Creating Visual Objects by Direct Manipulation*

Toshio TONOUCHI† Ken NAKAYAMA Satoshi MATSUOKA‡
Department of Information Science
The University of Tokyo

Satoru KAWAI
Department of Graphic and Computer Sciences
The University of Tokyo

Abstract

Low-cost implementations of graphical user interfaces (GUIs) have relied on the widget library framework. Although conventional widgets are suitable for developing typical GUIs with predetermined interaction styles, application-specific customization of interactions is rather difficult, especially for a non-programmer. Instead, we propose a new framework whereby the GUI designers can arbitrarily compose new visual objects recursively from intrinsic primitive objects. The behavior of a composed object is governed by constraints extracted from the trace of operations issued to the graphic editor. A prototype system Oak based on the framework is successfully implemented. Oak allows GUI designers to compose visual objects by direct manipulation allowing non-programmers to create customized widgets of high-degree of complexity.

1 Introduction

The widget framework greatly reduces the cost of GUI development. However, some problems still remain: (i) Commonly used widgets such as buttons and scrollbars are prefabricated and already available, but creation of new widgets and/or major customization of preexisting widgets to meet the requirement of non-standard interaction style is difficult. (ii) As has been pointed out by Myers[9], callback functions closely bind the interface and the application — for example, when one constructs a calculator application, both the numerical calculation (which is an application dependent action) and the visual feedback via hilighting of buttons (which is an application independent action) are invoked via callback functions bound to clicks on the button widgets that represent the numerical keypad. (iii) Programmers describe the behavioral part of widgets as textual programs; thus, it is extremely difficult for non-programmers, such as GUI designers, to create or customize application-specific widgets. Furthermore, it is difficult even for programmers to foresee how the GUI being constructed will be visualized during the coding process.

Previous advanced widget frameworks have attempted to address these issues: For example, UniDraw [16] does relegate some amount of non-application specific code into the widget framework, but only primitive behaviors such as widget layouting are supported without customization. MoDE [12] UIMS provides composibility of constituent parts of widgets. It also enables the programmer to join the modules via direct manipulation to obtain a composite widget. However, MoDE restricts the composition to rather high-level prefabricated constituents, and furthermore, still requires the programmer to manage the complex "spaghetti" of callback functions. Myers [7] proposed a system that uses direct manipulation to specify the desired connection among widgets. Although it does increase the modularity by separating the application code and widgets, the level of widgets it supports are prefabricated ones (e.g. buttons); thus, only simple and obvious compositions such as dialogue boxes are available. It also requires coding knowledge for specifying complex connections, making it difficult for non-programmers to use the system.

To cope with these problems, we propose a new framework, in which widgets are arbitrarily recursively composible from predefined primitives by direct manipulation [15]. We refer to these extended widgets with arbitrary composition capability as *visual objects*. When a designer wants to create a previously undefined visual object, he composes it from the predefined ones with a graphic editor, instead of writing program code. For example, to define a slider object, the designer composes a knob, a groove, and a frame with a graphic editor in an obvious way (Figure 1).

Using this framework, a variety of visual objects can be constructed using the arbitrary composition capability, simply by assembling the predefined *subobjects* via direct manipulation. Furthermore, composed objects can also be used as subobjects of more complex objects. The composite 'visual' behavior of composed

*To be presented at the 1992 IEEE International Workshop on Visual Languages, Seattle, Washington, Sept. 1992.

