75

Zeus: A System for Algorithm Animation and Multi-View Editing

Marc H. Brown

February 14, 1992

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in l984—their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

Abstract

Algorithm animation is a form of program visualization that is concerned with dynamic and interactive graphical displays of a program's fundamental operations. This paper describes the Zeus algorithm animation system. Zeus is noteworthy for its use of objects, strong-typing, and parallelism. Also of interest is how the system can be used for building multi-view editors.

Review by Jim Meehan

In many applications, it is at least as important to observe a program's progress as it is to obtain a final result, if indeed there is any. The Zeus system described in this paper provides support for watching and hearing a program in action, through several different views. The programmer animating an application provides a description of the application's fundamental operations, called "interesting events." Whenever an interesting event occurs, each view updates its visual or aural display appropriately. The Zeus system exploits Modula's object-inheritance, lightweight threads, and compile-time type-checking, so the programmer can take advantage of the predefined classes and methods to construct a sophisticated and efficient animation quickly and easily.

1 Introduction

Algorithm animation systems provide facilities for users to view and interact with an animated display of an algorithm, and for programmers to develop such animations. For *users*, the system provides ways to control the data given to algorithms, the ensemble of views, and the execution of algorithms. For *programmers*, the system provides support to make producing an animation of an algorithm almost as easy as producing a textual trace of it.

The common approach to animating algorithms specified in high-level procedural languages was pioneered in BALSA [4]. Briefly, the approach is as follows: An *algorithm* is annotated with markers that identify its fundamental operations that are to be displayed. These annotations, called *interesting events*, can have parameters that typically identify program data. Each *view* controls some screen real estate and is notified when an event happens in the algorithm. A view is responsible for updating its graphical display appropriately based on the event. Views can also propagate information from the user back to the algorithm.

This paper describes the Zeus algorithm animation system. We began developing the system in the summer of 1988, and the system has been stable and in use for the last three years. In addition to animating algorithms from the domains of computational geometry, operating systems, hardware design, distributed spanning trees, and communication protocols, Zeus is the conceptual framework for FormsEdit, a multi-view editor for building graphical user interfaces [1]. Zeus is noteworthy for its use of objects, strong-typing, and parallelism. Also, Zeus has allowed us to explore the use of color and sound, previously uncharted areas in algorithm animation [3]. A videotape of some algorithm animations that have been developed using Zeus is available [5].

All client code (as well as the system itself) is implemented in an in-house dialect of Modula-2, tailored for building large, integrated, object-based, concurrent programs. However, since that language is not distributed, we shall, for the sake of illustration, present the examples in Modula-3[7, 9].

The next section describes the facilities that Zeus offers to a user. Following that, we describe how a programmer views the system and we give an example of how an algorithm and a view are actually coded using Zeus. Next, we present the essentials of the system implementation. The final section describes how Zeus can be used for building multi-view editors.

2 The User's Perspective

When the user invokes a Zeus application, the control panel shown in Fig. 1 appears in a window on the screen. The control panel provides the user with configuration and interpretive facilities.

The *configuration* facilities let the user select which algorithm to run, which views to open, and which data to give to the selected algorithm. Each view will appear in its own window, which is installed into the workstation's window manager. The contents of the *Data* subwindow are specific to each algorithm, and Zeus provides many defaults for giving data. Other configuration facilities let the user write a snapshot of the state of the system to a file (e.g., the locations of view windows, data values for the selected algorithm), and restore the system from a previously created snapshot.

The *interpretive* facilities allow starting, stopping, and single-stepping an algorithm. The user can also control the speed of the animation. Zeus's "interpreter" is special-purpose and works in terms of the interesting events generated by the algorithm. For instance, the user's command to single-step causes Zeus to allow the algorithm to advance until the next event is generated.

By intention, Zeus's runtime facilities are minimal. The specific features we chose to implement are those we felt would be most important for our expected users, based on our experiences using BALSA and BALSA–II [2], where considerable effort was devoted to the user interface. For instance, had we expected that Zeus would be used in a classroom setting, as the BALSA systems were, then we would have implemented "scripts": high-level recordings of a user's session that can be replayed.

2.1 Utility Views

A novel feature of Zeus is that it can generate some utility views automatically based on the set of interesting events that the algorithm generates. Two of these views appear in the animation of Selection sort shown in Fig. 2.

The *Transcript* view contains a typescript that displays each interesting event as a symbolic-expression as it is generated. Actually, the editable part of the typescript contains a Lisp "read-eval-print" loop, with preloaded functions whose names are the events. When a function is invoked, the system behaves as if the algorithm or a view had caused the event to be generated.

