Journal of Visual Languages and Computing (1997) **8,** 453–486

Collaborative Active Textbooks

MARC H. BROWN AND MARC A. NAJORK

Digital Equipment Corporation, Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, U.S.A.,
{mhb, najork}@pa.dec.com

Received 9 December 1996; revised 31 May 1997; accepted 1 June 1997

This paper describes CAT, a Web-based algorithm animation system. CAT augments the expressive power of Web pages for publishing passive multimedia information with a full-fledged interactive algorithm animation system. It improves on previous Web-based algorithm animations by providing a framework that makes it easy to construct new animations, including those that involve multiple views. Because views of the same running algorithm may reside on different machines, CAT is particularly well-suited for electronic classrooms and remote learning environments. This strategy is an improvement over the electronic classroom systems we are aware of, which simply display the same X window on multiple machines and are only viable in local-area networks. We believe our framework generalizes to electronic textbooks in arbitrary domains.

1. Background

THIS PAPER describes CAT, a system for creating active and collaborative electronic textbooks. By active, we mean that the reader can interact with parts of the textbook; by collaborative, we mean that a group of people, such as a teacher and a set of students in an 'electronic classroom' setting, can share a common interaction experience. Moreover, because CAT requires only limited network bandwidth, it generalizes the notion of electronic classrooms, previously limited to local-area networks, to encompass wide-area networks as well. This makes CAT suitable for remote learning environments.

We are particularly interested in computer science education. A significant part of computer science deals with the design and analysis of algorithms. Algorithm animation, the visualization of the fundamental operations of a running program, has proven to be a powerful tool in the teaching of algorithms [18].

Our electronic textbook consists of a set of Web pages. A Web page can contain not only text and passive multimedia (e.g. images, movies and so on), but also 'applets', i.e. regions of the page that are drawn by programs dynamically loaded through the Web. A typical section of an algorithms textbook describes and analyzes an algorithm; we use the passive features of the Web page to give this conventional description and the active features to actually implement the algorithm together with one or more animated, interactive views of it. A videotape showing CAT in action is available [7].

Algorithms and views are implemented as applets. When a Web page is loaded, the applets contained in it are downloaded over the network, and then executed by and displayed in the Web browser. CAT is able to display multiple, simultaneously animated

1045-926X/97/040453+34 $25.00/0/vl970056

views of an algorithm. The views either may appear on a single page, or on separate pages, possibly in different browsers. Multiple users who attach their views to the same algorithm will see synchronized animations. The algorithm is controlled by one user, but all users can interact with their views, and the interactions are independent of each other. For example, views of 3D scenes allow the viewer to interactively change the camera position and settings; this is crucial for perceiving 3D scenes on a 2D screen.

When CAT is used in an 'electronic classroom', an instructor can control an animation and students can all view the animation simply by pointing their Web browsers at the appropriate page or pages. It is worth noting that students can, and indeed must, operate their Web browsers themselves. (Also, CAT is flexible enough to implement different floor-control schemes, where the instructor can transfer the control of the shared animation to individual students. However, we have not implemented this.)

The approach we have taken is in stark contrast to the other electronic classroom systems we are aware of. The most common approach is to use an X protocol multiplexer, such as XMX or Shared-X, which displays an arbitrary X window on multiple workstations. The advantage of this approach is that any software can be used without modification. The drawback is that students are completely passive and that there is the potential for significant network traffic.

There are numerous applets of animated algorithms available on the Web. For example, Erickson maintains an index listing several dozen Web-based animations of computational-geometry algorithms [14]. Most of these animations are stand-alone, self-contained applets, while others follow a client–server model where the visualization is displayed by an applet and the algorithm is executed on a remote server. Hausner and Dobkin's system is representative of the stand-alone approach [16]; Mocha is an excellent example of the client–server paradigm [2]. Among these systems, CAT stands out in a number of ways. To the user, CAT animations are the only Web-based algorithm animations with support for collaboration. To the programmer, CAT provides a framework and a rich infrastructure for constructing algorithm animations; most other algorithm animation applets were constructed in an *ad hoc* fashion. Our framework is based on the programming model pioneered by BALSA [10] and followed by most algorithm animation systems. In this model, an algorithm is separated from views and they are connected by way of 'interesting events'. The system provides the glue for relaying the 'interesting events' generated by the algorithm to the views. CAT also provides a high-level animation library [13] that is tailored for algorithm animation and well-proven in practice. Finally, CAT provides an algorithm-independent control panel for starting, pausing and stopping the animation and controlling its speed.