†Currently with C&C Systems Research Laboratories, NEC Corporation, Tokyo. Japan, tonouchi@btl.cl.nec.co.jp

‡{ken,matsu}@is.s.u-tokyo.ac.jp

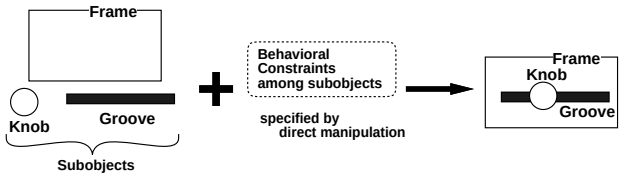


Figure 1: Composition of a slider object

objects is governed by constraints, separating the application-independent action from the application-dependent action. In this sense, our framework might seem similar to the one proposed in Peridot [6], but is actually quite different because (i) we allow and stress arbitrary compositionality of visual objects, and (ii) we allow specification of richer classes of application-specific visual objects by the use of a more powerful constraint solver, and (iii) application-independent action of complex composite objects is encapsulated within the object itself, whereas Peridot does not support specification of coordinated behavior of widgets.

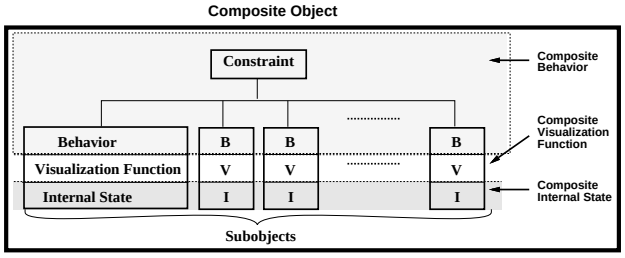


Figure 2: The BVI model

To realize the framework, we propose the underlying *BVI model* where we regard a visual object as consisting of three constituents: the *behavior* (B), the *visualization function* (V) and *internal state* (I). The internal state holds the attributes of the visual object. The visualization function displays the visual object, reflecting its internal state. The behavior specifies application-independent action of the visual object: it updates the internal state of the visual object in response to the interaction by the user. In the slider-object example, the internal state of a knob object is its coordinates. Its visualization function displays a circle at the coordinates specified by the internal state. The behavior specifies that the knob follows the mouse cursor when dragged. The behavior of the composite slider object further dictates that the knob is constrained to move along the groove.

We have implemented a prototype system Oak, which is based on the model. The (non-programmer) designer can arbitrarily compose new visual objects by composing predefined subobjects by direct manipulation without textual programming. In Oak, the designer first draws a visual object on the graphical editor which provides elementary (CAD-like) geometric operations. The system derives constraints between subobjects from the trace of operations on the graphic editor. The code generator then compiles the description of the subobjects (the geometric relation among the subobjects) into C++ as special subclasses of InterViews [1]. The resulting visual objects are available for use within the application with the integration of a run-time routine, which includes a special-purpose constraint solver that can handle non-linear constraints, and is tailored for interactive GUI.

2 Visual Object Composition with Constraints

To simplify the treatment of composition of visual objects, we propose the *BVI model* (Figure 2). In this model, interaction from a user is treated as follows: when the user interacts with visual objects using the mouse, the behavior of the objects updates the internal state of the objects if necessary. The visualization function visualizes the new appearance of the objects.

Based on this model, various kinds of visual objects are composed from predefined subobjects. The composition of visual objects is defined as a set of three independent composition of B, V, and I constituents:

- Composite internal state is the collection of all the internal states of subobjects.
- Composite visualization function is the collection of all the visualization functions of subobjects (with appropriate precedence).
- Behavior composition requires supplementary information to determine composite behavior. The composite behavior is not only a collection of behaviors of subobjects, but also constraints are composed so that the behavior of the interrelated subobjects are coordinated or restricted.

For example, the composite internal state and visualization function of a slider object is merely a collection of those of the knob, the groove, and the frame, while the composite behavior keeps the knob constrained to the groove. That is to say, the intrinsic behavior of a knob is to exactly follow the mouse cursor in response to user dragging, but the imposed constraint of the composite slider forces it to be on the groove. The constraint solver arbitrates these constraints and updates the composed internal state.

Visual objects also support the *export value* mechanism to specify the application-dependent action. Whenever an export value is updated via user interaction or constraint solving, the corresponding callback function attached to the value is invoked to notify the application.

