Black-Box Reuse within Frameworks based on Visual Programming

presented at CUC '96, Munich

Bernhard Wagner	bwagner@ifi.unizh.ch
Ian Sluijmers	sluijm@ifi.unizh.ch
Dominik Eichelberg	eichel@ifi.unizh.ch
Philipp Ackerman	ackerman@ifi.unizh.ch

University of Zurich
Dept. of Computer Science
MultiMedia Laboratory
Winterthurerstr. 190
CH-8057-Zürich
Switzerland
VOICE +41-1-257-4569
FAX +41-1-363-00-35
http://www.ifi.unizh.ch

Abstract

Application frameworks allow structured reuse of object-oriented design and source code, provided that the developer understands the source code and has knowledge of the framework's design conventions. The notion “white-box reuse” refers to the process of developing software by writing subclasses with the knowledge and understanding of the internals of the parent classes. When applying black-box reuse however, new functionality is obtained by composing objects without knowing their internals, only their interface. We took the idea of object composition a step further by developing a visual programming environment for the easy reuse of a multimedia application framework's classes. Instantiations of these classes are represented as black-box components in our visual programming environment.

1. Introduction

1.1. The Multimedia Application Framework MET++

Our visual programming environment is based on the existing multimedia application framework MET++ [Acke95] which is an extension to the ET++ application framework [Wein92, Gamma92]. Event handling and message passing between the framework's objects fall under the responsibility of the framework and are already preimplemented. Within multimedia applications, events occur as discrete data resulting from user interaction, from data being read off storage media and from external input devices such as mouse, keyboard, and MIDI instruments. Values generated by time functions controlling the temporal behaviour of media are also considered as events. Most standard multimedia application functionality is provided, including time synchronisation and multimedia-specific user interaction besides multiple undo of commands and standard editor functions such as cut/copy/paste and drag&drop. A variery of editors is readily available for all supported media, so the media-specific manipulation can be reused by the application programmer.

1.2. Object Composition in Application Frameworks

Black-box reuse or reuse by instantiation is easier to apply than white-box-reuse, since the internals of the involved classes don't have to be understood, only their interfaces. If several objects are linked in a black-box manner, the term *object composition* is often used. Through object composition it is possible to change a constellation of objects at runtime. Also, object composition is a valuable alternative to multiple inheritance when properties of several classes should be combined. In ET++ and MET++ object composition is heavily used. While black-box-reuse is easier to apply than white-box-reuse, the understanding of the interplay between instantiated black-box-components is harder than understanding an inheritance hierarchy by looking at the source code of the involved classes. For this reason runtime-debugging support has been integrated into ET++, that allows to display the actual composition of objects in a running application [Gamma92].

1.3. Visual Object Composition

Our experience showed that programming of applications based on application frameworks like MET++ is hard to learn. The difficulty to teach and encourage black-box-reuse of MET++ classes lead to the idea of representing the reusable objects as graphical components with their interconnections depicted as wires. This way of presenting the possibilities and supported media of MET++ is much more playful and intuitive than confronting a newcomer with the bare C++ interfaces. The idea of object composition is presented in an interactive graphical way.

1.4. Wrappers

To make the existing framework's classes available in the visual programming environment we applied the Wrapper or Adapter design pattern [Gamma95]. Wrappers are used to offer a different interface for an existing class. The Wrapper-technology has successfully been applied before in MET++ for wrapping multimedial time-dependent data streams, e.g. audio, MIDI, video, time functions for the control of geometric aspects of 2D- and 3D-graphics. The new interface allows wrapped classes to be combined into a generic temporal layout system. Whenever a new time-dependent object is developed and properly wrapped (i.e., by implementing a set of abstract methods for start, stop, calculation of durations, etc.), it can be used in the existing temporal layout system without changing the latter [Acke95]. The visual programming environment also requires a specific interface, so the Wrapper design pattern was applied here as well. Now all media that have been prepared for usage in the visual programming environment have two kinds of wrappers: a temporal wrapper and a visual programming wrapper. These two kinds of wrappers represent orthogonal access methods to the wrapped media. An application of this orthogonality is shown in Figure 1. The temporal wrapper shows an (interpolated) time function to control the z rotation of the small gear in MET++'s Time Composition View. The Visual Program below it shows two data units that wrap the two gears. The small gear is controlled by the time function and the changes are propagated through a scaling of -0.5 to adapt the rotation speed for the larger and slower gear. In a conventional animation system the two gears would have to be animated separately and if one gear's speed would change, the speed of the other would have to be adapted manually. In our environment the *relations* between objects are specified in the visual program and only the keyplayers are animated. The relations defined in the visual program are maintained and so allow to specify animations redundancy-free. Animations can be specified and manipulated in a more structured way than animating each object with its own time function.