A second view that Zeus provides automatically is the *Control Panel*. This view

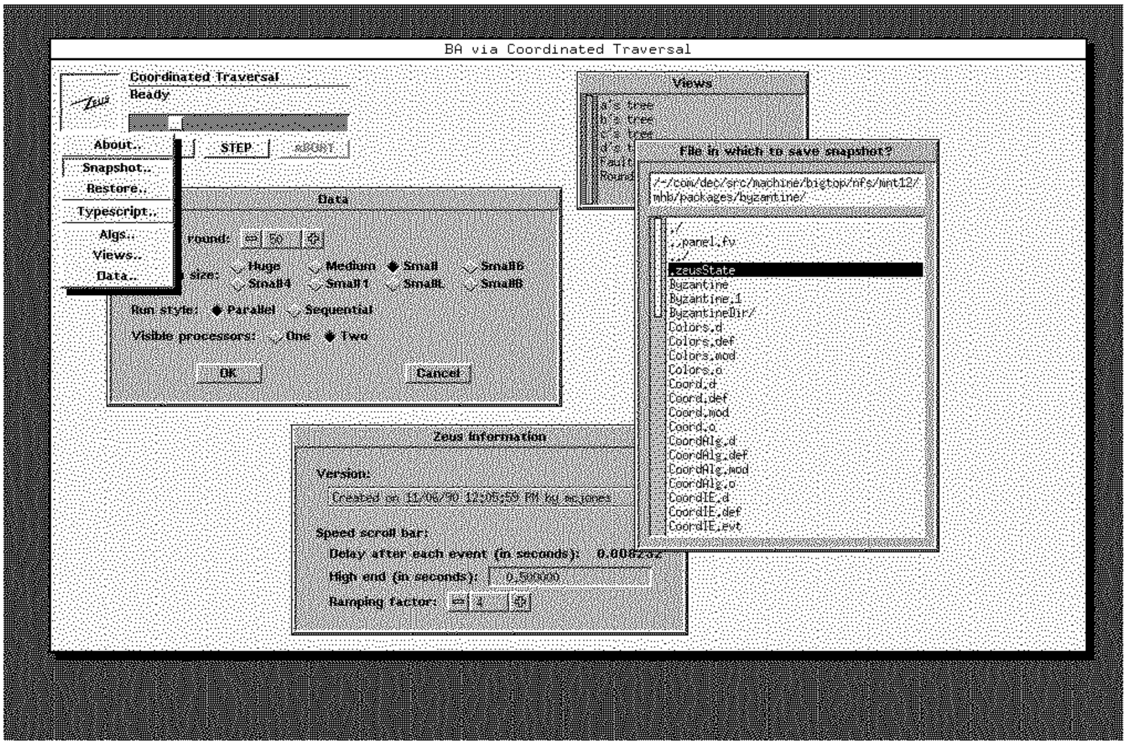


Figure 1: The Zeus control panel.

has buttons corresponding to each interesting event, with appropriate graphical widgets for specifying each parameter. Clicking on a button causes an event to be generated with the specified parameter values.

These views have proven to be extremely valuable for debugging both algorithms and views. Indeed, one can develop (and debug) a view before — and even without ever — implementing an algorithm!

3 The Programmer's Perspective

To a programmer, Zeus can be thought of as a domain-independent framework for associating multiple client-defined *views* with a set of client-defined *interesting events*, generated by a client program called the *algorithm*. Each view is an animated picture portraying the events as they are generated by the algorithm.

For example, the canonical view of a sorting algorithm (see Fig. 2) shows the elements being sorted as a row of sticks, where the height of each stick is proportional to the value of the corresponding element in the array. When the algorithm exchanges the values of two array elements, and it generates the event "exchange." In response to the exchange interesting event, the view changes the height of the sticks corresponding to the two array elements being swapped.

3.1 Interesting Events

Interesting events are specified as procedure signatures. Zeus's preprocessor, Zume, reads a file containing event specifications and generates definitions of algorithm and view classes (we'll use the terms "class" and "object" interchangeably). Zume also generates the utility views described in Section 2.1, and procedures for dispatching events between algorithms and views, as we shall see.

To a first approximation, a view object is a subclass of a window, with additional methods to process events generated by the algorithm. Similarly, an algorithm is a window subclass (typically never seen on the screen) with additional methods to process events that views may generate. Recall, that a view typically generates events in response to a user gesture. Actually, views and algorithms are both subclasses of a `Zeus` class—a window that has been subclassed to support algorithm animation and multi-view editing.

Here is the file of event specifications that many elementary sequential sorting algorithms use:

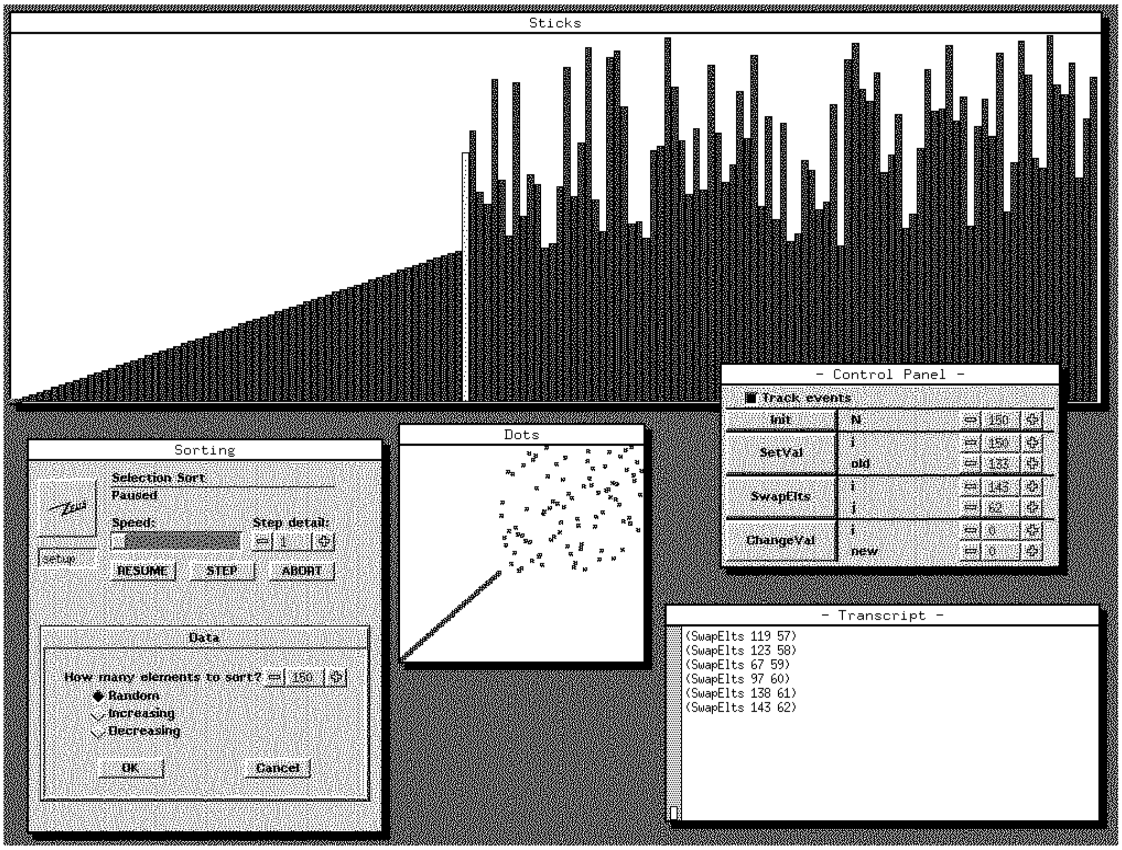


Figure 2: Animation of Selection sort.

```
EVENTS Sort;
ALGDATA
  a: ARRAY [1..100] OF Key;
  N: CARDINAL;
OUTPUT Init (N:CARDINAL);
OUTPUT SetVal (i:CARDINAL; old:Key);
OUTPUT SwapElts (i,j:CARDINAL);
FEEDBACK ChangeVal(i:CARDINAL; new:Key);
```

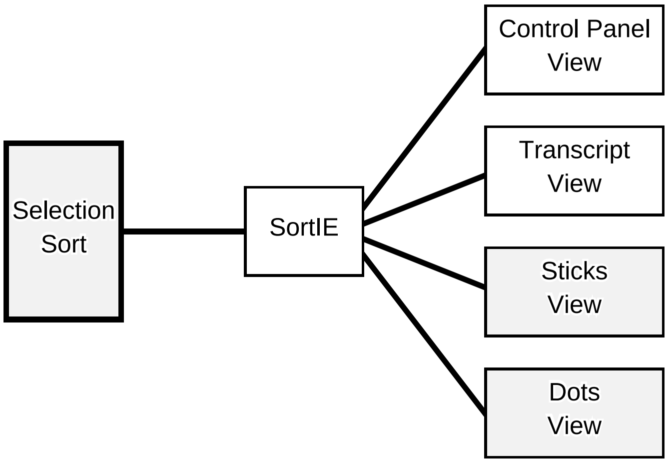
The name following the keyword `EVENTS` is used for naming the objects and files that Zume generates. The `ALGDATA` keyword specifies data fields for the sorting algorithms class that will be generated. The keyword `OUTPUT` indicates an event that will flow from the algorithm to all views, and `FEEDBACK` is used for events that flow from a view to an algorithm.

Here are the definitions, generated by Zume, of the procedures that dispatch these events between algorithms and views:

```
INTERFACE SortIE;
PROCEDURE Init (me:SortAlg.T; N:CARDINAL);
PROCEDURE SetVal (me:SortAlg.T; i:CARDINAL; old:Key);
PROCEDURE SwapElts (me:SortAlg.T; i,j:CARDINAL);
PROCEDURE ChangeVal (me:SortView.T; i:CARDINAL; new:Key);
```

The algorithm is annotated with calls to the first three routines, passing an identifier of itself as the first parameter to each. When one of these procedures is called, it invokes a method on each view that is designated to respond to the event. In a similar way, views may be annotated with calls to `ChangeVal`, typically in response to user gestures. The body of `ChangeVal` invokes the corresponding method on the algorithm. The implementation of these dispatching routines is discussed in Section 4.

