Building Algorithm Animations with Zeus *(Draft)*

Marc Najork[*]

February 26, 1993

1 Introduction

Zeus [1] is a system for animating algorithms which was developed by Marc H. Brown and others at Digital Equipment's System Research Center.

Zeus allows its user to run an *algorithm* and to observe it through several *views*. Algorithms and views communicate by passing *events* back and forth. For this course, we only care about events passed from the algorithm to the view. Such events are called *output events*.

A view is a visual representation that (hopefully) shows some of the internal state of the algorithm. Users can provide (where "providing" means "programming") as many views as they want. Most of the views are graphical: they might show trees, lists, or arrays internal to the algorithm, and how they are modified[1]. But a view can also be textual, and in fact, there is a textual standard view, called the *Transcript View*, that shows all the events passed between algorithm and views. There are standardized ways to construct two other textual views: a *Code View*, which shows the program code and highlights the command that is just being executed, and a *Data View*, which shows the value of variables in a textual form. There are no "standard" graphical views, but there is a rich set of graphics and animation libraries that makes it easy to construct such views.

One of the design goals of Zeus is to make it easy to build new animations. This is achieved in several ways:

- Algorithms and views are separated, and can to a certain extent be tested separately
- Events are described through a high-level Event Description Language, and some of the code for algorithms and views is generated automatically from this event description.
- Zeus is written in Modula–3 [8, 5, 4], an object-oriented language, and subclassing is used extensively to reuse and interface with existing code.

In the following, we will go through the moves required to build a simple animation for the Insertion Sort algorithm. Building any animation involves three basic steps:

[*]Author's address: Department of Computer Science, University of Illinois, 1304 West Springfield Avenue, Urbana, IL 61801, e-mail: najork@cs.uiuc.edu

[1]Some of the "views" experimented with at DEC SRC are aural: sounds that indicate interesting events during the execution of the algorithm.

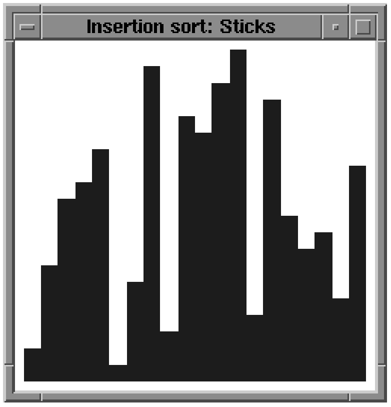


Fig. 1: The Sticks View of Sorting

1. Decide on what the interesting events should be, and write an Event Specification.
2. Write the algorithm you want to animate, and instrument it at the right places with procedure calls that generate the right events.
3. Write a view that receives the events generated by the algorithm, and shows them in some way.

Steps 1 and 2 are usually quite straightforward (especially if you take a textbook algorithm). Step 3 is what adds the spice to life: you have to decide on what a nice animation should look like, and figure out how to bring it to the screen.

2 The Event Specification File

Suppose we want to animate the Insertion Sort algorithm, and we want to show the array to be sorted as a row of sticks, such that the height of each stick indicates the value of the array element (the higher the larger). Fig. 1 shows the view we are looking for.

We see that there are really two kinds of interesting events: an event that sets up the animation (let's call it `Init`) and an event that sets the value of a particular array element, i.e. the height of a particular stick (let's call it `SetVal`).

Here is our first cut at an event specification:

```
OUTPUT Init (N : INTEGER [Fmt.Int]);
OUTPUT SetVal (i : INTEGER [Fmt.Int]; val : INTEGER [Fmt.Int]);
```

Event specifications are contained in files, which have to end with the suffix "`.evt`". Let's assume this specification is in a file called `Sort.evt`. This file will be read by a preprocessor called `m3zume`, which generates Modula–3 files from it. In the following, I will refer to the Event Specification file as the *Zume* file.

`OUTPUT` is a Zume keyword which indicates that the event we are describing is an output event, an event passed from the algorithm to the views attached to it. `Init` is the name of this output event. The event should clear the view, and set it up for a row of `N` sticks. `N` is of type `INTEGER`. `SetVal` is supposed to set array element `i` to value `val`.

`m3zume` generates (among other things) the code for the Transcript View mentioned above. The Transcript View gives us a textual log of all the events that are passed around. Hence we must supply a function that converts the integer `N` into text. Luckily, `Fmt`, one of Modula–3's standard modules, already contains such a function, namely `Int`. In general, if you specify an output event to be

```
OUTPUT EventName (x :  T [Foo.TtoText]; ··· );
```

then the function *TtoText*, defined in the module *Foo*, should convert a value of type T to a value of type `TEXT`.

This is the time for a little digression. As the name suggests, Modula–3 features the module-concept. A *module* consists of an *interface* part and an *implementation* part[2]. Procedures defined in the implementation part are *exported* to the outside world if their procedure header appears in the interface part. An interface can also define types, constants, variables, etc., which are then available to outside clients as well. Interfaces are not automatically visible to a client (i.e. another interface or module), they have to be explicitly *imported*.