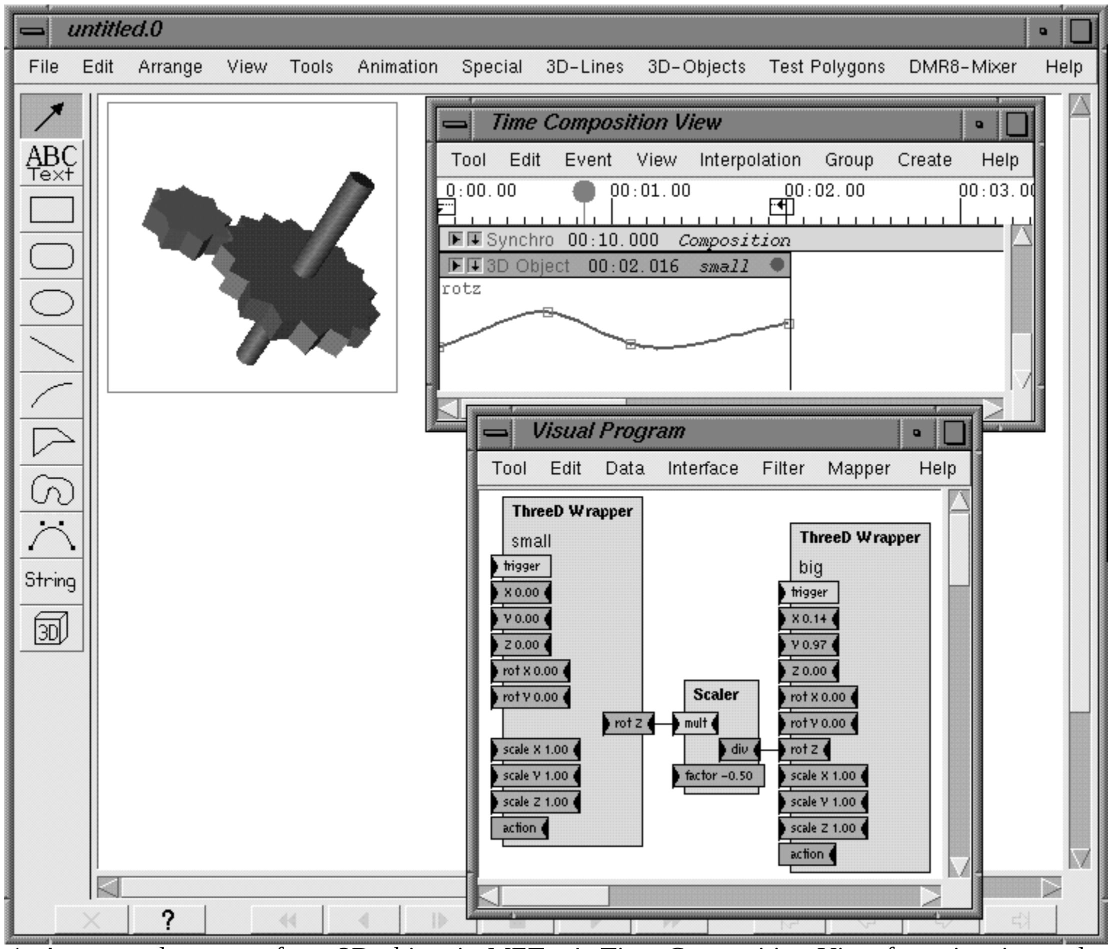


Figure 1: A temporal wrapper for a 3D object in MET++'s Time Composition View for animation and two wrappers for manipulation from within a Visual Program.

2. Building Blocks

A visual program in our environment is defined by a data flow diagram. The components of a visual program are *data units* receiving, generating or processing events. New instances of data units which wrap objects of the framework can be created in a specific editor with menu and drag&drop commands. They appear as two-dimensional graphic representations and can be wired on the screen. Bi-directional connections are made between ports belonging to different data units to allow communication between them. Ports have a title prompting their meaning to the user and an arrow-like representation which determines whether they are meant for input, output or both.