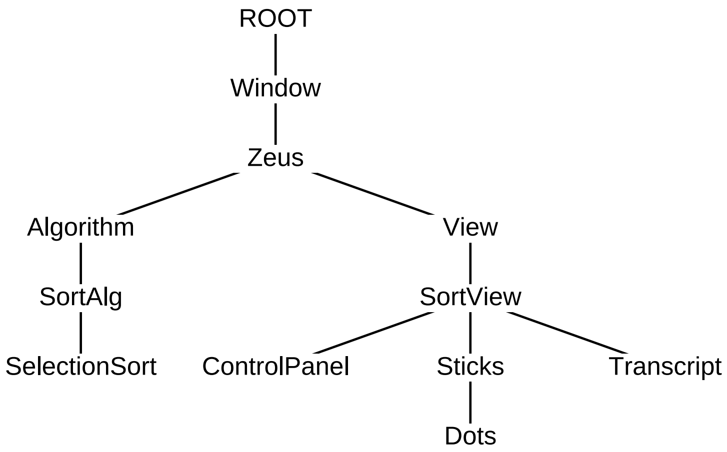
The flow of events between the algorithm and views appearing in Fig. 2 is as follows:



Output events flow from left to right; feedback events from right to left. Each box represents a module: those in white are generated by Zume based on the contents of the event specifications, whereas those in gray are implemented by a programmer animating an algorithm. We'll look at the implementation of Seletion sort in Section 3.3 and the Sticks view in Section 3.4. The remainder of this section describes the class hierarchy leading to the implementations of Seletion sort and the Sticks view.

3.2 Basic Classes

The class hierarchy for the views and algorithms in the animation of Selection sort appearing in Fig. 2 is as follows:



The `ROOT` object is part of the Modula-3 language; it is the basis of all objects. The `Window` object is a window that can be installed on a screen. The contents of windows are maintained by `Window` subclasses. The `Zeus` class is a window with methods that Zeus needs for multi-view event processing. The `Algorithm` and `View` subclasses are domain-independent; they are subclassed in domain-specific ways. The `SortAlg` and `SortView` subclasses are generated by Zume from the specifications of the sorting events we saw in Section 3.1. Subclasses of `SortAlg` are sorting algorithms that have been instrumented for animation; subclasses of `SortView` are the views that are meaningful for displaying sorting algorithms. Four such views appear in Fig. 2.

The definition of the `Zeus` class is as follows:

```
INTERFACE Zeus;
TYPE T = Window.T OBJECT METHODS
  init();
  dispose();
  startrun();
  endrun();
  configure(z: T; deleted: BOOLEAN);
  snapshot(Wr.T);
  restore(Rd.T);
END;
```

The `init` method is used to perform any initialization that must happen exactly once—before any other use of a `Zeus` object. The `dispose` method is called when the object is no longer needed. For instance, a view is no longer needed when the view window is deleted by the user from the window manager; an algorithm is no longer needed when the user changes which algorithm to run. The `dispose` method should release any resources it has. The `startrun` method is called whenever the user starts the algorithm; this is a handy way for views to reinitialize their displays. The `endrun` method is called when the algorithm finishes, possibly because the user explicitly aborted the execution using the "Abort" button. The `endrun` method makes it easy for an algorithm to clean up after itself, without concern for whether it terminated normally or prematurely (via the "Abort" button). The `configure` method is invoked whenever a view is deleted or added. Occasionally, it is useful for the algorithm or other views to know which views are currently on the screen and available to the user. Finally, the `snapshot` and `restore` method are used to implement the snapshot facility mentioned in Section 2.

The `Algorithm` class is a subclass of the `Zeus` class with two additional methods. The `run` is invoked when the user starts an algorithm by clicking on the "Go" button in the Zeus control panel. The `data` is invoked when the user clicks on the "Data" button in the control panel. Here is the definition:

```
INTERFACE Algorithm;
TYPE T = Zeus.T OBJECT METHODS
  run();
  data();
END;
```

The run method typically collects the data specified by the user and then starts generating events. It is what one typically thinks of as "the algorithm." The data method typically displays a dialog that lets the user specify data to be given to the algorithm. Examples of such dialogs appear in the *Data* subwindows in Figs. 1 and 2.

The methods that are inherited from the Zeus class have the following typical behaviors: The init method fills in the initial values of the dialog, and the dispose method releases all resources used by the data dialog or the algorithm. The snapshot and restore methods cause the values of interactors in the data dialog to be saved or restored. The startrun, endrun, and configure methods do nothing. Finally, keep in mind that although an algorithm is, by inheritance, also a window, it is typically not installed into the window manager.

The algorithm class SortAlg, generated by Zume from the event file, is a domain-specific subclass of Algorithm. It contains data fields specified as ALGDATA information in the event specifications, and methods to process each feedback event. Here is the definition:

```
INTERFACE SortAlg;
TYPE T = Algorithm.T OBJECT
  a: ARRAY[1..100] OF Key;
  N: CARDINAL;
METHODS
  feChangeVal(i:CARDINAL; new:Key);
END;
```

The feChangeVal method will be invoked when a view interprets a user's gesture to mean that the value of a key being sorted should change. The algorithm is not told which view is initiating the change, because an algorithm's response to a message that a key's value has changed is independent of the view in which the user gestured.

The implementation of SortAlg would provide default methods for all of the Algorithm methods. These methods would support a dialog with ways to enter a set of integers to be sorted.

The View class is simply a subclass of the Zeus class with no additional methods:

```
INTERFACE View;
TYPE T = Zeus.T OBJECT END;
```

Of the methods inherited from the `Zeus` class, only the `snapshot` and `restore` methods have interesting defaults. The default `snapshot` method records the screen location of its window. Views that allow user interaction to control viewing parameters, that is, information that is not given to the algorithm but is local to the view, must override the default snapshot method to also encode the current parameters set by the user. The `restore` procedure is the inverse of the snapshot procedure.

