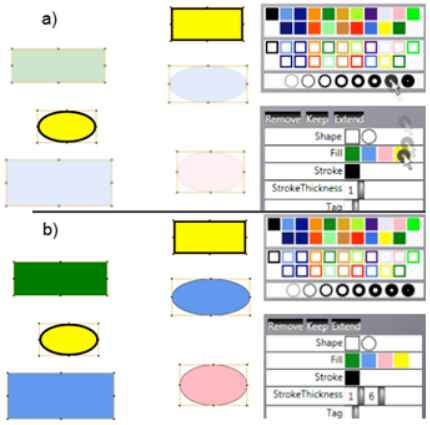

Figure 7. Starting from Figure 5, a) the user drags a "stroke thickness: 6pt" sample over the "fill: yellow" shared value. Immediate feedback turns the stroke thickness of all yellow items to 6pt. b) the user has dropped the sample, the modification is applied.

Since a shared value also reifies a set of objects, hovering over a shared value highlights the relevant objects while blurring others with a short animation (Figure 6). This makes it easy to figure out which set is made of what (*identify sets R1.4*), and to detect outliers and fix them. Users can drag a sample (a value) from the sample panel onto a shared value (considered as a set of objects) to modify at once a property for multiple objects (*R2.3 scope*) (Figure 7). Users can also drag a shared value (value) onto another shared value (set) (Figure 8).

To select objects, users can click on them in the workspace, or draw a selection rectangle. In order to refine the selection, users can use three meta-instruments (i.e. instruments that control instruments, here the selection): *Remover*, *Keeper* and *Extender*. The interaction consists in a drag and drop of the representation of the instrument onto a shared value. *Remover* throws out of the selection all objects that have this shared value (Figure 9). *Keeper* keeps in the selection the objects that have this shared value, and throws away the others. *Extender* adds to the selection all objects that are not selected but that possess this shared value. The instruments can also be dropped onto an object of the scene to add or remove it from the selection. These interactions extend the set of example-based queries introduced above (*R1.3 modify sets*).

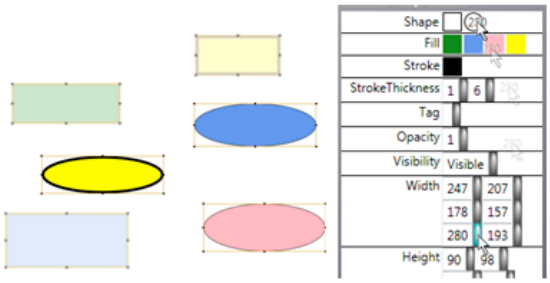

Figure 8. The user drags the "width: 280" shared value and drops it on the "shape: circle" shared value. All circles in the selection now have a width set to 280.

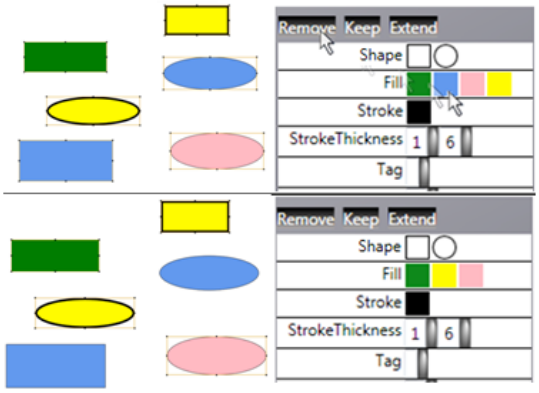

Figure 9. The user drags the Remove tool onto the "fill: blue" shared value. Blue objects are removed from the selection.

Explicit structure: the property delegation graph

Besides ManySpector, we have explored an interactive tool that enables users to structure the content explicitly. Users can specify that a property of an object (the clone) depend on the property of another object (the prototype). A prototype is similar to a master in Sketchpad: when users change a property of a prototype by dropping a sample from ManySpector onto the prototype, all dependent clones are changed accordingly (*R1.3 modify sets, R2.3 scope*).