Constraint-based specification [13, 14, 10] has the following advantage: In a conventional widget framework, the programmer is responsible for governing all the control flow information using complicated call-

back functions. Constraint-based visual object composition frees the designer from such responsibilities, since the constraint solver guarantees the (behavior described as) constraints to be always satisfied. Moreover, by defining constraints to be composible, the advantage is maintained for composite objects as well.

3 Prototype System Oak

3.1 Oak Overview

Based on the BVI model, we have implemented a prototype system Oak that supports construction process of visual objects by direct manipulation. The Oak system consists of three subsystems: the *graphic editor*, the *code generator*, and the *run-time routines*. In Figure 3, the three rectangles with dashed borders indicate the subsystems, and arrows describe the flow of information in the construction of visual objects.

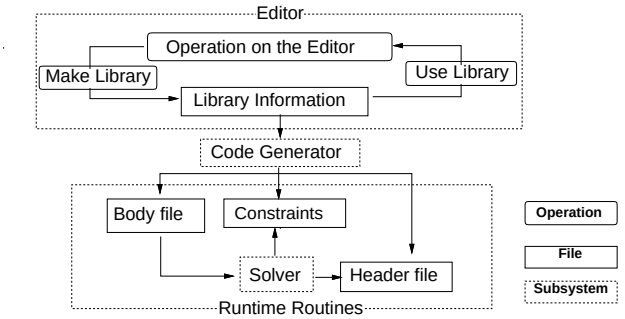


Figure 3: Overview of the system

We give a simple example of generating a equilateral triangle object, to overview how each subsystem works. The equilateral triangle object we are generating is assumed to have two actions: (i) the lengths of its three edges are constrained to be always equal even if the user drags its vertices, and (ii) the specified callback function is invoked when the lengths of edges are changed. The former is an application-independent action and the latter is an application-dependent action.

Graphic editor The *graphic editor* is a tool with which the GUI designers compose visual objects from subobjects via direct manipulation. It provides primitive operations that are similar to elementary geometric operations in traditional graphics, such as drawing a circle with a pair of compasses or drawing a line with ruler. These operations are listed in Figure 4. Using these operations, the designer specifies the composition of visual objects. Most operations of the graphic editor, such as "put a point on an object", imply the constraints on the objects involved. In such cases, the implied constraints are added to the application-independent action of the composed object.

In the equilateral triangle example, the designer issues the following operations in sequence (Figure 5): (i) specify two points, (ii) draw two circles centered at the points so that each circumference goes through the other point, (iii) select one of the intersection points of the two circles, then (iv) draw three lines connecting the three points[1]. The circles are drawn as auxiliary lines (displayed as dashed lines) which can be specified with a toggle switch at the right-bottom corner of the graphic editor. Auxiliary lines become invisible in the runtime phase. The constraint that three edges of the equilateral triangle objects have the same length is implied by step (iii) in the sequence of the operations. We cover the constraint extraction in Section 3.2.

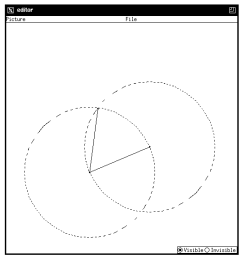


Figure 5: Composition of an equilateral triangle

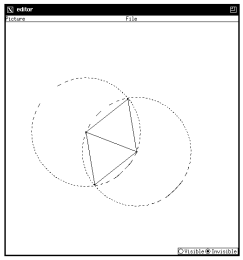


Figure 6: A triangle used as a library command

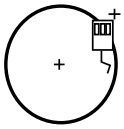
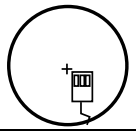
In addition, the designer designates *export values* in the composed object for interfacing with application-dependent action. By this designation, (i) methods to access the export values, (ii) appropriate invocations of the corresponding callback functions, are generated during the code generation phase. In the example, the designer selects 'Length' export value command, and suggests which length of the equilateral triangle he is interested in. Then, the editor requires the designer to enter the name of the callback function to be invoked when the length is changed.

