Visual Programming in an Object-Oriented Framework

Philipp Ackermann Dominik Eichelberg Bernhard Wagner
[ackermann,eichelberg,wagner]@echtzeit.ch

echtzeit-perspectix ag
Sonneggstrasse 76
CH-8006 Zürich

MultiMedia-Laboratorium
Institut für Informatik der Universität Zürich
Winterthurerstrasse 190
CH-8057 Zürich

Abstract

Instead of scripting, a visual programming approach was explored for a component runtime environment. It provides generic access to framework objects and supports black-box reuse through interactive assembly. This paper describes the execution model of this intuitive composition approach and outlines useful applications that deal with changing requirements, short prototyping cycles, and user-friendly system reconfigurations.

Introduction

Visual programming languages [Burn95] which represent data and their flow with pictures rather than text (source code) are not widely used because the specification of algorithms based on a visual data flow is very limited. The visual representation of procedural abstractions is not appropriate for large systems. This disadvantage is reduced in object-oriented systems because objects encapsulate state (data) and behavior (methods/algorithms) so that visual programs are no longer used for algorithmic specification, but may be applied for object composition by defining inter-object relations.

The MET++ multimedia application framework [Acke96] is a C++ class library that contains several frameworks. A framework is a reusable and extendable abstraction built of preintegrated, collaborating objects. Developers may extend a framework by subclassing existing abstractions and adding their specific functionality. Because the internals of parent classes must be understood by the developer, subclassing is called white-box reuse. Like many other frameworks, MET++ matured over time and, besides generic abstractions, a lot of ready-to-use classes were developed. New functionality is often obtained by instantiating and composing objects without knowing their internals. In MET++, this black-box reuse is supported 1) by framework- or domain-specific editors, and 2) in a generic approach by a visual programming environment [Wagn96] in which configurations of objects are dynamically assembled.

Visual Component Runtime Environment

We understand a component as an object or object cluster which is embedded in a run-time environment that provides functionality and dynamic behavior to the component. Frameworks often integrate such a runtime environment. For example. a presentation framework supports the display, event handling, and automatic layout of graphical components (GUI elements) based on hierarchical grouping; a synchronisation framework manages the temporal presentation of animations and time-dynamic media (audio, video) based on temporal composition structures; an audio framework handles the signal processing based on a source-filter-sink media flow. In such frameworks or component environments, configuring applications is mainly realized by object composition. To avoid long compile/link cycles, component environments often provide an interpreted scripting language to support dynamic object configuration. Due to the following two reasons, scripting is not favored in MET++:

1) Considering the end-user's view-point, the formal approach of scripting languages is not easy to understand. A point-and-click interface with the relations clearly evident from visual cues is more intuitive and therefore more user friendly than scripting.

2) Opening a class library to scripting introduces countless stub procedures to order to make the important methods of existing classes accessible to the scripting language. Furthermore, two separate protocols need to be kept consistent, and they introduce redundancy which may confuse the developer if the protocols are (slightly) different to use.

The *visual component runtime environment* of MET++ is an orthogonal extension to the existing frameworks. It sits on top of the MET++ class libraries as an additional layer and provides a generic run-time environment for object composition. The execution model is based on a data flow approach. A visual program graphically represents components and their relations. Relations are defined by wires between input/output ports. There are no messages or objects passed between components. Components communicate with each other by sending simple data types such as integer/float values or character strings through the wires. A port typically encapsulates an instance variable of an object or a parameter of a function.

Visual Wrappers for Framework Objects

The visual representation of a component is called *data unit*. The `DataUnit` class may operate as a wrapper to a framework's object or as a wrapper to a procedure. A data unit models a wrapped object as a mathematical function with independent and dependent variables according the expression $(d_1, d_2,\ldots, d_n) = f(i_1, i_2,\ldots, i_m)$. As an approximation, independent and dependent variables can be seen as input and output values. A data unit transforms the input values to output values by calculating the dependent variables as a result of a function of the independent variables. A data unit evaluates its "function" and fires its dependent values as soon as all independent values are set. Some data units trigger their evaluation on specific events, e.g. on user interaction, timer interrupts, or on state changes in an object (change propagation). Since the flow in the visual program is *bidirectional*, a data unit also has an "inverse function". Filters which transform input-output values in both directions are an obvious representation in this model. Examples are mathematical transformations (multiply/divide; sin/arcsin; …) and control structures (if-then-else, threshold). Wrapped objects are also modeled as *functional* representations. A wrapper of a GUI element such as a knob evaluates its "function" in case of a user interaction or if its independent variable is triggered. The "function" gets the value of the knob and sends it out of the dependent port. But the knob as GUI element can also work as a visual indicator. If the dependent port receives data, the "inverse function" evaluates the data value and sets the rotation of the knob accordingly. The independent port of the knob wrapper works as trigger to fire the knob's value. An independent port can also act as an *index* into a list. Depending on the received value, the result of the output variables changes. A relation in a data base can therefore be modeled as such a functional abstraction.