Zume files also have to import the interfaces they rely on. The only exception here is the `Fmt` interface, which is automatically imported, because almost every Zume file will use it. But suppose you wanted to describe an event that sets up a graph structure in a view. You have already built a module `Graph`, which exports `T`, the type of graphs, and `ToText`, a function that converts graphs to text. Your Zume file would look something like this:

```
IMPORT Graph;

OUTPUT SetUpGraph (g : Graph.T [Graph.ToText]);
...
```

The DEC SRC implementation of Modula–3 also provides a tool called `m3make`, which is a simplification of the Unix `make` mechanism. An `m3makefile` contains only the names of the modules etc. one wants to build, but not the files they depend on. This information is inferred automatically. Another convention is that you put *your* files into a subdirectory called `src`, and create a second subdirectory called `SPARC` if you work on a Sun SparcStation, or IBMR2 if you work on an IBM RS/6000. Let's assume your base directory for this sorting animation is called `sorting`. By now, you would have the following files:

```
sorting/
sorting/src/
sorting/src/Sort.evt
sorting/src/m3makefile
sorting/SPARC
```

For now, our `m3makefile` for the sorting example contains just one line, namely the name (or, more precisely, the prefix) of the Zume file `Sort.evt`:

```
zume (Sort)
```

When you run `m3make` *from the base directory (!)*, all the temporary files (files created by `m3zume`, object files, executables, etc.) will go to `sorting/SPARC`. This makes it easy to keep track of files. Let's try it out:

[2]For a module `Foo`, the interface must be in a file `Foo.i3`, and the implementation must be in a file `Foo.m3`.

```
$ m3make
============================ Building in SPARC
/delta/std/m3/bin/m3zume ../src/Sort
m3zume processing file ../src/Sort.evt...
m3zume creating SortAlgClass.i3...
m3zume creating SortAlgClass.m3...
m3zume creating SortViewClass.i3...
m3zume creating SortViewClass.m3...
m3zume creating SortIE.i3...
m3zume creating SortIE.m3...
m3zume creating SortTranscriptView.i3...
m3zume creating SortTranscriptView.m3...
m3zume creating SortTranscriptView.fv...
m3zume creating SortEventData.fv...
m3zume finished.

$ ls -R
SPARC   src

SPARC:
SortAlgClass.i3        SortIE.m3               SortViewClass.i3
SortAlgClass.m3        SortTranscriptView.fv   SortViewClass.m3
SortEventData.fv       SortTranscriptView.i3
SortIE.i3              SortTranscriptView.m3

src:
Sort.evt       m3makefile
```

3 Instrumenting an Algorithm

Next, we have to supply the algorithm we want to animate, and instrument it with procedures that generate output events.

The `m3zume` preprocessor has generated a module `SortAlgClass` from the `Sort.evt` file. The interface of this module (i.e. the file `sorting/SPARC/SortAlgClass.i3`) exports a type `T`, the type of sorting algorithms. We want to create a subtype (or subclass[3]) of this type, which shall be the class of insertion sort algorithms. We thus create a new file, let's say `sorting/src/SortAlg.i3` (in the following, I will leave out the `sorting/src/` and `sorting/SPARC/` prefixes, with the understanding that all the files *we* create are in `sorting/src/`).

```
MODULE SortAlg;

IMPORT SortAlgClass;

TYPE
  InsertionSort = SortAlgClass.T BRANDED OBJECT
    a : ARRAY [0 .. 101] OF INTEGER;
    N : INTEGER;
  OVERRIDES
    run := Run;
  END;
```

[3]A "class" is called "object type" in Modula–3 terminology

```
BEGIN
END SortAlg.
```

This describes the implementation part of a module `SortAlg`, and defines the type `InsertionSort` to be a subtype of `SortAlgClass.T`. `InsertionSort` adds two new fields to `SortAlgClass.T`, namely `a`, the array we want to sort, and `N`, the used size of the array. It also *overrides* one of the methods defined in a superclass, `run`, with a new procedure, `Run`. Consequently, we have to define this procedure next.

```
PROCEDURE Run (self : InsertionSort) RAISES {Thread.Alerted} =
  VAR
    j, v : INTEGER;
  BEGIN
    GetData (self);
    WITH a = self.a, N = self.N DO
      FOR i := 2 TO N DO
        v := a[i];
        j := i;
        WHILE a[j - 1] > v DO
          a[j] := a[j - 1];
          SortIE.SetVal (self, j, a[j]);
          DEC (j);
        END;
        a[j] := v;
        SortIE.SetVal (self, j, v);
      END
    END
  END Run;
```

`Run` takes `self`, an `InsertionSort` object, and might raise an exception called `Thread.Alerted`. As a matter of fact, Modula–3 supports multi-threading (i.e. parallelism), and Zeus uses this feature: algorithms, views, and the central controller execute in independent threads. If you press the "Abort" button, the central controller will notify the algorithm thread of this, the algorithm thread will raise an exception `Thread.Alerted`, which will, thanks to the `RAISES {Thread.Alerted}` declaration, be passed on to the caller of `Run`, who will handle it in some appropriate way.