Object wrappers are grouped according to their purpose, i.e., *data repositories*, *filters*, and *mappers*.

• *Data repositories* include all kinds of discrete data including a number of file formats, arrays, single data elements, constants, user interface components, and system functions (e.g. timers) that generate events and hardware port access, e.g. reading input from MIDI and audio ports.

• *Filters* are a collection of data processing units. They perform functions on data either in the mathematical sense or as control structures with built-in conditions (e.g. if-the-else and thresholds).

• *Mappers* provide capability to map data to different kinds of representations (e.g. 3-D graphics, 2-D bitmaps, text tables, audio, and MIDI output).

3. Executable Documents

Certain wrappers include user interface components that can be detached from the data unit's representation. They may even be separated completely from the visual program by dragging-and-dropping them into a dedicated user interface window. A button is provided to highlight the component if it needs to be located among others. These features provides the basis for a flexible user interface builder where executable documents can be created. Executable documents contain multimedial data types such as text, GUI elements, audio, animations, etc. whose behaviour is controlled by visual programs. The appearance of GUI components is also 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.

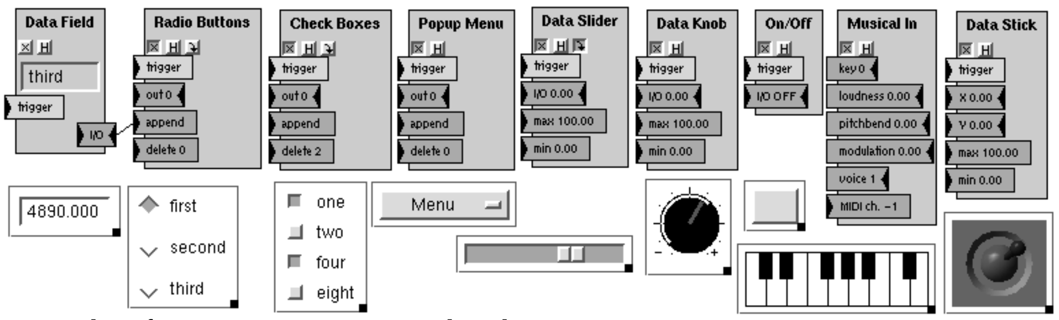


Figure 2: Examples of GUI components wrapped as data units.

A simple typical example for an executable document is the Celsius to Fahrenheit converter shown in Figure 3. Dynamics of such executable documents rely on object composition only, no compiling, linking, or script interpretation is needed. As soon as the user creates a new instance of an object in the visual editor, it “comes to life” and reacts on any messages passed to it or any interaction via its user interface. In this sense, visual programming moves up to a higher level of abstraction as the future user of a program and its developer can communicate interactively while implementing and designing the software simultaneously. The specification and the implementation are no longer separated.

We consider these visually programmed executable documents as an alternative to Web-programming like Java and CGI. While these languages still require programming skills, our environment allows to visually specify a page's behaviour. A Web-browser that supports this kind of executable documents as plug-ins has been developed.