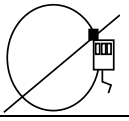
```
id1 = POINT(VISIBLE)
id2 = POINT(VISIBLE)
id3 = CIRCLE(id1, id2, INVISIBLE)
id4 = CIRCLE2(id2, id1, id2, INVISIBLE)
id5 = LINE(id1, id2, VISIBLE)
id6 = INTERSECTION(id3, id4, VISIBLE)
id7 = LINE(id6, id2, VISIBLE)
id8 = LINE(id6, id1, VISIBLE)
```

Figure 7: `triangle.lib`

The trace of the operations the designer performs on the graphic editor is recorded in a file called the *library information file*. This file is processed by the code generator in the next phase. The library information file of the equilateral triangle library is shown in Figure 7. The format of the

[1] The editor automatically selects an intersection when the user specifies a location near the intersection.

Operation `Command`	Arguments	Generated Object	Constraints	
Put a point `POINT`	*none*	point	*none*	
Draw a line `LINE`	(point, point)	line	a line must be drawn between the two points.	
Draw a circle `CIRCLE`	(point, point)	circle	the center of the circle must be the first point and its circumference must pass through the other point	
Draw a circle with the same radius of the previous circle `CIRCLE2`	(point, circle)	circle	the center of the generated circle must be the point and its radius must be the same as the radius of the specified circle	

Generate an intersection
`INTERSECT0`
`INTERSECT1`
	(object, object)	point	the point is on the intersection of two objects	

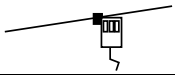
Put a point on an object `ON`	(object)	point	the point must be on the object	

Figure 4: The list of operations of the graphic editor

file is: <*object_identifier*> = *command*(*Arguments*, [invisible|visible]). Commands are listed in Figure 4, and the last argument indicates whether the object should be visible or invisible. This file is also used as a *library* to extend the editor. Using a library, generated visual objects can easily be used as subobjects of another new object, just like primitive objects. For example, an equilateral triangle, once defined, can be used within the graphic editor as a new library. Two equilateral triangles in Figure 6 is drawn easily using the equilateral triangle library.

Code Generator The *code generator* translates the library information file into subclasses of InterViews [1] widgets, using the library information file. Three files are generated: the *header file* contains the class declaration of the generated object, the *body file* contains the methods of the generated object, and the *constraint specification file* contains the constraints that will be handled by the run-time routines. The constraint specification file of the equilateral triangle object is shown in Figure 8. The constraint specification file consists of a set of vector equations. '`Id`*n*' denotes a position vector which corresponds to the objects in Figure 7. '`@Id`*n*' denotes the amount of mouse movement used for approximation by the constraint solver. The equivalent vector equations in a more familiar notation are shown in Figure 9.

Run-time routines The *run-time routines* support the run-time behavior of the constructed visual objects. Central to the run-time routines is the constraint solver; it receives requests for updating the internal state as a result of user interaction, upon which it determines how to change the internal state according to the constraints that define the composite behavior. The run-time routines also support the export value mechanism.

```
2 @Id2 ( Id2 - Id1 ) + 2 @Id1 ( Id6 - Id2 )
 + 2 @Id6 ( Id1 - Id6 )
 + ( Id2 - 2 Id1 + Id6 ) ( Id2 - Id6 )

( Id6 - Id2 ) ( Id6 - Id2 ) - 2 @Id6 ( Id6 - Id2 )
 + 2 @Id2 ( Id6 - Id2 ) - ( Id2 - Id1 ) ( Id2 - Id1 )
 + 2 @Id2 ( Id2 - Id1 ) - 2 @Id1 ( Id2 - Id1 )
```