Why does `Run` need to have the signature it has? Well, `InsertionSort` is a subtype of `SortAlgClass.T`, which in turn is a subtype of `Algorithm.T`. The `Algorithm` interface looks like this (well, almost ...):

```
INTERFACE Algorithm;

IMPORT Thread, ZeusClass;

TYPE
  T <: Public;
  Public = ZeusClass.T OBJECT
             ...
           METHODS
             ...
             run () RAISES {Thread.Alerted};
           END;

END Algorithm.
```

We see that the `run` *method* takes no arguments. The *procedure* associated with a method belonging to an object type T always has one extra parameter (the first one), which must be of type T. So, the `Run` procedure associated with the `run` method of `Algorithm.T` takes an `Algorithm.T`, the `Run` procedure associated with the `run` method of `InsertionSort` takes an `InsertionSort`.

The body of `Run` (the one above) is a straightforward implementation of insertion sort. The only two interesting points are the calls to `SortIE.SetVal`. `SortIE` (which stands for "Sort-Interesting-Event") is one of the modules that have been automatically generated by `m3zume` from the `Sort.evt` file. A call to `SortIE.SetVal` will pass the `SetVal` output event on to all views attached to the algorithm `self`.

Recall that we declared the `SortVal` output event as

```
OUTPUT SetVal (i : INTEGER [Fmt.Int]; val : INTEGER [Fmt.Int]);
```

Based on this, `m3zume` created the interface `SortIE.i3` which contains the procedure declaration

```
PROCEDURE SetVal (initiator: Algorithm.T; i: INTEGER; val: INTEGER) RAISES {Thread.Alerted};
```

When `Run` calls `SortIE.SetVal`, it passes `self`, which is an `InsertionSort`, to `initiator`, which is an `Algorithm.T`, i.e. a supertype of `InsertionSort`.

Of course, the insertion sort algorithm needs some data to start out with. For now, we will just set `self.N` to 20 and fill `self.a` with a random permutation of the numbers from 1 to 20. This is done in `GetData`:

```
PROCEDURE GetData (self : InsertionSort) RAISES {Thread.Alerted} =
  VAR
    t : INTEGER;
  BEGIN
    WITH a = self.a, N = self.N DO
      N := 20;
      FOR i := 1 TO N DO
        self.a[i] := i;
      END;
      FOR i := 1 TO N DO
        WITH j = Random.Subrange (Random.Default, 1, i) DO
          t := a[i];
          a[i] := a[j];
          a[j] := t;
        END;
      END;
      SortIE.Init (self, N);
      FOR i := 1 TO N DO
        SortIE.SetVal (self, i, a[i]);
      END;
    END;
  END GetData;
```

`GetData` also sends the `Init` output event to all the attached views (cleaning them up), and sets up the initial array representation by sending a `SetVal` event for each array element.

There is one more thing to do: we have to register the insertion sort algorithm with the central controller. This is done by adding one line to the main body of the `SortAlg` implementation:

```
BEGIN
  ZeusPanel.RegisterAlg (New, "Insertion sort", "Sort");
END SortAlg;
```

`Sort` is again the prefix of the Zume file `Sort.evt`. "Insertion sort" is an arbitrary string, that will be used in the algorithm selection panel of the central controller. `New` is a procedure that returns a new `InsertionSort`. We have to define this procedure:

```
PROCEDURE New (): Algorithm.T =
  BEGIN
    RETURN NEW (InsertionSort).init();
  END New;
```

That leaves us with a consistent version of `SortAlg.m3`:

```
MODULE SortAlg;

IMPORT SortAlgClass, SortIE;                   (* derived from Sort.evt *)
IMPORT Algorithm, Random, Thread, ZeusPanel;   (* predefined modules *)

TYPE
  InsertionSort = SortAlgClass.T BRANDED OBJECT
    a : ARRAY [0 .. 101] OF INTEGER;
    N : INTEGER;
  OVERRIDES
    run := Run;
  END;

PROCEDURE GetData (self : InsertionSort) RAISES {Thread.Alerted} =
  VAR
    t : INTEGER;
  BEGIN
    WITH a = self.a, N = self.N DO
      N := 20;
      FOR i := 1 TO N DO
        self.a[i] := i;
      END;
      FOR i := 1 TO N DO
        WITH j = Random.Subrange (Random.Default, 1, i) DO
          t := a[i];
          a[i] := a[j];
          a[j] := t;
        END;
      END;
      SortIE.Init (self, N);
      FOR i := 1 TO N DO
        SortIE.SetVal (self, i, a[i]);
      END;
    END;
  END GetData;

PROCEDURE Run (self : InsertionSort) RAISES {Thread.Alerted} =
  VAR
    j, v : INTEGER;
  BEGIN
    GetData (self);
    WITH a = self.a, N = self.N DO
      FOR i := 2 TO N DO
```