Although the `View` and `Zeus` objects may appear to be equivalent, they are not the same data types in Modula-3. The Zeus system exploits the fact that it can use language features to distinguish `View` from `Algorithm` subclasses of the `Zeus` class.

The view class `SortView`, generated by Zume from the event file, is a subclass of `View`, with additional methods to process each output event:

```
INTERFACE SortView;
TYPE T = View.T OBJECT METHODS
  oeInit (N: CARDINAL);
  oeSetVal (i: CARDINAL; old: Key);
  oeSwapElts (i, j: CARDINAL);
END;
```

These methods will be invoked as the algorithm runs and events are generated.

Thus, a view is essentially a window with two additional sets of procedures. One set is common to all Zeus views (i.e., the `Zeus` class), and the other set is common to all views of a particular algorithm (e.g., the `SortView` class).

3.3 Algorithms

Let's consider the elementary sorting algorithm Selection sort. It is a subclass of the `SortAlg.T` class we have examined, with the `run` and `feChangeVal` methods overridden:

```
MODULE SelectionSort;
TYPE T = SortAlg.T OBJECT OVERRIDES
  run := Run;
  feChangeVal := FEChange;
END;
```

The `run` method can be copied almost verbatim from any textbook:

```
PROCEDURE Run(self: T) =
  VAR min: INTEGER; t: Key;
  BEGIN
    GetData(self);
    WITH a=self.a, N=self.N DO
      FOR i := 1 TO N-1 DO
        min := i;
        FOR j := i+1 TO N DO
          IF a[j] < a[min] THEN min := j END
        END;
        t := a[min]; a[min] := a[i]; a[i] := t;
        SortIE.SwapElts(self, i, min);
      END;
    END;
  END Run;
```

The call to `SortIE.SwapElts` is what we referred to earlier as "annotation of an algorithm with 'interesting events.'" The call to `SortIE.SwapElts` will cause the `oeSwapElts` method on each view to be invoked in order to update the displays; the actual implementation of `SortIE.SwapElts` is generated by Zume.

The `feChangeVal` method is also instructive to examine. This method is invoked whenever a view interprets a user gesture to change the value of a key. This procedure changes the specified element, and then broadcasts to all views that a key's value has been changed:

```
PROCEDURE FEChange(self: T;
    view: SortView.T; i: CARDINAL; new: Key) =
  BEGIN
    WITH a=self.a, N=self.N, old=a[i] DO
      a[i] := new;
      SortIE.SetVal(self, i, old);
    END;
  END FEChange;
```

(In a multi-threaded environment such as Zeus, it is possible that `FEChangeVal` will be called while the `Run` method is executing. Zeus simplifies client code by following a reader/writer locking scheme: a view may only generate an event when it holds the write-lock, and the algorithm may not generate an event unless it has a read-lock. Note that this scheme allows a multi-threaded algorithm to generate events in parallel.)

Of course, changing the value of data elements while a program is underway may be a dicey proposition. It would certainly cause Selection sort to perform incorrectly! On the other hand, editing the underlying data from within views is the essence of multi-view editors, as we shall explore in Section 5.

3.4 Views

A difficult part of animating an algorithm is creating views. (Overall, the hardest—but most enjoyable—part is deciding what the view should look and sound like in order to convey interesting information!) Some systems, such as TANGO [10], provide a powerful two-dimensional graphics package. TANGO and other systems [6, 8] allow users to graphically demonstrate how views should look and behave.

Zeus does not have any sophisticated graphics packages or specially built graphical editors. However, Zeus does allow the algorithm animator to graphically demonstrate how an instance of an object used by a view should look, and does have some rudimentary library procedures to interpolate changes of object parameters over time. The editor for defining objects is the FormsVBT user-interface development environment [1]. Although FormsVBT was originally designed for "dialog boxes," it is general-purpose and completely (and very easily) extensible. Thus, one can quickly incorporate new widgets that are appropriate as building blocks for views. Fig. 3 shows a view from Fig. 4 being constructed using FormsVBT.

Another way that Zeus helps programmers create views stems from the fact that Zeus's views are true objects. First, the standard types of behavior like saving state and installing in the window system are provided by inheritance. Sophisticated views can customize this behavior, whereas simple views need not be concerned at all. The algorithm animation system does not dictate a long list of rules for how a view must behave, as do other systems. Second, it is easy to subclass and compose views. For example, in Fig. 4, the *Back-to-Back Stem (All)* view is composed of seven instances of the same view.

Finally, views can be programmed directly. For instance, the *Sticks* view in Fig. 2 is coded by subtyping `SortView` and using the `RectsVBT` window class to maintain a collection of rectangles. Here is the definition of the `SticksView` object that implements the the *Sticks* view:

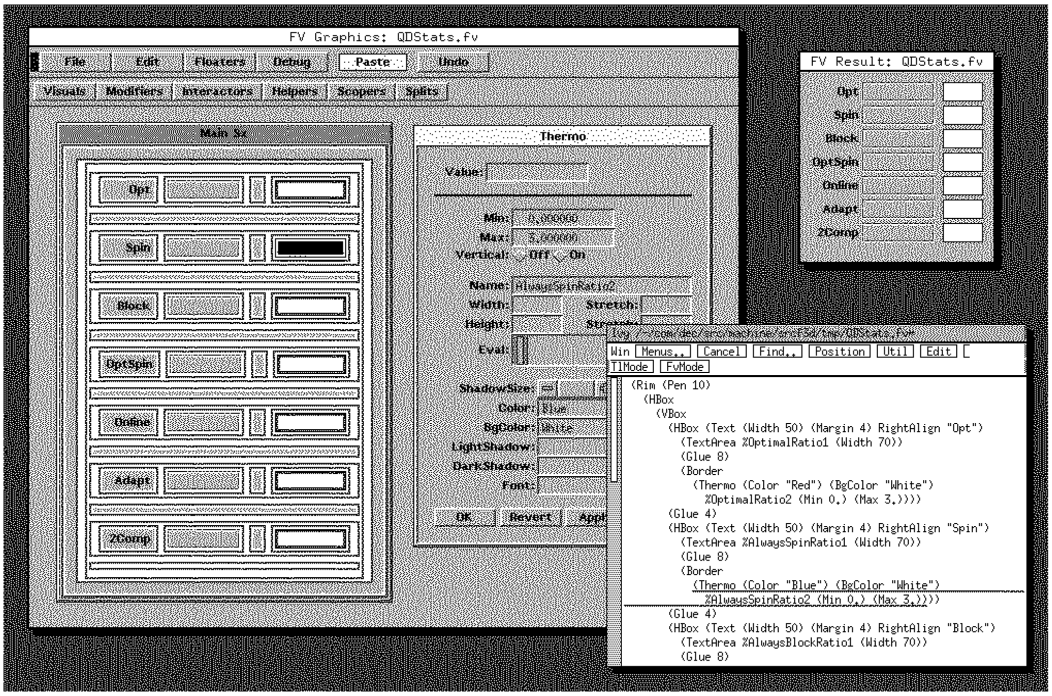


Figure 3: Constructing a view graphically using a multi-view editor.

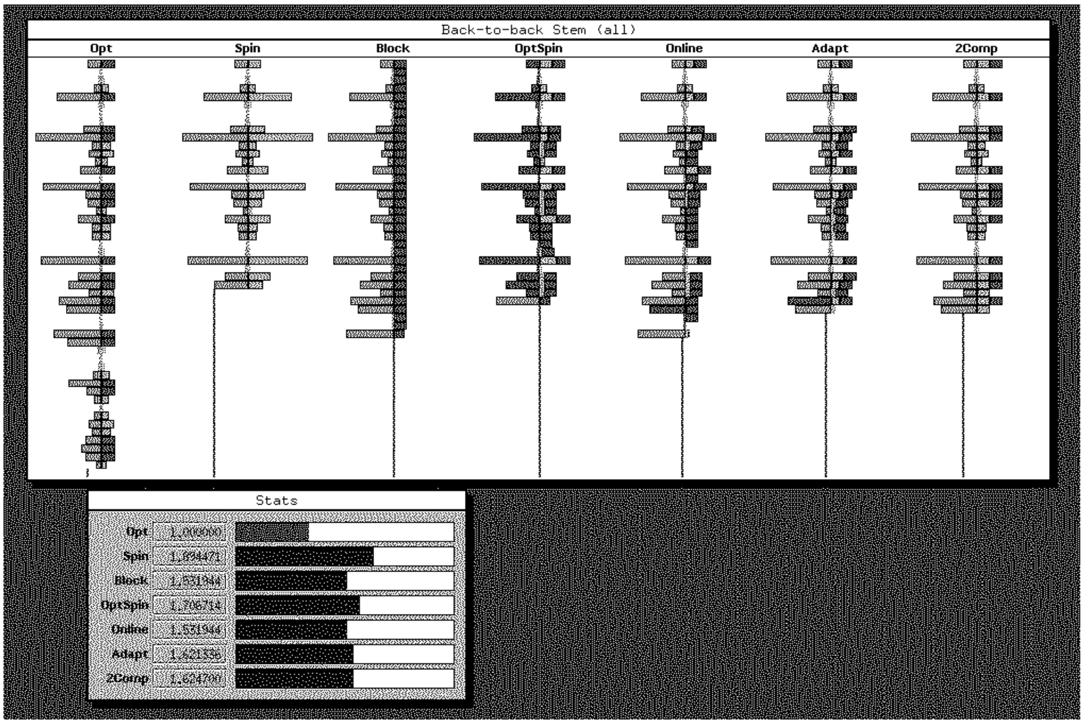


Figure 4: Multiple views of multiple algorithms.

```
TYPE SticksView = SortView.T OBJECT
  rects: RectsVBT.T;
OVERRIDES
  init := Init;
  oeInit := OEInit;
  oeSetVal := OESet;
  oeSwapElts := OESwap;
END;
```

The `Init` method creates a new `RectsVBT` object, stores a handle to it in the `SticksView` object, and installs it in the `SticksView` window. The three event processing methods are straightforward calls to entry points in the `RectsVBT` module. Here is one of the events:

```
PROCEDURE OESwap(self: T; i,j: CARDINAL) =
  BEGIN
    WITH a=Zeus.GetAlg(self).a DO
      RectsVBT.Set(self.rects, i, a[i], 0, i, i+1);
      RectsVBT.Set(self.rects, j, a[j], 0, j, j+1);
    END
  END OESwap;
```

The parameters to `RectsVBT.Set` are the handle to the window class, an unique identifier of the rectangle, and its north, south, east and west coordinates.