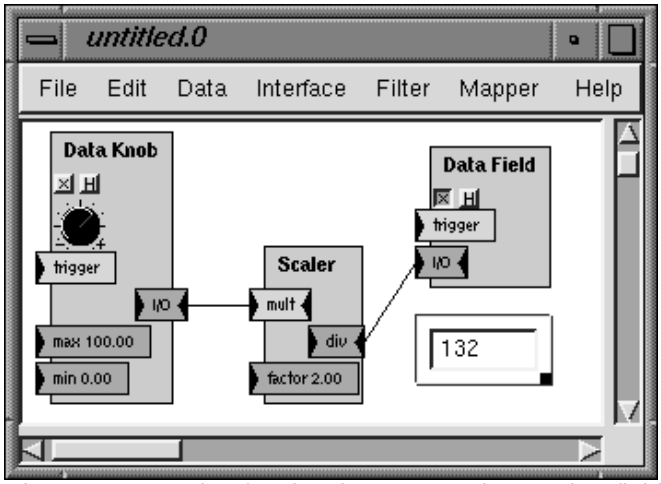

Figure 1: Example of a visual program. The number field is detached from its data unit.

A port holds meta information about its data type, value domain, and step size. It is therefore possible to actively iterate over a port's domain. In Figure 4, the text and 3D mappers iterate over a 2-dimensional array to visualize its data. Because only simple types are sent through the communication wires, and ports additionally provide automatic type conversion, a port can be connected to any other one without any restriction. This generic feature allows a highly flexible configuration of components even from different frameworks. GUI components can be combined with temporal media components, data containers can be connected to generic filters, 3D objects can be related with external devices (e.g. MIDI devices) that are wrapped by a data unit, etc. Developers can add their own type of data unit by subclassing from an abstract `DataUnit` class which handles the ports and the triggering mechanism. New wrappers are built by adding the needed ports and by overwriting the `EvaluateDependent()` and `EvaluateIndependent()` methods.

Interactive Component Editor

Visual programs are interactively realized in a specific editor (Figure 1) which displays a component diagram with wires connecting ports. Ports are titled with a name prompting their meaning to the user. Data units (wrappers) are generated by pull-down menus or by drag&drop of an object into the editor window. All visual programs are "live": they are fully operational without compile/link cycles. Graphical user interface (GUI) elements can be separated from the visual program. Within the pictorial representation of a data unit, GUI components (e.g. knobs, buttons, sliders, edit fields, ...) are detached by clicking on the left button. A detached GUI element can then be placed in any other window by a drag&drop operation. The appearance of GUI components can be visually programmed. E.g., the radio buttons on the left side of Figure 2 were defined by sending character strings to its "append" port. Text entries can be removed by sending a number as index to the "delete" port. Check boxes and pop-up menus are defined in the same way.

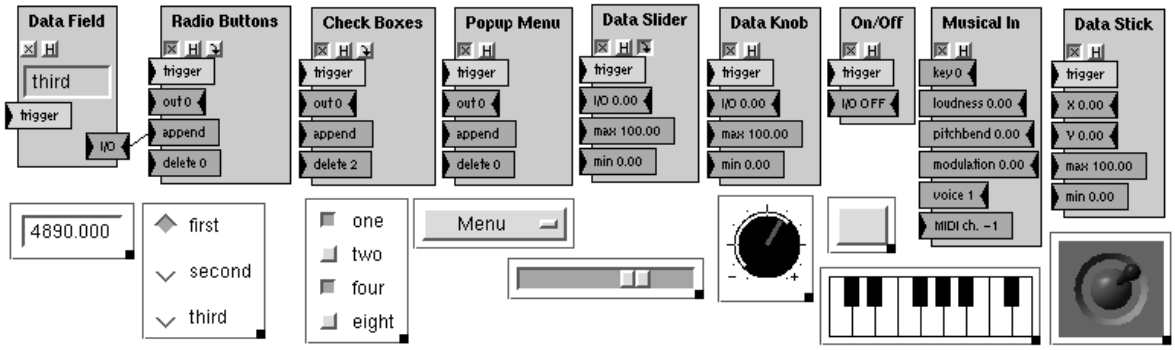

Figure 2: Examples of GUI components wrapped as data units.