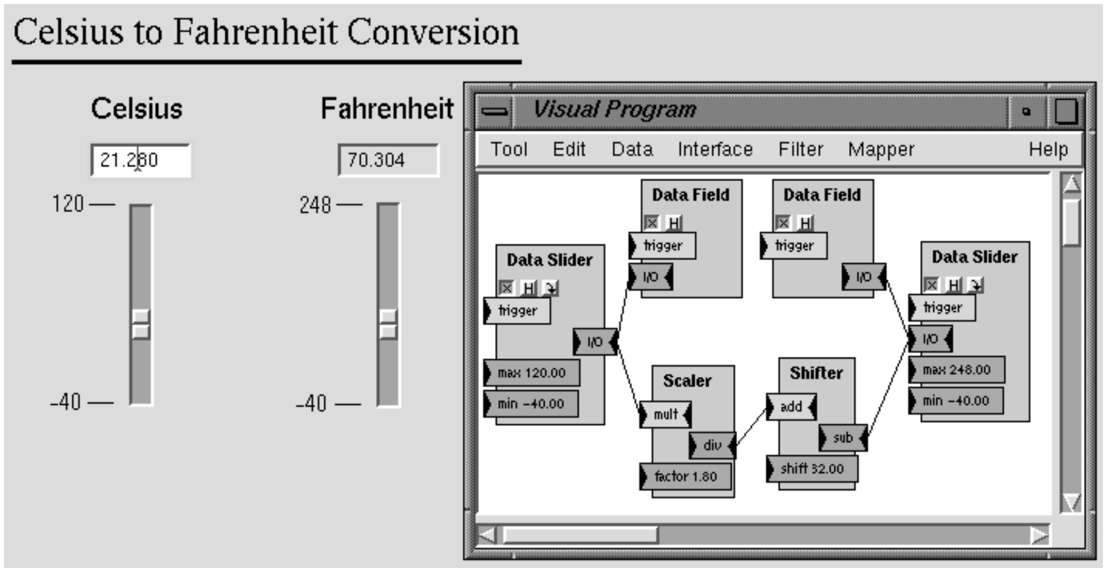


Figure 3: A visual program that implements Celsius to Fahrenheit conversion. GUI components of the data units are detached and placed into another window with drag&drop.

4. Data Flow

The object-oriented principle (making the objects responsible for their functionality and only passing messages) is transferred to the design of our visual programming environment. This does lead to specific problems concerning the order of execution in a program. Triggering and data type conversion is strictly handled locally within the ports of the wrapper. Data types are kept simple. Only standard numeric and alphanumeric types are used. No sending of scripts or specific synchronization triggers is needed. A locking mechanism prevents messages from looping endlessly in cyclic connection patterns. This would occur due to back and forth triggering as events are passed bi-directionally.

4.1. Bidirectionality

Data flow diagrams and similar representations usually contain *directed* connections, allowing the flow of data in one direction only. Our approach provides bidirectional connections between data units allowing them to pass messages in both directions regardless of their position in the data flow. This implies bidirectional behaviour of data units.

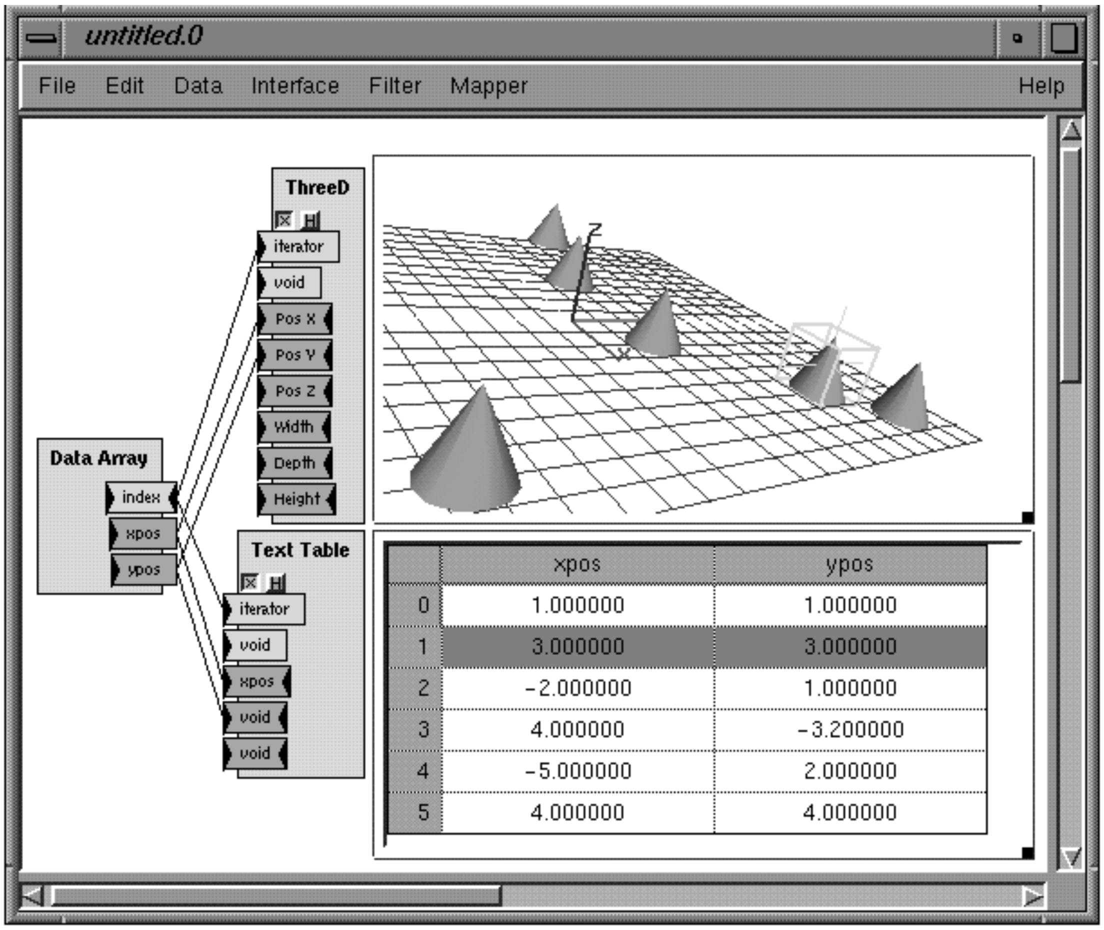
Data are units acting as data repositories (array, NetCDF file [NetCDF], data base), event generators (timer, clock, music in/out, random, audio in/out) and user interface components (entry fields, sliders, knobs, joystick, different kinds of buttons). Bidirection is possible for repositories: They can be read and written. User interface components are bidirectional as well: if a certain value is fired at their I/O-port, the corresponding state is depicted in the visual component. Event generators do not allow bidirecion.