It is safe for a view's method to access a sequential algorithm's data fields because Zeus stops the current algorithm thread from running while an event is in progress. A multi-threaded algorithm might have other threads modifying its data fields while an event in one thread is in progress, so views must be careful to acquire an appropriate lock from the algorithm before accessing the algorithm's data. Another complication that arises in a multi-threaded window system (regardless of whether or not the algorithm itself is multi-threaded) concerns repaint requests issued by the window manager. The view, as a subclass of a window, must handle repaint requests whenever issued. The view's repaint method must be careful either to not use the algorithm's data (since the algorithm may be running concurrently), or coordinate a locking scheme with the algorithm.

4 System Implementation

The Zeus system comprises the control panel, event-dispatching, and the default methods for algorithm and view classes. We have already seen the gist of the default algorithm and view classes.

The implementation of the configuration aspects of the control panel is straightforward. Most of the commands (e.g., Snapshot) just invoke the appropriate Zeus method (e.g., the `snapshot` method) on the algorithm and current set of views.

The implementation of the control panel's interpretive commands is tricky, primarily because user commands happen asynchronously while the algorithm is running. The "Go" button (hidden by the pull-down menu in Fig. 1) causes the algorithm's `run` method to be invoked in a separate thread. This thread is terminated when the user invokes the "Abort" button. The "Step" command is implemented by setting the Zeus variable `stepFlag` to be true; this variable will be checked by the event-dispatching code. The "Step" command also awakens the algorithm thread, in case it is currently stopped and must be advanced. Finally, whenever the program is stopped, the "Go" button is replaced by a "Resume" button. The "Resume" button is implemented by awakening the algorithm thread, but without setting the `stepFlag`.

Zeus event-dispatching is implemented by the bodies of the event procedures generated by Zume. In the case of an event sent from the algorithm to the views, the event forks a thread for each view, and the thread invokes the appropriate method. After all views have completed, the message dispatcher returns to the algorithm—unless the `stepFlag` has been set. In that case, the flag is cleared, and the algorithm sleeps until awakened as the result of the user issuing a "Resume" or "Step" command.

Here's pseudo-code of the SwapElts implemention:

```
PROCEDURE SwapElts(alg:SortAlg.T; i,j:CARDINAL)=
  BEGIN
    FOREACH view IN Zeus.GetViewList(alg) DO
      FORK view.oeSwapElts(i, j);
    END;
    wait for all threads to join
    IF Zeus.stepFlag THEN
      Zeus.stepFlag := FALSE;
      sleep until awakened
    END;
  END SwapElts;
```

The actual code is slightly more complicated for two reasons: First, because user-events can happen concurrently with event-dispatching, access to the list of views and the step flag must be protected by a lock. Second, if any of the forked view methods raises an exception, this must be caught and reported back to the caller.

Unlike events in other event-based algorithm animation systems (notably BALSA and TANGO), events in Zeus are strongly typed. This makes it is impossible for an algorithm to invoke an event with the wrong number or types of parameters; likewise, it is impossible for a view to respond to an event without retrieving the correct number and types of parameters. A discussion of the benefits and costs of strong type-checking is beyond the scope of this paper, but after experiencing both types of systems, we are strong proponents of strong type-checking.

4.1 Zume Preprocessor

The Zume preprocessor plays an important role in the Zeus system: it generates class definitions, bodies of the event procedures, and various utility views.

It has been important that Zume be flexible in what it can generate. We achieve flexibility by driving Zume from the file of event definitions *and* various template files. A template file is expanded using the event procedure signatures. For example, the actual template for an event is as follows:

```
MODULE #(_ALGNAME_)IE;
#{
PROCEDURE #(_EVENT_)(
    alg: #(_ALGNAME_)Alg.T;
    #[#(_ARGNAME_): #(_ARGTYPE_); #])=
BEGIN
  FOREACH view IN Zeus.GetViewList(alg) DO
    FORK view.oe#(_EVENT_)(#[#(_ARGNAME_),#]);
  END;
  wait for all threads to join
  IF Zeus.stepFlag THEN
    Zeus.stepFlag := FALSE;
    sleep until awakened
  END;
END #(_EVENT_);
#}
END #(_ALGNAME_)IE.
```

Zume used this template, along with the file of sorting events, to generate the body for `SwapElts` we saw above.

The initial version of Zume was written using Unix's awk, sed, and trans (an in-house, awk-like filter). The shell script was about 80 lines long and consisted of about a dozen calls each to sed, awk, and trans.

Unfortunately, text manipulation of template files alone is not rich enough to generate the *Control Panel* view, because that view must know the base type of each parameter in order to use a type-specific widget for displaying the parameter's value. Zume was subsequently reimplemented in Modula and linked with the type system of our existing compiler tools.