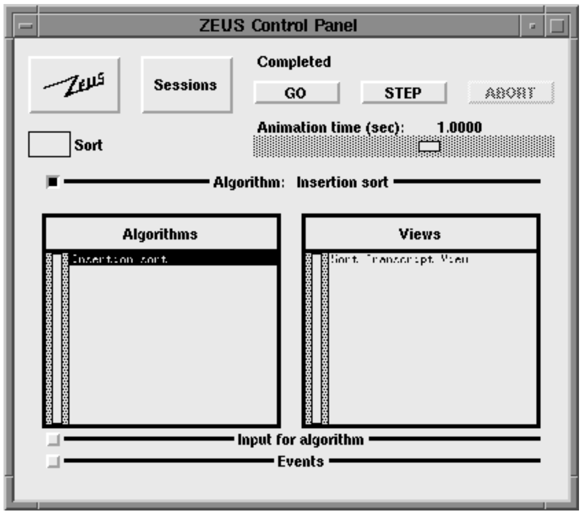
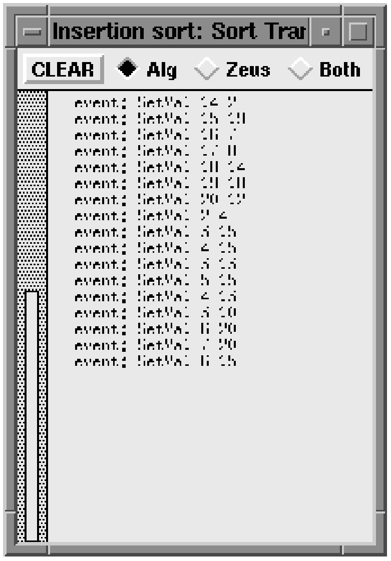


Fig. 2: The Central Control Panel



```
event: SetVal 14 9
event: SetVal 15 19
event: SetVal 16 7
event: SetVal 17 11
event: SetVal 18 14
event: SetVal 19 18
event: SetVal 20 19
event: SetVal 9 4
event: SetVal 3 15
event: SetVal 4 15
event: SetVal 3 13
event: SetVal 5 15
event: SetVal 4 13
event: SetVal 3 10
event: SetVal 6 20
event: SetVal 7 20
event: SetVal 6 15
```

Fig. 3: The Transcript View

```
        v := a[i];
        j := i;
        WHILE a[j - 1] > v DO
          a[j] := a[j - 1];
          SortIE.SetVal (self, j, a[j]);
          DEC (j);
        END;
        a[j] := v;
        SortIE.SetVal (self, j, v);
      END
    END
  END Run;

PROCEDURE New (): Algorithm.T =
  BEGIN
    RETURN NEW (InsertionSort).init();
  END New;

BEGIN
  ZeusPanel.RegisterAlg (New, "Insertion sort", "Sort");
END SortAlg.
```

We also have to supply an interface file `SortAlg.i3`:

```
INTERFACE SortAlg;
END SortAlg.
```

For a Modula–3 program, there is always one implementation part that contains the main program. The body of the main program gets executed after all the bodies of the modules have been executed. Here is the file `Main.m3`:

```
MODULE Main;

IMPORT Rsrc, SortBundle, ZeusPanel;

BEGIN
  ZeusPanel.Interact (path := Rsrc.BuildPath(SortBundle.Get()));
END Main.
```

`ZeusPanel.Interact` pops up the central control panel and starts the user interaction. `SortBundle` is the name of a module generated by `m3make` which contains *resources*, among them the description of the central control panel.

We also have to update our `m3makefile`:

```
import_obj ($(ZEUSLIB))
bundle (SortBundle)
zume (Sort)
module (SortAlg)
implementation (Main)
program (Sort)
```

The `program` directive specifies a name for the executable. The `import_obj` directive tells `m3make` where to look for object files when linking the entire program. `implementation` defines an implementation, `interface` defines an interface, `module` defines both.

The `bundle` directive specifies a name for the module into which all *resources* are collected. We did not supply any resources so far, but there are some resources behind the scene, for instance the central control panel, so the directive is needed. Note that the name of the bundle is the same as the name used in `Main.m3`.

If we execute `m3make` at this point, it will construct a program `Sort` (which is located in the `SPARC` directory). Running `Sort` will pop up the central control panel (Fig. 2). If we click onto the "Transcript View" line in the view selection panel, a Transcript View pops up. Pressing the "Go" button starts the algorithm, and generated output events are shown in the Transcript View (Fig. 3).

4 Building a Control Panel

So far, we have "hard-wired" the number of elements in the array into the `GetData` procedure. It would be nice if we could change this number without having to recompile the entire application. In fact, we *really* would like to have a graphical "widget" on the control panel that allows us to adjust this number. Fortunately, the libraries and tools coming along with Modula–3 make this really easy.