Application Scenarios

The visual programming enviroment provides several wrappers to existing classes of the MET++ multimedia application framework. It includes data units for visual objects, user interface components, 3D graphics objects, cameras, lights, and time-dynamic media types such as animations, audio, note sequences, etc. A visual program and its wrapped objects can be stored to a file which we call an *executable document*. By reading an executable document, all saved objects and their relations are instantiated and work immediately (Figure 3). The visual program approach of MET++ is used for 1) the definition of multimedia presentations, 2) data visualization and sonification as seen in Figure 4, 3) the creation of interactive and dynamic virtual reality scenes, 4) prototyping of graphical user interfaces, and 5) as a generic media patcher for realtime performances. All these applications profit from the intuitive, fast and playful access to the properties of objects and the ability to interactively define relations, constraints, interactions, and dynamics without having to write code. Figure 3 shows an example of an executable document created with an MET++-based multimedia authoring tool. A WWW browser which can present such executable documents embedded in HTML pages was a best of show finalist at CeBIT '96 [BYTE96].

Conclusion

Our visual programming enviroment, which is based on a data flow approach, has proven that component-oriented composition does not have to be a language feature but can be built as layer on top of existing libraries and applications. Visual wrappers to object-oriented structures allow the interactive configuration of object relations even for end-users.

References

[Acke96] Ackermann, Ph., "Developing Object-Oriented Multimedia Software – based on the MET++ Application Framework", dpunkt-Verlag, Heidelberg 1996.

[Burn95] Burnett, M., M., Goldberg, A., Lewis, T., G., "Visual Object-Oriented Programming", Manning Publications, Greenwich, 1995.

[BYTE96] International BYTE magazine, “Best of CeBIT ‘96”, p. 32, June 1996.

[Wagn96] Wagner, B. et.al., "Black-Box Reuse within Frameworks based on Visual Programming", Proceedings of the Component Users's Conference '96, München 1996.

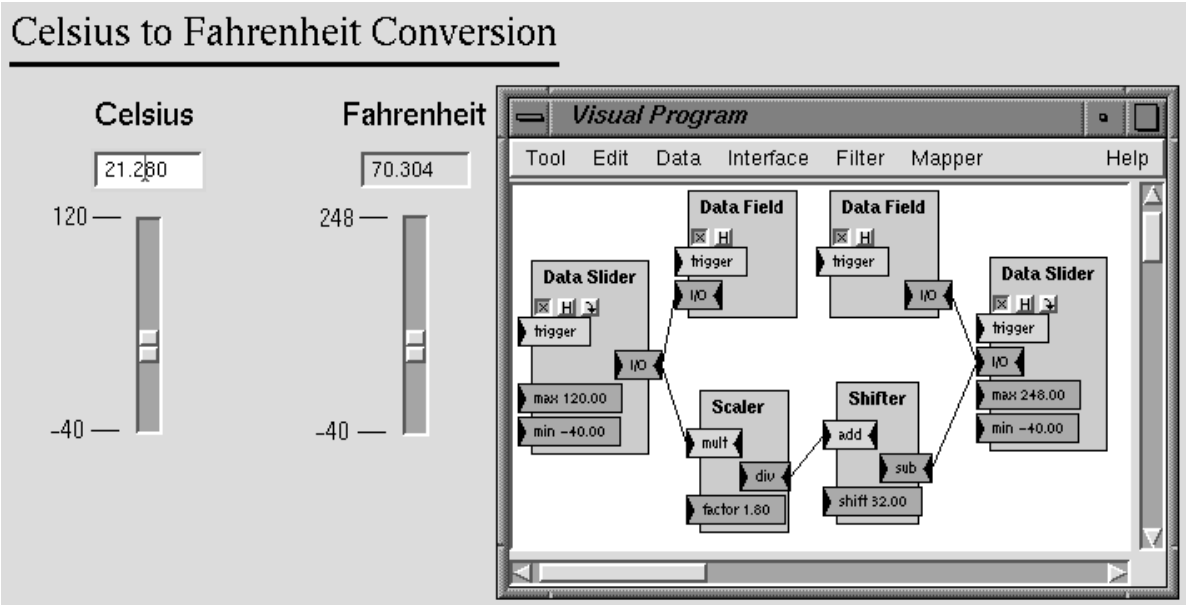

Figure 3: A visual program that implements Celsius to Fahrenheit conversion. GUI components of the data units are detached and placed into another window with drag&drop. If this "executable document" is loaded, the visual program will not be displayed.