Figure 8: `triangle.cnt`

$$
\begin{aligned}
&2\Delta\vec{\mathrm{Id}2}\cdot(\vec{\mathrm{Id}2}-\vec{\mathrm{Id}1})+2\Delta\vec{\mathrm{Id}1}\cdot(\vec{\mathrm{Id}6}-\vec{\mathrm{Id}2})\\
&\qquad+2\Delta\vec{\mathrm{Id}6}\cdot(\vec{\mathrm{Id}1}-\vec{\mathrm{Id}6})\\
&\qquad+(\vec{\mathrm{Id}2}-2\vec{\mathrm{Id}1}+\vec{\mathrm{Id}6})\cdot(\vec{\mathrm{Id}2}-\vec{\mathrm{Id}6})=0\\
&(\vec{\mathrm{Id}6}-\vec{\mathrm{Id}2})\cdot(\vec{\mathrm{Id}6}-\vec{\mathrm{Id}2})-\Delta\vec{\mathrm{Id}6}\cdot(\vec{\mathrm{Id}6}-\vec{\mathrm{Id}2})\\
&\qquad+2\Delta\vec{\mathrm{Id}2}\cdot(\vec{\mathrm{Id}6}-\vec{\mathrm{Id}2})-(\vec{\mathrm{Id}2}-\vec{\mathrm{Id}1})\cdot(\vec{\mathrm{Id}2}-\vec{\mathrm{Id}1})\\
&\qquad+2\Delta\vec{\mathrm{Id}2}\cdot(\vec{\mathrm{Id}2}-\vec{\mathrm{Id}1})-2\Delta\vec{\mathrm{Id}1}\cdot(\vec{\mathrm{Id}2}-\vec{\mathrm{Id}1})=0
\end{aligned}
$$

Figure 9: Vector equations of `triangle.cnt`

3.2 Composition of Visual Objects in Oak

When the designer is editing a visual object, there are two alternative schemes whereby constraints could be specified: (i) constraints are constructed from the 'trace' of the operations issued to the graphic editor, and (ii) constraints are extracted from the resulting picture [6, 8, 11, 3]. In Oak, we adopt the former approach for the following reason: the former approach incurs less ambiguity in constraint extraction compared to the latter, because the information of the composition of the visual objects, such as sequence of editor operations, could be used to extract the constraints in addition to the resulting picture. For example, when a knob coincides with a groove, it is not clear whether it is intentional (when the knob was explicitly placed on top of the groove) or accidental (when the groove was inadvertently moved to coincide with the knob) in the latter approach, whereas it could be easily distinguished in the former.

In Oak, each operation of the graphic editor implies corresponding constraints, as described in Figure 4. A non-programmer designer merely 'draws' and composes the visual objects in an intuitive fashion using these operations. Constraints are accumulated and composed when the points in the subobjects are shared by the operations. For example, in the third step of the construction of the equilateral triangle object in the previous section, the designer selects the intersection of the two circle. The system then extracts the following constraints: (i) the distance between the intersection and the center of the first circle should be equal to that between the centers of the two circles, and (ii) the distance between the intersection and the center of the other circle should be equal to the distance between the centers of the two circles. In the fourth step, the designer draws three line segments among the three points; the lines (edges) are constrained to be equal by the extracted constraints.

Composition of the application-dependent actions are achieved simply by considering the union of all export alues and having the appropriate callback functions invoked when one of the export values changes. New callback functions could also be added to a subset of the union of export values.

4 Implementation Issues

4.1 The Constraint Solver

Since the application-independent action of the visual objects in Oak is governed by constraints, the constraint solver must be robust enough to express rich classes of visual objects and still be fast enough to realize interactive response, as the efficiency of the constraint solver directly affects the performance of the entire application program. More specifically, the constraint solver must satisfy the following requirements:

- The solver must be efficient enough so that the complex visual objects can follow the motion of the mouse cursor in real-time.
- The solver must be able to handle non-linear equations. This is important because some primitive constraints, such as equality of distances, are non-linear.
- The solver must be able to solve multi-way as well as cyclic constraints. This is beneficial for allowing arbitrary compositions.