Filters need to be bidirectional, because they are placed in the path of data flow. Data can pass through them both ways. If the *Filter's* function has an inverse function defined, it will be performed if data arrives at the *output port* instead of the *input port*. As discussed in section 1.4. a class has to be wrapped to make it compatible with the visual programming environment. Wrapping here mainly consists of implementing the abstract methods `EvaluateIndependentValues()` and `EvaluateDependentValues()` which are called when all dependent or independent ports have been set, respectively. In the case of a filter for calculating the sinus function `EvaluateDependentValues()` calculates the sin(x) function and `EvaluateIndependentValues()` calculates the arcsin(y) function using the provided independent value x, or dependent value y, respectively.

Mappers are designed to read data from a set of input ports and map it to a specific visualization or sonification. This interface generally provides an interactive mode, allowing the manipulation of the visual representations. The resulting events will be sent out of the ports of the mapper invoking an update of any connected object.

Bidirectional behaviour can be inhibited for special situations. For example, it may not be desirable to write to certain files or to change the system time.

The following example illustrates the bidirectional behaviour of mappers (Figure 4): Given an array containing five sets of 2-D data. It is represented by an independent port ranging from 0 to 4 and two dependent ports, one supplying x-values, the second supplying y-values. To map this array to a three-dimensional representation, the mapper's iterator port is connected to the index of the array, the dependent ports one by one to the x- and y-ports of the mapper. As a result five 3-D objects would become visible each at their own co-ordinates in the 3-D space. Each dependent port of a mapper has its default value in case nothing was connected to it. In Figure 4 the z-coordinate remains zero, width, depth, and height remain 1. The array is also connected to a text table mapper. The 3D elements can be moved within the 3D space to change values of the array. Such changes are immediately reflected in the table. Vice versa, editing of values in the table will force updates of the 3D view.



	xpos	ypos
0	1.000000	1.000000
1	3.000000	3.000000
2	-2.000000	1.000000
3	4.000000	-3.200000
4	-5.000000	2.000000
5	4.000000	4.000000

Figure 4: Bidirectional data flow between a data array, a 3D-view, and a spreadsheet.

4.2. Triggering of Data Units

Some visual programming environments use specific control types, e.g. "BANG" in Max [Zica95] and "PING" in HP VEE [Helsel95]. This kind of explicit control information can be useful in developing real-time applications, but has the disadvantage that it has to be handled separately. It implies another step of thinking in developing programs, because the firing of data can result in bugs, if a function is programmed to fire before it has received all the data it needs.

Our approach has neither specific control types nor is the order of execution dependent on the placement of the objects in the editor. It is a simple philosophy: everything is handled locally within the data units. Each port has a trigger that is either set or reset. A data unit will fire whenever all its independent ports or, in the other direction, when all its dependent ports have received data, meaning they have been triggered. The underlying object's firing method will call the firing method of the connected objects, whose firing method will do the same and so on until every object has fired. Since links are bidirectional, this could result in a back-and-forth firing ad infinitum. To avoid this, a locking mechanism is implemented in the port. Infinite loops can be avoided this way and "wild" cyclic connection patterns cause no harmful effects.