First, we need to specify the way our input control panel should look like. Let's assume we want to build a panel that looks like the one shown in Fig. 4. The SRC Modula–3 environment includes a "widget system" called `FormsVBT`, which uses a lisp-like language to describe control panels. Along with the language comes `formsedit`, a "multi-view" editor that allows you to type in the lisp-description of a form in one window and see the resulting form in another window. `formsedit` also provides a short on-line manual that covers the FormsVBT language in more detail than we can provide here. The definitive guide to FormsVBT is [2].

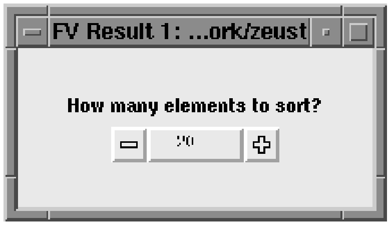


Fig. 4: The Input Control Panel

The underlying metaphor of the FormsVBT language is based on TEX: "hboxes" (horizontal boxes) are used to arrange elements horizontally, "vboxes" (vertical boxes) are used to arrange elements vertically. "Glue" provides horizontal or vertical spacing of fixed size, "fill" acts as a glue of infinite stretchability.

There are various interactors: buttons, radio buttons, scrollers, numeric and text input devices, menus, pop-up subwindows, etc. For our example, we only need two types of interactors: a text interactor that allows us to show a text constant, and a numeric input device.

Here is the grammar of the fragment of FormsVBT we are using:

e	::=	`(Rim (Pen` n`)` e`)`	creates a space of width n around e
	|	`(HBox` $e_1 \cdots e_n$`)`	shows $e_1 \cdots e_n$ horizontally arranged from left to right
	|	`(VBox` $e_1 \cdots e_n$`)`	shows $e_1 \cdots e_n$ vertically arranged from top to bottom
	|	`(Glue` n`)`	create a space of width (if in an hbox) or height (if in a vbox) n
	|	`Fill`	creates a space that has no width or height, but is infinitely stretchable
	|	`(Text "foo")`	creates the text "foo"
	|	`(Numeric %name)`	creates a numeric interactor with connector "name"

A *connector* is a string which is used by the application program to access and modify the value of an interactor.

By convention, files containing FormsVBT expressions always end with "`.fv`". Let's assume that the following expression, which describes the form shown in Fig. 4, is contained in a file `Input.fv`:

```
(Rim
  (Pen 20)
  (VBox
    (Text "How many elements to sort?")
    (Glue 5)
    (HBox Fill (Numeric %data) Fill)))
```

This file is one of the ominous *resources* we mentioned above. We have to declare it in our `m3makefile`:

```
...
resource (Input.fv)
bundle (SortBundle)
...
```

Now we have to modify the `SortAlg` module to query the Input Control Panel before creating a new set of random numbers.

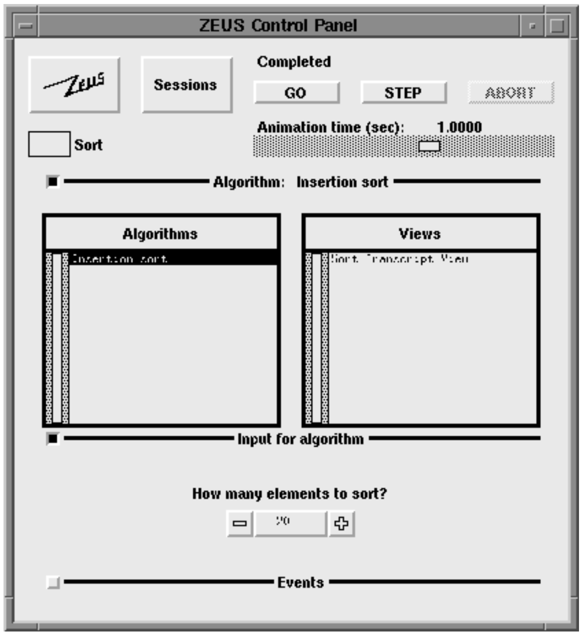


Fig. 5: The new Central Control Panel

The first thing we have to do is to provide a field in the `InsertionSort` type to hold a `FormsVBT.T` object, that is, the widget object created at runtime from the expression in `Input.fv`. But it turns out that there is already such a field: `InsertionSort` is a subtype of `Algorithm.T`, and this object type provides a field `data`, which can hold an object of type `VBT.T` or a subtype thereof (`FormsVBT.T` being one of them). Next, we have to modify the procedure `New` to create a `FormsVBT.T` object whenever it creates an `InsertionSort` object:

```
PROCEDURE New (): Algorithm.T =
  BEGIN
    RETURN NEW (InsertionSort, data := ZeusPanel.NewForm("Input.fv")).init();
  END New;
```

And finally, we have to modify `GetData` to query the Input Control Panel for the value of the numeric interactor before generating a new set of random numbers. We can query a numeric interactor by calling the function `FormsVBT.GetInteger` and passing it the form the interactor is located in (i.e. `self.data`) and the name of the connector, i.e. `"data"`, referring to the `%data` connector in `Input.fv`.

```
PROCEDURE GetData (self : InsertionSort) RAISES {Thread.Alerted} =
  VAR
    t : INTEGER;
  BEGIN
    WITH a = self.a, N = self.N DO
      N := FormsVBT.GetInteger (self.data, "data");
```

...