It is difficult to construct a constraint solver satisfying all the abovementioned requirements, because expressive power and speed are contradictory requirements in constraint solving. In order to realize the interactive response, previous approaches have been to restrict the power of the constraint solver by employing propagational algorithms (e.g., [4, 17]).

In Oak, we take a different approach, taking into account the characteristics of mouse interaction: (i) the movement of the mouse is mostly continuous, and (ii) when the user is making a rapid mouse movement, errors incurred in constraint solving will be nearby indistinguishable, as long as the visual objects follow the mouse cursor. When the user is making a precise adjustment, and/or when the user has finished the interaction, the exact shape of the visual object needs to be computed and displayed. Under such observation, we provide two kinds of constraint solvers: one solver approximates non-linear equations with linear equations, utilizing the nature of continuity of dragging. It is very fast but embodies some inaccuracy; this is called the *approximate solver*. The other solver is accurate but is slower; this is called the *accurate solver*. While the user drags the object on the display, the approximate solver satisfies the constraints and decides its behavior. When the user finishes the interaction, the non-linear equations are exactly solved using the accurate solver, and the display is updated to match the solution.

In the approximate solver, non-linear equations are approximated as follows: when the user drags the object, we consider the dragging as a summation of its minute movements. Ignoring the second-order terms of the minute movements, the non-linear equations are approximated with linear equations.

To give a simple example, line segments AB and BC are orthogonal to each other, and their lengths are equal (Figure 10 (a)). When the user drags Point A, the line segments move to maintain the constraint. The equations that express the constraints are approximated with linear equations in advance by the code generator. At run-time, mouse movements occur successively while the user drags Point A. Every time an event occurs, the approximate solver solves the linear equations, and moves the line segments according to the solution (Figure 10 (b)). When the dragging is

finished, the exact solution is found by the accurate solver (Figure 10 (c)).

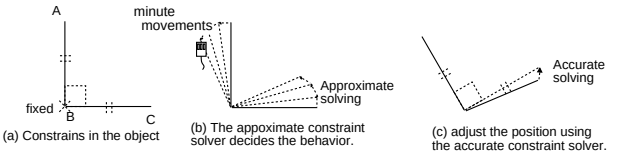


Figure 10: Two orthogonal line segments of equal lengths

So far, the examples constructed with our prototype have behaved remarkably well. Real-time response has been maintained for the examples given in this paper. Even experienced GUI users did not initially guess that two different solvers are actually being used.

4.2 Generalization of Operations for Constraint Extraction

Oak basically translates operations of the graphic editor *one-to-one* into corresponding constraints. There are some cases, however, in which generalization on the number of the operations are necessary. For example, suppose that the designer wants an arbitrary number of parallelograms. Without generalization, it is impossible to express constraints on an arbitrary number of visual objects by a finite number of examples.

Previous GUI systems based on the *programming by example* paradigm have supported generalization from the examples provided by the designer. Specifically, Peridot [6] generalizes operations on a given list of user items, and both Eager [2] and Metamouse [5] perform loop synthesis of dynamic interactions performed by the user at run-time. The Oak system performs loop synthesis at object construction time by generalizing on the number of repeated operations when it detects the repetition of the following pattern of commands: predefined library commands are issued whose arguments are the points generated by prior 'put a point' commands.