4.3. Independent and Dependent Ports

In the editor of our visual programming environment, data units provide ports for their interconnection. Common data units use two types of ports: *independent* and *dependent*. The type is indicated by a defined colour of the port's representation. For data units acting as data repositories (*Data*) and representing the contents of files and arrays, a set of independent and dependent ports is generally used for addressing the data. The independent ports work like indices of an array (or record). The dependent ports behave like the array's values. A data unit evaluates its dependent values as soon as all its independent ports' triggers are set. Still considering the array example, this would mean that the value or the set of values at the indexed position in the array would be fired as soon as all its indices are set. To maintain the bidirectionality discussed in 4.1, the firing can be induced by triggering the dependent ports with a set of values. As soon as they are all triggered, the corresponding independent indices are searched and fired from the independent ports.

Filters typically are functions, a great many of them are mathematical functions. For simplicity reasons, only one-dimensional filters are described (and implemented so far): $y = f(x)$. x is the independent value, and y is dependent. The filters have one independent and one dependent port. Their triggering behaviour is simple: when a value x arrives at the independent port, $f(x)$ is fired from the dependent port. The opposite direction may be possible (if the corresponding inverse filter function $x = f^{-1}(y)$ is defined) and if so, $f^{-1}(y)$ is fired from the independent port.

Mathematically expressed, the relationship between independent and dependent ports can be represented as a set of n functions of arity $m >= 1$:

$$(y1, y2, ..., yn) = f(x1, x2, ..., xm); (1)$$

where x_i are the independent and y_j the dependent variables and n is not necessarily equal m. Each of the independent variables x_i has its own domain d_i with cardinality X_i and each of the dependent variables y_j its range r_j with cardinality Y_j. In different terms, the set of ranges r_j can be regarded as a family indexed by the cartesian product (product set) d of the domains d_j:

$$f: d1 \times d2 \times ... \times dm \rightarrow r1 \times r2 \times ... \times rn; (2)$$

The cardinalities D and R of the domain product set d and the range product set r, respectively, are calculated as follows:

$$D = X1 * X2 * ... * Xm; (3a)$$

$$R = Y1 * Y2 * ... * Yn; (3b)$$

This leads us to a specialization of the independent port: the *iterator*. It extends the role of the independent port allowing *data mappers* to be built.

4.4. Mapping and Iterators

Mapping is invoked as soon as a data unit is connected to a data mapper. For each independent port x_i of the data source, an iterator port is allocated in the data mapper. Each iterator analyzes the independent port it is connected to and registers its domain d_i. The cartesian product P of these domains is built, and D representation entities according to (3a,b) are prepared. Next, the elements of d (a set of index-sets) are stepped through and fired from the iterators to the data unit connected to the data mapper. The data unit responds by sending its dependent values corresponding to the received index set. The dependent values are then applied to the representations influencing their properties.

Another example will illustrate the mapping process (Figure 5): Two arrays, one containing six, the other containing four sets of values, are connected to an *image mapper*. Each array has one independent index port. The *image mapper* prepares D = 6 * 4 = 24 pixels in a bitmap representation and then iterates over all the index pairs making the data arrays fire the indexed values giving each pixel its colour. Note that the source is not a two-dimensional 6 * 4 array, but two one-dimensional arrays of 6 and 4 values, respectively. The image mapper now combines each of the first array's values with each of the second array's values, resulting in D = 6 * 4 = 24 combinations.

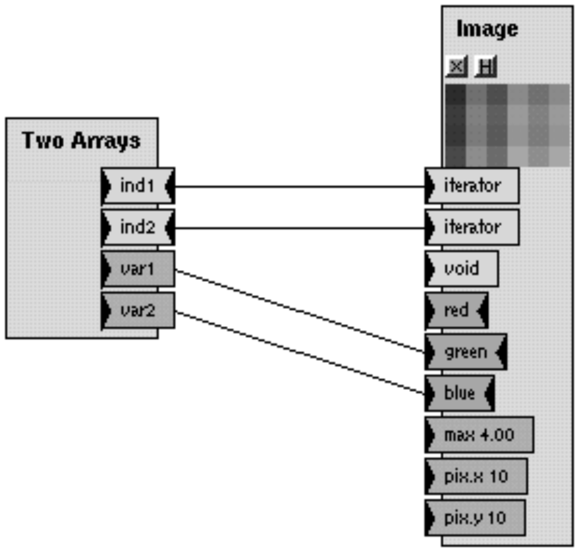


Figure 5: Mapping over two indices with different domains using an image mapper.