Of course, as we refer to the module `FormsVBT`, we have to add it to our import list.

If we now remake the `Sort` application, the input section of the Central Control Panel contains the Input Control Panel. We can adjust the values by pressing the "+" and "–" buttons of the numeric interactor, or by clicking and typing into its text field.

5 Building a Code View

Zeus provides an easy way to view the code of the algorithm as it is executed, with the current statement being highlighted. Such a view is called a *Code View*. The code displayed in a code view is not the actual code used to implement the algorithm, it does not have to be written in the same language, as a matter of fact, it does not have to be written in any programming language at all. The "code" used by a code view is a simple piece of text, containing a set of *non-overlapping* regions[4]. These regions are marked by a special character and numbered by positive integers.

So, building a Code View involves two steps: first, we create the pseudo-code file, and then we instrument our real code to highlight the appropriate regions in the pseudo-code.

Here is a pseudo-code file for insertion sort (let's call it `Insertion.pseudo`):

```
@Insertion
PROCEDURE InsertionSort (a : ARRAY OF INTEGER) @=
  VAR
    j, v : INTEGER;
  BEGIN
    @1 FOR i := 2 TO LAST(a) DO@
      @2 v := a[i];@
      @3 j := i;@
      @4 WHILE a[j - 1] > v DO@
        @5 a[j] := a[j - 1];@
        @6 DEC (j);@
      END;
      @7 a[j] := v;@
    END;
  END InsertionSort;
@Insertion
```

Next, we have to declare the code view whenever we create a new `InsertionSort` object. `Algorithm.T`, the supertype of `InsertionSort` which also contained the `data` field, contains a `codeViews` field. So we just have to modify our procedure `New` as follows:

```
PROCEDURE New (): Algorithm.T =
  BEGIN
    RETURN NEW (InsertionSort,
                data := ZeusPanel.NewForm("Input.fv"),
                codeViews := List.List1 (List.List2 ("Modula-3 Code View", "Insertion.pseudo"))
               ).init();
  END New;
```

[4] This turns out to be a slight limitation. Although we can build Code Views for "statement-oriented" languages like Pascal, we cannot build them for "expression-oriented" languages like Lisp.

`List.List1` constructs a one-element list, `List.List2` a two-element list. `codeViews` expects a list of n two-element lists. Each two-element list contains two values of type `TEXT`, the first one being an arbitrary name that will appear in the View Selection Panel, and the second one being the name of the file that contains the pseudo-code[5].

Finally, we have to instrument the procedure `Run` with calls to highlight the right regions. This is done as follows:

```
PROCEDURE Run (self : InsertionSort) RAISES {Thread.Alerted} =

  PROCEDURE At (line: INTEGER) RAISES {Thread.Alerted} =
    BEGIN
      ZeusCodeView.Event (self, line);
    END At;

  VAR
    j, v : INTEGER;
  BEGIN
    GetData (self);
    ZeusCodeView.Enter (self, "Insertion");
    WITH a = self.a, N = self.N DO
      FOR i := 2 TO N DO
At(1);
At(2);        v := a[i];
At(3);        j := i;
              WHILE a[j - 1] > v DO
At(4);
At(5);          a[j] := a[j - 1];
                SortIE.SetVal (self, j, a[j]);
At(6);          DEC (j);
              END;
At(4);
At(7);        a[j] := v;
        SortIE.SetVal (self, j, v);
      END
    END
  END Run;
```

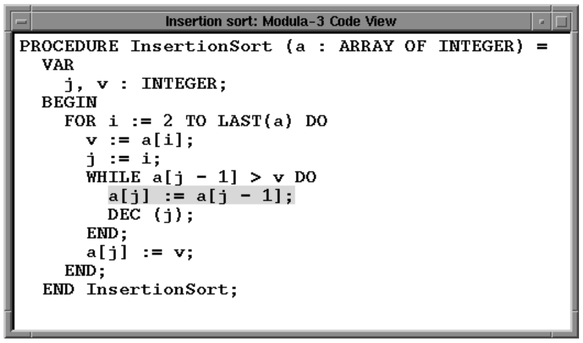
The call to `ZeusCodeView.Enter` pops up a window on the section of the pseudo-code between the two occurrences of `@Insertion`, and highlights the region between the first `@Insertion` and the following `@`. Calling `At(`n`)` causes a call to `ZeusCodeView.Event`, which highlights the region between `@`n and `@` in the code view window.

After we have added `List` and `ZeusCodeView` to our import list, and updated our `m3makefile` by adding a line

```
...
resource (Insertion.pseudo)
...
```

we can remake our application and run it again. Fig. 6 shows the Code View we have just constructed.