As an example, let us consider the designer drawing a ribbon of connected parallelograms (Figure 11). We assume that the graphical editor already has the parallelogram library command. The designer draws the first parallelogram, then draws the next one so that its two vertices share points with the first one. The resulting library information file would be as follows:

$$
\begin{array}{ll}
p_1 = \text{Point}, \; p_2 = \text{Point}, & \\
p_3 = \text{Point}, & c_1 = \text{Parallelogram}(p_1, p_2, p_3) \\
 & \quad \text{— point } p_4 \text{ is generated} \\
p_5 = \text{Point}, & c_2 = \text{Parallelogram}(p_4, p_3, p_5) \\
 & \quad \text{— point } p_6 \text{ is generated} \\
p_7 = \text{Point}, & c_3 = \text{Parallelogram}(p_6, p_5, p_7) \\
 & \quad \vdots
\end{array}
$$

This library information includes a pattern which is generalized: the 'put a point' commands are issued three times immediately followed by alibrary command that uses the three points as arguments. This pattern is repeated, and Oak generalizes the above information and generates the following library information file, where the arbitrary number of parallelogram generations can be done with a loop:

$$
\begin{array}{l}
p_1^1 = \text{Point}, \; p_2^1 = \text{Point}, \; p_3^1 = \text{Point} \\
c_1 = \text{Parallelogram}(p_1^1, p_2^1, p_3^1) \\
\qquad\qquad \text{— point } p_4^1 \text{ is generated} \\
\textbf{loop}\,(i = 2, 3, \cdots)\{ \\
\quad p_1^i = \text{Point}, \; c_i = \text{Parallelogram}(p_4^{i-1}, p_3^{i-1}, p_1^i) \\
\}
\end{array}
$$

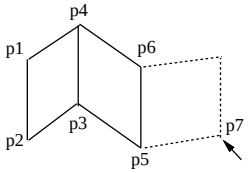


Figure 11: Generalizations of connected parallelograms

5 Further Examples

In order to illustrate the expressive power of Oak, we constructed a less conventional radar chart[2] with six arms. The application using the radar chart widget can display six numerical data, and the user can manipulate them with a mouse. The graphs are updated in response to the user's interaction.

Construction of the widget consists of the following four steps: First, we draw the picture of a radar chart as shown in (Figure 12). Second, we designate the export values. In the example, six distances between the center and the six arms are the export values. Third, we generate the library information file. We select the 'Make Library' command in the 'Library' menu, and then input the library name, and the system generates the file. Finally, we translate the library information file into the three files as mentioned previously, which are linked with the application program along with the run-time routines.

6 Conclusion

We have proposed the BVI (Behavior, Visualization function, and Internal state) model, in which GUI designers can arbitrarily compose new *visual objects* (extended widgets) recursively from intrinsic primitive

[2] A radar chart displays the balance of some related attributes. For example, a radar chart may be employed to display the balance of a person's nutrition.