The interaction to specify a dependency is as follows (Figure 10): by clicking on an object, users can toggle the display of the properties around it. They can press on a property, draw an elastic link, and drop it onto another object as if they were dropping a sample. The clone object appearance reflects immediately the appearance of the clone for that property. Users can remove a link by pressing the mouse button in the blank space, drawing across the links to be deleted, and release the button.

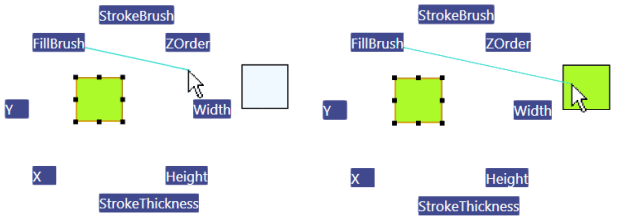

Figure 10. The user draws a link between the fill property of the green object (the prototype) into the blue object (the clone) to specify a dependency. The fill color of the clone turns to the color of the prototype (green).

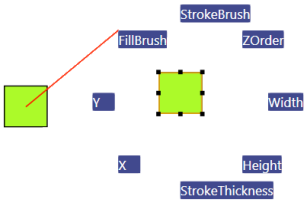

Figure 11. The user has selected the clone to see the dependency.

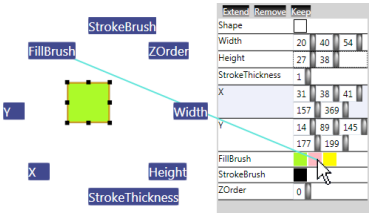

Figure 12. The fill property is dragged onto a used value to specify that the fill property of a set of objects depend on the prototype.

The system proposes two ways of creating new objects from existing ones: either by copying it or by cloning it (*R1.3 modify sets*). Copying is the regular copy operation: properties from the copy are independent from the properties of the source. Cloning enables users to get a clone, whose properties are entirely delegated to the copied object (the prototype) (Figure 11). By creating a clone, users minimize the number of actions required to specify a single difference with the prototype: if they copied instead of cloned, they would have to link all shared properties.

Explicit structuring is supposed to bring more action power, at the expense of increasing viscosity and hindering exploratory design since users have to manage a structure. We have lowered these drawbacks with a posteriori structuring and by leveraging off ManySpector. For example, choosing to clone or to copy may be premature at the moment of the creation of a new object from an existing one. To solve this problem, users can decide to change them to a copy or a clone after the creation of the object (*R1.3 modify sets, R3.5 a posteriori structuring*). This is made possible by tracing the history of objects, and how they were created. Toggling between copy and clone only affects the properties that were not set explicitly by the user. Another problem is to interact with similar objects in order to make them depend on a prototype. A viscous solution would be to interact with each object and making it a clone of the prototype. A more efficient solution consists in selecting the objects that are to be clones, and in dropping the property of the prototype onto an object of the selection (*R1.3 modify sets, R3.5 a posteriori structuring*). Users can also drop the property onto a shared value in ManySpector (Figure 12), which links all objects sharing that value to the prototype.

The property delegation graph is an extension of the delegation tree found in prototype-based languages [14]. However, with a tree, objects cannot have multiple parents. For example, the scene tree available in illustrator may be helpful to conceptualize the scene, but is unable to help specify cross-branches relationships. Conversely to a tree, a node in our graph of properties can have multiple parents. This enables users to be more specific about the parent that holds a particular property: a node can delegate 'fill' to a prototype A, and 'stroke-width' to a prototype B.

Discussion about the design

The interactions are consistent: they all use modeless interaction based on drag and drop, be it from or on an object on the scene, a shared value, or a prototype. With immediate feedback and a posteriori structuring, they also support exploratory design. The properties are immediately visible (no need to devise a query): users can try and test by hovering over and off the used values, and assess the results thanks to immediate feedback without applying the change (button still pressed).