[5]The $1,000,000 question: How would you connect *two* Code Views to the algorithm?



```
PROCEDURE InsertionSort (a : ARRAY OF INTEGER) =
  VAR
    j, v : INTEGER;
  BEGIN
    FOR i := 2 TO LAST(a) DO
      v := a[i];
      j := i;
      WHILE a[j - 1] > v DO
        a[j] := a[j - 1];
        DEC (j);
      END;
      a[j] := v;
    END;
  END InsertionSort;
```

Fig. 6: The Code View

6 Building a Graphical View

Both the Transcript View and the Code View are textual views. So how do we construct a graphical view? Well, as it turns out, there is no straightforward recipe to do this. Each graphical view is different from the next, and must be implemented slightly different. The SRC Modula–3 environment uses its own window system, called Trestle [6, 7], which is implemented on top of X windows. On top of trestle is another layer called the VBT layer [3] (VBT stands for "Virtual Bitmap Terminal"). A VBT is a Modula–3 object which roughly represents a graphical entity — a window, a button, a scroller, a file browser, etc. `VBT.T` is the topmost class in the class hierarchy of VBTs, and it has *lots* of subclasses: `FormsVBT.T`, the class of forms objects which we used to build a control panel, `GraphVBT.T`, a VBT for displaying graph structures, `ListVBT.T`, a VBT for displaying list structures, and so on.

For our particular example — a Sticks View for an array-based sorting algorithm — we use a VBT class called `RectsVBT.T`, which allows its clients to display a set of colored rectangles.

First, we have to create a new module (let's call it `SticksView`). We start with the implementation part `SticksView.i3`:

First we want to create a subclass of `SortViewClass.T`, which is the class of sort views and one of the things that have been automatically created by `m3zume`. Recall that our Zume file looked like this:

```
OUTPUT Init (N : INTEGER [Fmt.Int]);
OUTPUT SetVal (i : INTEGER [Fmt.Int]; val : INTEGER [Fmt.Int]);
```

From this, `m3zume` generated the `SortViewClass` module, whose interface looks (almost ...) like this:

```
INTERFACE SortViewClass;

IMPORT View;

TYPE
  T <: Public;
  Public = View.T OBJECT
           METHODS
             oeInit   (N: INTEGER);
```

```
          oeSetVal (i: INTEGER; val: INTEGER);
        END;

END SortViewClass.
```

So, `SortViewClass.T` is a subclass of `View.T`, and it introduces two new methods, `oeInit` and `oeSetVal` (the `oe` stands for "output event"). We want to create a subclass of `SortViewClass.T`, add whatever fields and methods we need, and override the two methods `oeInit` and `oeSetVal` with procedures that initialize a Sticks View and that set the height of a particular stick.

```
TYPE
  T = SortViewClass.T BRANDED OBJECT
        rects : RectsVBT.T;
      OVERRIDES
        oeInit   := Init;
        oeSetVal := SetVal;
      END;
```

`rects` is a `RectsVBT.T`, a VBT that displays a set of colored rectangles. Next, we need to define the procedure `Init`, which is connected to the `oeInit` method:

```
PROCEDURE Init (self: T; N: INTEGER) =
  BEGIN
    RectsVBT.SetWC (self.rects, 0.0, 0.0, FLOAT(N + 1), FLOAT(N + 1));
    RectsVBT.SetN (self.rects, N);
    WITH rgb   = ColorName.ToRGB ("Blue"),
         color = PaintOp.FromRGB (rgb.r, rgb.g, rgb.b) DO
      FOR i := 1 TO N DO
        RectsVBT.Color (self.rects, i, color);
      END;
    END;
  END Init;
```

`Init` first sets the coordinate system of `self.rects` to range from 0 to $N+1$ in both dimensions (we have N sticks, whose height range between 1 and N, and we want to provide for some space around the borders). Then it initializes `self.rects` to be able to hold up to N rectangles. Finally, it sets the color of each of these rectangles to blue.

```
PROCEDURE SetVal (self: T; i: INTEGER; val: INTEGER) =
  BEGIN
    RectsVBT.Erase (self.rects, i);
    RectsVBT.Position(self.rects, i,
                      FLOAT(i) - 0.5, 0.5,
                      FLOAT(i) + 0.5, FLOAT(val) + 0.5);
    RectsVBT.Draw (self.rects, i);
  END SetVal;
```

The procedure `SetVal`, which is connected to the `oeSetVal` method, is supposed to adjust the height of stick i. It does this by first erasing the stick from the window, then updating its coordinates (the lower left corner being at $(i-\frac{1}{2}, \frac{1}{2})$ and the upper right corner being at $(i+\frac{1}{2}, val+\frac{1}{2})$, and the entire stick thus being 1 unit wide and *val* units high), and finally redrawing the stick.