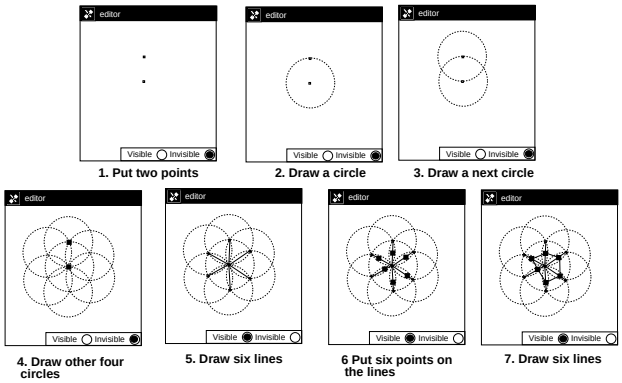


Figure 12: Snapshots of the radar chart construction

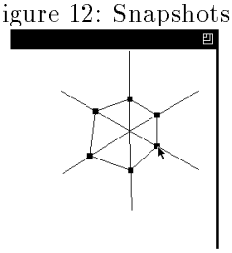


Figure 13: Testing the finished radar chart

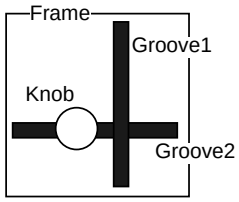


Figure 14: A cross slider object

objects using direct manipulation. With this framework, a variety of visual objects can be easily constructed by simply assembling the subobjects without requiring coding knowledge. Moreover, this framework clearly separates the application-independent actions from dependent ones, by the use of robust constraints. A prototype system Oak allows GUI designers to compose visual objects by direct manipulation. The designer first draws a visual object on the graphic editor that provides elementary geometric operations. Constraints between subobjects are extracted from the trace of operations of the graphic editor, whereafter the code generator compiles it into C++ classes which are subclasses of InterViews [1]. The resulting visual objects are available for use in application programs. To facilitate interactive response while providing a rich set of specifiable constraints, a fast approximated constraint solver has been developed as a part of the runtime system.

As a future work, we intend to extend our prototype in the following ways: Firstly, it is difficult to allow an operation to temporarily break a constraint, because constraints express relationships which must always be maintained. Such temporary non-satisfaction is beneficial in describing the dynamic behavior of visual objects: for example, given two boxes aligned horizontally, exchanging the boxes moving smoothly along two arcs temporarily breaks the alignment constraint. It may be possible to handle non-satisfaction properties with the use of hierarchical constraints [4], but more work needs to be done. Secondly, composite behavior can be specified only by supplementing constraints with Boolean 'and' relation in our current system. By allowing 'or' relations, the class of specifiable visual objects can be enriched. For example, the cross slider object in Figure 14 can be constructed, which is not supportable under the current Oak prototype because the knob must be constrained to both grooves simultaneously.

References

[1] Paul Calder, Mark Linton, and John Vlissides. Composing user interfaces with InterViews. *IEEE Computer*, 1989.

[2] Allen Cypher. Eager : Programming repetitive tasks by example. In *ACM Human Factors in Computing Systems*, pages 33–39, 1991.

[3] Steven Feiner David Kurlander. Inferring constraints from multiple snapshots. Technical report, Department of Computer Scence, Columbia University, 1991.

[4] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, Jan 1990.

[5] David L. Maulsby, Ian H. Witten, and Kenneth A. Kittlitz. Metamouse: Specifying graphical procedures by example. *ACM Computer Graphics*, 23(3):127–136, July 1989.

[6] Brad. A. Myers. Creating highly-interactive and graphical user interfaces by demonstration. *Computer Graphics*, pages 249–258, 1986.

[7] Brad A. Myers. Separating application code from toolkits: Eliminating the spagetti of call-backs. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, 1991.

[8] Brad A. Myers. Text formatting by demonstration. In *ACM Human Factors in Computing Systems*, pages 251–256, 1991.

[9] Brad A. Myers et al. Garnet, comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, pages 71–85, November 1990.

[10] Ken Nakayama, Satoshi Matsuoka, and Satoru Kawai. Visualization of abstract concepts using generalized path binding. In *Proceedings of CG International '90*, pages 377–402. Springer-Verlag, 1990.

[11] Robert P. Nix. Editing by example. *ACM Transactions on Programming Languages and Systems*, 7(4):600–621, October 1985.

[12] Yen-Ping Shan. MoDE: A uims for smalltalk. In *OOPSLA '90 Conference Proceedings*, 1990.

[13] I. E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proc. AFIPS Spring Joint Conf.*, volume 23, pages 329–346, 1963.

[14] Shin Takahashi, Satoshi Matsuoka, Akinori Yonezawa, and Tomihisa Kamada. A general framework for bi-directional translation between abstract and pictorial data. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, November 1991.

[15] Toshio Tonouchi. Creating visual objects by direct manipulation. Master's thesis, The Univ. of Tokyo, 1992.

[16] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 158–167, 1989.

[17] Bradley T. Vander Zanden. *Incremental Constraint Satisfaction And Its Application To Graphical Interfaces*. PhD thesis, Department of Computer Science, Cornell University, October 1988.