The interactions we devised can be considered as a kind of surrogates [12]. We have expanded them by explicitly taking into account the interaction to manage the selection and explicit structuring. Furthermore, our version exposes not only common properties but also all used values, which makes direct the access to more subsets and expands notably the scope of interactions. Of course, existing systems enable users to obtain the same final results, and even by relying on similar concepts (flash, sketchpad). Those systems actually provide the same *functionalities*, but not the same *interactions*. For example, existing tools do enable users to perform a graphical search, but with an indirect manipulation (through a menu and a dialog box). This prevents users from quickly trying and testing changes and hinders exploratory design. In addition, interactions are not well integrated *e.g.* in Illustrator, there is a tree view, but users can use it only to select a branch then apply a limited set of changes on the selection.

As such, the prototypes have issues. For example, more work needs to be done with respect to scalability: ManySpector is not able to handle very large sets of used values. The solution with a scrollbar and progressive disclosure may not be sufficient. The prototype/clone view also needs more work: if the links are numerous, the scene may result in a mess of tangled links. Again, progressive disclosure is a possible solution but we are also exploring other representations and interactions [10]. Furthermore, the system does not check for cycle when the user tries to link two properties. Appropriate feedback is necessary to prevent it, such as displaying the links to show a potential cycle when hovering over a property.

USER STUDY

We have argued in the previous sections that our tools are novel, consistent and effective for performing structure-based interaction. Assessing those claims is not a straightforward task. We were especially concerned with the understandability of the used values concept, and the fact that they refer either to a value or the set of objects that share this value. Would it be too difficult for users to grasp the shared value concept and linked properties? Even if users understand them, how would they struggle when trying to use them to interact with multiple objects? Finally, can users translate high-level problems into graphical interactions with used values and linked properties?

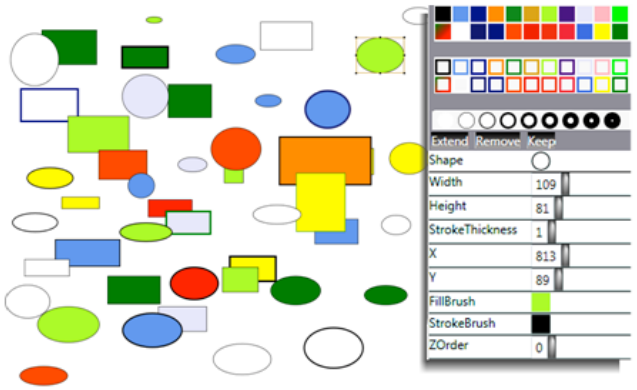

Figure 13. The scene containing many objects.

Tasks

The evaluation session was divided into three parts, each dedicated to one of the three questions above. The first part was devoted to a tutorial that teaches users about used values and links, and how to interact with them in the graphical editor. The two other parts are scenarios that were designed so that they implement the requirements.

In the tutorial, we instructed users to create a few objects, link them, change their color or stroke thickness, with a single object or a set of objects. The tutorial lasted 10min and included 15 simple tasks. Users were actually manipulating the mouse and performed interactions while they were listening to our instructions. The goal of this tutorial was not only to instruct users, but also to see if they understood the design. We assessed their understanding by observing them perform small tasks with no instructions and by asking them if they were confident in their understanding. We did not assess discoverability since we began with a tutorial. This aspect is left for future work.

The second part of the session was an actual test. The test was still using the graphical editor, but this time with a scene containing multiple (50) differing objects (see Figure 13). We asked users to perform more complex tasks such as 'change the thickness of all yellow circles to the maximum of all thicknesses'. We did not give any instructions, and left users perform the tasks by themselves. One of the expected benefits of used values is to help users select a set of objects with minimal interactions. Hence, we designed the tasks to make traditional selection (i.e. a selection rectangle, or adding shapes to the selection by shift-clicking on them) more and more difficult either because they involve multiple objects (*scope R2.3*), or because they involve graphical properties that are not perceptually pre-attentive (*search R1.1, identify sets R1.4*). For examples, the task "change all circles' color" is difficult because users need to find all circles in a scene, a visual task known to be non pre-attentive and that requires a cumbersome one-by-one scan of graphical objects (try on Figure 13). Users were free to carry out the tasks the way they want, either by selecting shapes with the traditional way or using ManySpector (*designate R1.2*). The goal of this second part was to assess the extent to which users would rely voluntarily on used values and links, whether they would be able to perform non-trivial graphical tasks (*specify action R2.1 and parameters) R2.2*), and how well they could interact with used values and links.