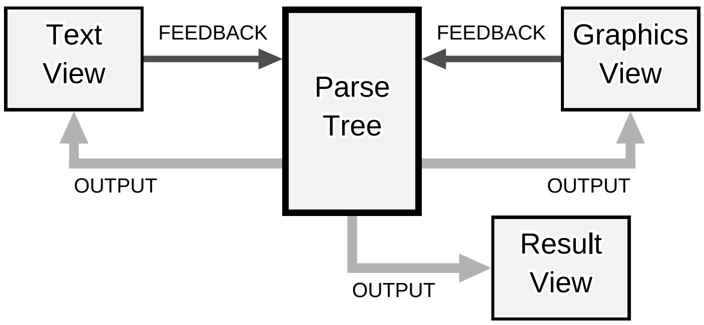
5 Multi-View Editing

Zeus can be used for building multi-view editors. In a multi-view editing system, the "algorithm" maintains the data structures that are shared among all "views" (i.e., the editors). Each view interprets user gestures and initiates feedback events to the algorithm; the algorithm updates the common data structures and sends output events to all views. Each view, including the view in which the user initiated the editing action, updates itself in response to the output events. Although the algorithm's `run` method is never invoked, it is still important to maintain a distinction between an “algorithm” and a “view” to ensure the proper directional flow of events.

Based on this framework, we implemented FormsEdit, a multi-view editor for creating user interfaces in the FormsVBT system (see Fig. 3). There are two editable views: The *Graphics* view on the left is a direct-manipulation graphical editor, and the *Text* view at the lower right uses a conventional text editor to display and edit the s-expression underlying the user interface in the graphics view. Changes in one editor are reflected simultaneously in the other editor. The *Result* view in the upper right shows the user interface as it will look at runtime, with proper reaction to mouse and keyboard activity, as well as proper sizing and stretching. The result view is updated as the user edits either the graphics or text view. Editing the result view does not change the underlying s-expression.

FormsEdit is organized around one central data structure, a parse tree. This tree represents an s-expression, having one node for each component in the s-expression. The “algorithm” (i.e., the ParseTree module) maintains the parse tree and communicates all tree changes to the views as output events. A change request, arising from user action in either of the two editable views, is issued by a feedback event from the editor to the parse tree module. This module makes the change to the tree itself, then generates an output event. In parallel, each view updates its local data structures and redisplays itself appropriately.

The following block diagram shows how the information flows between the modules in FormsEdit:



It is important to realize that the modules do not actually call each other as the arrows in the diagram above suggest. Rather, modules are annotated with events, and the body of the event routine (generated by hand, not by the Zume preprocessor, for historical reasons) invokes a method on each editor (for output events) or on the parse tree module (for feedback events).

Because the modules generate events rather than calling other modules directly, new editors, or multiple instances of the same editor, can be added without changing any of the existing editors or the algorithm module. In FormsEdit, for example, it might be convenient to run with multiple instances of the *Result* view.

Finally, it is important for the view initiating the editing action to report error conditions, even though it may be the algorithm or another view that detects the error. This is handled by appropriate bookkeeping in the event dispatching procedure.

6 Conclusion

Systems for algorithm animation have matured significantly in the last decade. Zeus contributes to this evolutionary path a practical system whose design and implementation are quite simple. Simplicity is achieved primarily by exploiting modern programming technologies, such as objects, type-checking, and threads.

Constructing animations in Zeus appears to be as easy and straightforward as in any other algorithm animation system. Objects make it easy to reuse views, and to build sophisticated views by composing and subclassing other views. The graphical editor helps to construct new views. Although it contains no support for specification of incremental transformations, we haven't felt hindered by this in practice. Zeus events are strongly typed, thereby eliminating a large class of common programming errors. Typed events allow the automatic creation of event-generating views.

Inspired by how well-suited Zeus has turned out to be for building a multi-view editor application, we look forward to discovering even more uses for algorithm animation systems.

Acknowledgments

Ken Brooks and John Hershberger implemented the Zume preprocessor. John and Lyle Ramshaw improved the clarity of this paper.

References

[1] Gideon Avrahami, Kenneth P. Brooks, and Marc H. Brown. A two-view approach to constructing user interfaces. *Computer Graphics*, 23(3):137–146, July 1989.

[2] Marc H. Brown. Exploring algorithms using BALSA–II. *IEEE Computer*, 21(5):14–36, May 1988.

[3] Marc H. Brown and John Hershberger. Color and sound in algorithm animation. In *Proc. IEEE 1991 Workshop on Visual Languages*, pages 10–17, October 1991. An expanded version of this paper is available as Research Report 76a from DEC Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301.

[4] Marc H. Brown and Robert Sedgewick. A system for algorithm animation. *Computer Graphics*, 18(3):177–186, 1984.

[5] Marc H. Brown, ed. An anthology of algorithm animations using Zeus. Research Report Videotape 76b, DEC Systems Research Center, 130 Lytton Ave., Palo Alto, CA, September 1991. A segment entitled "An Introduction to Zeus: Audio Visualization of Some Elementary Sequential and Parallel Sorting Algorithms" is part of the CHI '92 video program.

[6] Robert A. Duisberg. Visual programming of program visualizations. In *Proc. Conf. on Visual Languages*, August 1987.

[7] Sam Harbison. *Modula–3*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[8] Esa Helttula, Aulikki Hyrskykari, and Kari-Jouko Räihä. Graphical specification of algorithm animations with Aladdin. In *Proc. of the 22nd Hawaii Int'l. Conf. on System Sciences*, pages 892–901, January 1989.

[9] Greg Nelson, ed. *Systems Programming with Modula–3*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[10] John T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, September 1990.