A data mapper has a generic architecture to guarantee a high degree of flexibility when connecting data units. Different data sources can be connected to the same mapper and so the iterators and dependent ports need to be dynamically generated as connections are made. Data mappers have one dummy port marked *"void"*. Connecting anything to a void port will make it change its name to "iterator" in the independent case or will take the name of the connected port if it is dependent. The void port is regenerated and stays free for further connections (Figure 6).

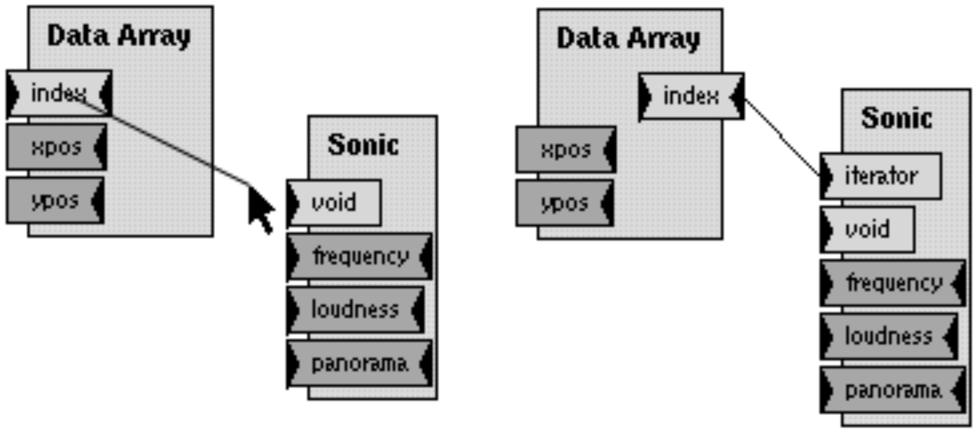


Figure 6: Connecting a *mapper*: dynamic generation of *iterator ports*.

4.4. Data Types

As data types integer, short, long, double, float, boolean, and string are supported. A data unit's ports hold meta-information on the data type used. Types are converted as the data flows through the program. If a port is not designed to convert to a certain data type (e.g. converting a string "anything" to an integer), a message box will appear on arrival of wrongly typed data, informing the user on what type is expected.

The set of data types is kept simple on purpose. No message script types like in MAX [Zica95] are implemented, because scripting components give the developer a certain "degree of freedom" which may let the complexity of the language get out of hand. The goal is to keep visual programming as visual as possible.

5. Applications

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, camera, lights, and time-dynamic media types such as animations and audio.

Summary and Outlook

By combining object-oriented framework technology with visual programming, we realized an environment that allows black-box reuse of an application framework’s classes. This takes reuse a step further towards rapid application development. Visual programs can potentially pass messages to any object of the framework once a wrapper is provided for it. Acting as a general media patcher, the visual editor will enable events to be collected, processed and distributed among the objects of the framework. In the future, the visual programming environment should become a distribution tool of events in the framework where anything can be processed and mapped to everything ("Media Patcher"). Visual programming can be used to define the behaviour of temporal dependencies, user interactions, as well as a general visualization tool for data sets. Further work is going on to wrap more and more of MET++'s classes and to allow the modularization of a dataflow-diagram by creating sub-diagrams made up of diagrams themselves.

Visual programs in executable documents may even be an alternative to scripting and program distribution which is now dominated by the hype about Java.

References

[Acke95] Ackermann, Ph., "Object-Oriented Synchronization of Audio-Visual Data in a Multimedia Application Framework", Dissertation at University of Zuerich, 1995.

[Gamma92] Erich Gamma: "Objektorientierte Software-Entwicklung am Beispiel von ET++" Springer Verlag; Berlin Heidelberg; 1992.

[Gamma95] Erich Gamma et al.: "Design Patterns", Addison-Wesley, Massachusetts, 1995.

[Helsel95] Robert Helsel: "Graphical programming: a tutorial for HP VEE"; Prentice Hall, 1995.

[NetCDF] is a special self-describing file-format for exchange of scientific data across different platforms. It is maintained by unidata. See http://www.unidata.ucar.edu/packages/netcdf

[Wein92] Weinand, A., "Objektorientierte Architektur für graphische Benutzungsoberflächen", Springer Verlag, Heidelberg, 1992.

[Zica95] Zicarelli, D., Puckette, M., "Getting Started with MAX", Opcode Systems Inc., Palo Alto, California, 1995.