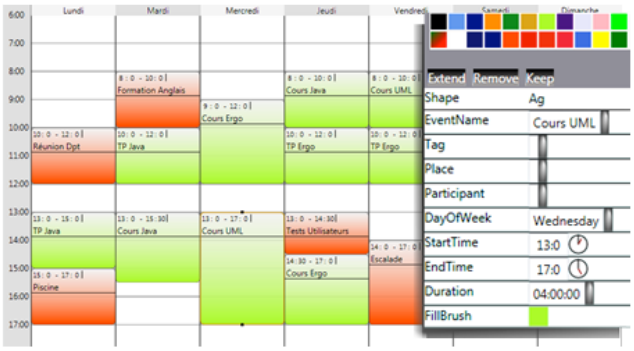

Figure 14. The calendar view.

The third part involved a calendar application. Users were manipulating events on a week view (see Figure 14). Events are represented with rectangles with a title text and a start hour text. They are placed horizontally according to day of occurrence in the week and vertically according to the time in the day. The screen is filled with seven columns, one per day in the week. Instead of graphical properties, the ManySpector window contained calendar-related properties such as start, duration, title etc. as in the iCal inspector. Conversely to iCal, ManySpector displays used values. This allows for modification of unrelated events, while iCal allows for modification of multiple repeated (i.e. recurring) events only. We provided a partially filled schedule and we asked users to act as if they were teachers trying to schedule lecture sessions during the week with a schedule "manager" (the role we played). For example, we asked them to place a 2-hour long lecture Wednesday afternoon. Then we told them that when we said "place a lecture at 10am", we actually meant "10:15am", so they had to change all "10am" lecture events to "10:15am" (*a posteriori structuring R3.5*). The goal of this third part was to assess whether users could translate higher-level tasks to graphical interactions with our tools. The tasks were high-level, and required users to *try R3.1*, *perceive the consequences R2.4*, *evaluate R3.2* and *perform short-term exploration R3.3*.

Since the calendar scene contained few elements only (~15), we were expecting that users would rely on traditional selection. Hence we asked them to use ManySpector instead of the traditional selection.

Subject profiles

We performed the tests with five subjects. Three of them use calendar application in a day-to-day basis, one of them was a graphical designer used to applications such as Illustrator, and one was a casual user of graphical tools such as presentation software. They were all aware about the viscosity problem that might occur when using such tools. Only the graphical designer was involved in the participatory design process, hence four users discovered the interactions for the first time.

Procedure

We asked subjects to think aloud [7] while they were acting. We observed them and logged what they tried, whether they struggled, made errors or succeeded. At the end of the second and third part, we made them fill a questionnaire to rate the difficulty and cumbersomeness of the tasks, and the usefulness of the design with a Likert scale from 1 (negative) to 5 (positive). Results are given in the following, with the mean and the standard deviation.

Results

We did not notice serious understandability problems. Users were able to manipulate shared properties and links, and succeeded in performing simple tasks at the end of the tutorial. When asked about their confidence, some of them felt that they needed some learning "*to do it well*". We showed them many interactions, but even if the interactions are well integrated, users felt that they could not get familiar with them within such a short time. In addition, because there were several possibilities to accomplish tasks, users were always eager to find the best way of accomplishing it, which adds to their feelings. Our confidence into users' understandability got stronger when we witnessed that they got more capable as they were performing the second and third part. We even observed users trying interactions that we did not designed but that were perfectly meaningful, such as using selection instruments (keep, remove) directly on samples to avoid the necessity to perform a selection of the entire scene, dropping a value onto a property name to apply it to all objects, or dragging a sample next to existing used values to extend the selection. This suggests that the design was consistent and predictable.