We also need a procedure that creates a new sticks view object:

```
PROCEDURE New (): View.T =
  VAR
    view := NEW(T, rects := NEW(RectsVBT.T).init());
  BEGIN
    RETURN SortViewClass.T.init (view, view.rects);
  END New;
```

`New` creates a new `T` (a sticks view object) which contains a new `RectsVBT.T`, and then invokes the `init` method of the superclass. This `init` method will install the new view in the window system, and will make `view.rects` a child window of `view`.

Finally, we have to register the new view with the central controller. This is done with the same technique we used to register our algorithm: We call `ZeusPanel.RegisterView` in the body of the module (recall that executing a program means first executing the bodies of all modules contained in the program, and then executing the main body).

```
BEGIN
  ZeusPanel.RegisterView (New, "Sticks", "Sort");
END SticksView.
```

`Sticks` is an arbitrary string used in the View Selection Panel. `Sort` is the prefix of the Zume file `Sort.evt`.

Here is the file `SticksView.m3` in its entirety:

```
MODULE SticksView;

IMPORT ColorName, PaintOp, RectsVBT, SortViewClass, View, ZeusPanel;

TYPE
  T = SortViewClass.T BRANDED OBJECT
        rects : RectsVBT.T;
      OVERRIDES
        oeInit   := Init;
        oeSetVal := SetVal;
      END;

PROCEDURE Init (self: T; N: INTEGER) =
  BEGIN
    RectsVBT.SetWC(self.rects, 0.0, 0.0, FLOAT(N + 1), FLOAT(N + 1));
    RectsVBT.SetN(self.rects, N);
    WITH rgb   = ColorName.ToRGB("Blue"),
         color = PaintOp.FromRGB(rgb.r, rgb.g, rgb.b) DO
      FOR i := 1 TO N DO
        RectsVBT.Color(self.rects, i, color);
      END;
    END;
  END Init;

PROCEDURE SetVal (self: T; i: INTEGER; val: INTEGER) =
  BEGIN
    RectsVBT.Erase (self.rects, i);
    RectsVBT.Position(self.rects, i,
                      FLOAT(i) - 0.5, 0.5,
                      FLOAT(i) + 0.5, FLOAT(val) + 0.5);
    RectsVBT.Draw (self.rects, i);
  END SetVal;

PROCEDURE New (): View.T =
  VAR
    view := NEW(T, rects := NEW(RectsVBT.T).init());
  BEGIN
    RETURN SortViewClass.T.init (view, view.rects);
  END New;

BEGIN
  ZeusPanel.RegisterView (New, "Sticks", "Sort");
END SticksView.
```

The interface part of the module (file `SticksView.i3`) does not need to export anything:

```
INTERFACE SticksView;
END SticksView.
```

Finally, we have to update our makefile to account for the new module. Here is the final version of `m3makefile`.

```
import_obj ($(ZEUSLIB))
resource (Input.fv)
resource (Insertion.pseudo)
bundle (SortBundle)
zume (Sort)
module (SortAlg)
module (SticksView)
implementation (Main)
program (Sort)
```

If we now remake the application and run it, we get the Sticks View shown in Fig. 1.

That concludes our introduction to the Zeus algorithm animation system. We omitted some minor details (a more complete description of the FormsVBT language, construction of Data Views, exhaustive description of the different classes of VBTs), but you should have gotten a basic understanding of the Zeus technology. It might be a good idea to sit down and try out all the examples given here yourself, and to play around with them. After that, the best way to learn how more complicated animations are done is by looking at the code of existing animations. The Zeus system comes with a demo version, called *mentor*, that contains some 15 sample animations (all of which have been done in the course of two weeks by novices to the system!). See if any of those animations contain elements that you might want to reuse! Good luck and have fun!!

References

[1] Marc H. Brown. *Zeus: A System for Algorithm Animation and Multi-view Editing.* Technical Report 75, DEC Systems Research Center, February 1992.

[2] Marc H. Brown and James R. Meehan. *The FormsVBT Reference Manual – Draft Version 2.1*, January 1993. Technical Report, DEC Systems Research Center, in preparation. Available via anonymous ftp from `gatekeeper.dec.com`.

[3] Marc H. Brown and James R. Meehan (editors). *VBTkit Reference Manual – A Toolkit for Trestle.* Technical Report, DEC Systems Research Center, in preparation. Available via anonymous ftp from `gatekeeper.dec.com`.

[4] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, G. Nelson. Modula–3 Language Definition. In *ACM SIGPLAN Notices*, Vol. 27, No. 8, August 1992, pp. 15 – 42.

[5] Samuel P. Harbison. *Modula–3.* Prentice-Hall, 1992.

[6] Mark S. Manasse and Greg Nelson. *Trestle Reference Manual.* Technical Report 68, DEC Systems Research Center, December 1991.

[7] Mark S. Manasse and Greg Nelson. *Trestle Turtorial.* Technical Report 69, DEC Systems Research Center, May 1992.

[8] Greg Nelson (editor). *Systems Programming with Modula-3.* Prentice-Hall, 1991.