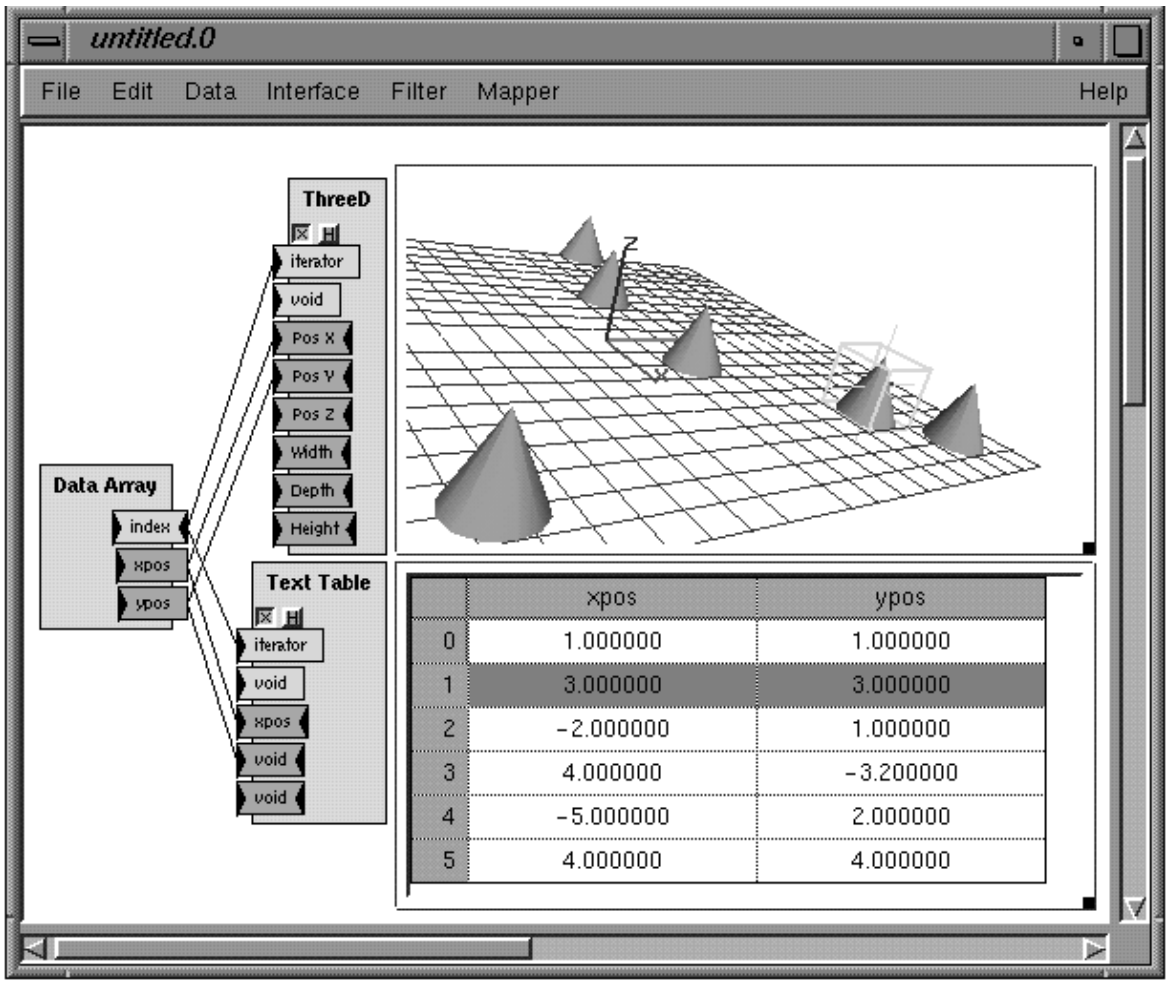

Figure 4: Bidirectional data flow between a data array, a 3D view, and a spreadsheet in a data visualization application. The mappers automatically iterate over the index of the two-dimensional array and dynamically create the corresponding visual representation. Moving a 3D object will immediately be reflected in the spreadsheet and vice versa.