We did notice some difficulties when users performed more complex graphical tasks in the second part (*ease of translation in graphic scenario*: mean: 3.6, stddev: 0.5). This can be explained by the fact that users were still learning the interaction. They also told us that the tasks were rather abstract. In fact, since the tasks were purposely complex, they lacked significance (none performed 'change the thickness of all yellow circles to the maximum of all thicknesses' in real-life). They struggled to understand and memorize them, which hindered their ability to devise a solution. The four non-graphical designers found the requests much less difficult in the last part with the calendar application and meaningful tasks. Still, all subjects were able to accomplish every tasks of the second part by themselves. (*mean of the easiness of the 9 subtasks of the graphic scenario*: 4.6; 0.5).

We were wondering about voluntary use. We observed what we expected: with tasks that involve pre-attentive properties (such as color-oriented one: 'turn yellow objects into red'), subjects were sometimes still using a traditional selection. However, they turned by themselves to used values with non-pre-attentive tasks, or when the number of objects was too important. They also used links when we asked them to repeat an interaction on the same set of objects: after a number of repetitions, some subjects turned a specific object into a master. This enabled them to be more efficient than devising a selection again with the ManySpector. All kinds of interaction were performed (with samples, used values, links), and all combinations of source and destination for drag and drop were witnessed.

We did not notice difficulties when users had to translate higher-level tasks into interactions in the calendar test (*ease of translation in calendar scenario*: 4.2; 0.8). We witnessed a tendency to use traditional selection for very simple tasks. When we forced users to employ our interactions instead, they did not have difficulties to do so (*mean of the easiness of the 7 subtasks of the calendar scenario*: 4.7; 0.5). This suggests that the interactions can be applied to other contexts than graphical edition.

Even if we did not plan to evaluate usability, the tests revealed some issues such as the difficulty of interacting with the text boxes. Users also found limits to the interactions we proposed: in some cases, users would have liked to keep objects based on a combination of values instead of a single one. As expected, links lacked visibility and legibility when numerous.

All in all, the study allowed us to answer positively to our concerns: the tools fulfill the requirements since users were able to understand the interactions, could perform complex graphical tasks with them and could translate higher-level tasks into them. Users judged ManySpector very useful (*ManySpector usefulness*: 4.8; 0.4). They liked explicit structuring with links though not as much as used values (*links usefulness*: 4.4; 0.9). They also praised the fact that there was no imposed strategy and that they could perform tasks their way.

CONCLUSION

We have tackled the problem of interaction with structures, and interaction with content through structures. We have defined a set of requirements and have explored a set of consistent interactions that provide partial answers to the

requirements: ManySpector, an inspector for multiple objects, and explicit delegation links. A study showed that users are able to perform complex graphical tasks with them. The examples involved a drawing editor and a calendar but the requirements and interactions are not specific to these applications, and can be applied to others.

Our interactions suffer from some problems such as scalability (though this may not be a problem for e.g. the calendar) and legibility. Other designs are possible: we are currently investigating other forms of explicit structuring with no links. We also plan to assess how well those interactions support exploratory design.

ACKNOWLEDGMENTS

We thank all participants of the workshops, and our colleagues for early and late feedback on the paper.

REFERENCES

1. Appert, C, Beaudoin-Lafon, M, Mackay, W. E. Context matters: Evaluating Interaction Techniques with the CIS Model. *Proc. HCI'04*, 279-295. Springer Verlag, 2004.
2. Beaudouin-Lafon, M. Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. In *Proc. CHI 2000*, ACM, 446-453.
3. Beaudouin-Lafon, M. and Mackay,AW. E. Reification, polymorphism and reuse: three principles for designing visual interfaces. In *Proc. of ACM AVI 2000*, 102-109.
4. Conversy, S., Chatty, S. and Hurter, C. Visual scanning as a reference framework for interactive representation design. In *Information Visualization*, Sage, 2011.
5. Dourish, P., Edwards W.K., LaMarca, A. and Salisbury, M. 1999. Presto: an experimental architecture for fluid interactive document spaces. *ACM Trans. Comput.-Hum. Interact.* 6, 2 (June 1999), 133-161.
6. D. M. Frohlich. The history and future of direct manipulation. *Behaviour & Information Technology*, 12(6): 315–329, 1993.
7. Ericsson, K., & Simon, H. (May 1980). Verbal reports as data. *Psychological Review*, 87 (3): 215–251.
8. Green, T.R.G., Cognitive dimensions of notations, *People & Computers V*, 1989, Cambridge Univ. Press, 443-460.
9. Green, T.R.G, and Blackwell, A. Cognitive dimensions of information artifacts: a tutorial. (Version 1.2), 1998.
10. Holten, D., Isenberg, P., van Wijk, J. J., Fekete, J.-D. 2011, An extended evaluation of the readability of tapered, animated, and textured directed-edge representations in node-link graphs, *IEEE PacificVis*, 195–202.
11. Johnson J.A., Roberts T.L., Verplank W., Smith D.C., Irby C.H., Beard M., and Mackey K. The Xerox Star: A retrospective. *IEEE Computer*, 22(9): 11–29, 1989.
12. Kwon, B., Javed, W., Elmqvist, N., and Yi, J.-S. Direct Manipulation Through Surrogate Objects. In *Proc. of ACM CHI 2011*, 627-636.
13. Kurlander, D, Bier, E.A. Graphical Search and Replace. In *Proc. of ACM SIGGRAPH '88*, 113-120.
14. Lieberman, H. 1986. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proc. of OOPLSA '86*. ACM, 214-223.
15. Lieberman, H. Your Wish is my command: Programming by example. Morgan Kaufmann, 2001.
16. Mackay, W.E. Which interaction technique works when?: floating palettes, marking menus and tool-glasses support different task strategies. In *Proc. of AVI '02*. ACM, 203-208.
17. Mackay, W.E. Using Video to Support Interaction Design. DVD Tutorial, *CHI'02*, ACM.
18. Maloney, J.H. and Smith, R.B. 1995. Directness and liveness in the morphic user interface construction environment. In *Proc. UIST '95*. ACM, 21-28.
19. Moore, I. 1996. Automatic inheritance hierarchy re-structuring and method refactoring. In *Proc. of OOPSLA'96*. ACM, 235-250.
20. Myers, B. A., Giuse, D. A. and Zanden, B V. Declarative programming in a prototype-instance system: object-oriented programming without writing methods. *SIGPLAN Notice* 27, 10 (1992), 184-200.
21. Ousterhout, J. K. Tcl & Tk Toolkit. Addison-Wesley, 1994.
22. Sutherland I.E. 1963. Sketchpad: a man-machine graphical communication system. In *Proc. of AFIPS'63*. ACM, 329-346.
23. Shneiderman, B. Direct manipulation: a step beyond programming languages. *IEEE Computer* 16(8), 57–69, 1983.
24. Terry, M. and Mynatt E.D. Recognizing creative needs in user interface design. *Proc. of Creativity & Cognition*. ACM, 38-44, 2002.
25. Terry, M. and Mynatt, E. D. Side views: persistent, on-demand previews for open-ended tasks. In *Proc. of UIST 2002*.ACM, pp. 71-80.
26. Terry M, Mynatt E.D, Nakakoji K, and Yamamoto Y. 2004. Variation in element and action: supporting simultaneous development of alternative solutions. In *Proc. of CHI '04*. ACM, 711-718.
27. Ungar, D, Smith R, B. SELF: The Power of Simplicity. In *Proc. of OOPSLA '87*. ACM, 227-242.