There are also Web pages that give access to algorithm animations, but the animations are not part of the Web page. For example, Stasko has a Web page that allows a user to run XTango on a remote machine, with the display set to the client's X workstation, exploiting the network transparency provided by the X window system [20]. Finally, there are a variety of MPEG and QuickTime movies of animated algorithms on the Web, such as our animation of Heapsort [5] ; obviously, movies are completely passive.

Although not 'algorithm animation' *per se*, we should mention Ibrahim's use of the Web as a front-end to (the logical equivalent of) a symbolic debugger [17]. The system allows users to run a program on the Web server, and to insert breakpoints in the code, display the contents of variables, and advance execution either line by line, or until

a breakpoint is reached. The display of the variables is text-only, using HTML forms, and the display is updated by loading new pages.

The contents of our pages is similar in spirit to Gloor's CD-ROM [15], which complements the textbook *Introduction to Algorithms* by Corman, Leiserson and Rivest. A fundamental difference is that CAT is collaborative, whereas Gloor's system is single-user. A second difference is that CAT gives the user control over the choice and placement of views of a program. A third difference is that we use the Web as our platform; Gloor used Hypercard.

The rest of this paper is organized as follows: The next section shows CAT in a classroom setting during a lecture on binpacking algorithms. In Section 3, we show how the binpacking animation was implemented. Section 4 describes how CAT is implemented. Section 5 describes the two animation libraries that CAT provides, and in Section 6, we take a look at a Java-based reimplementation of CAT. We conclude by summarizing the contributions of this work.

2. User Perspective

This section shows how CAT might be used during a lecture on binpacking algorithms. The binpacking problem is to fit a number of blocks of varying sizes into bins with fixed capacity. The 'first-fit' algorithm examines the bins in order, and adds the new block into the first bin that has enough room. In the animations of binpacking (see Figures 1–3), each block is shown as a rectangle whose height corresponds to its size, and the bins are drawn from left to right.

Figures 1 and 2 show WebScape, a Mosaic-style Web browser. The instructor's screen, Figure 1, displays a Web page with two views of the algorithm (a 'Probes' view, and below that, a 'Packing' view just barely visible) and a control panel. The control panel contains general controls ('GO', 'PAUSE' or 'RESUME', and 'ABORT' buttons, and a slider for adjusting the speed of the animation) and algorithm-specific controls (numeric widgets for specifying the number of bins and blocks). Figure 2 shows a student's screen at the same time that Figure 1 was taken. The student is looking at three views of the same algorithm; these views are embedded into a different Web page. The control panel is not part of the student's Web page; instead, there is a 'Glossary' view showing the number of bins and blocks specified by the instructor. Below that is the 'Probes' view (the block numbered 21 is clearly too large to be inserted into the 6th bin, and it is about to slide over to the 7th bin for consideration) and the 'Transcript' view (showing each event generated by the algorithm, along with the parameters). The student can scroll through the 'Transcript' view and clear its contents. However, he cannot control the algorithm; this can only be done through the control panel on the instructor's workstation.

One can imagine a more embellished version of this page. For example, there might be a link to a page containing more detailed views, oriented towards students having problems understanding the program using just the 'Probes' and 'Transcript' views. Similarly, there might be a link to a page with an alternative to the 'Packing' view, visible in Figure 3, that uses grayscale intensity rather than color hue. Such a page would be intended for students who are color-blind or have difficulty perceiving differences in hue. The page shown in Figure 2 contains a link to the algorithm source code. A student

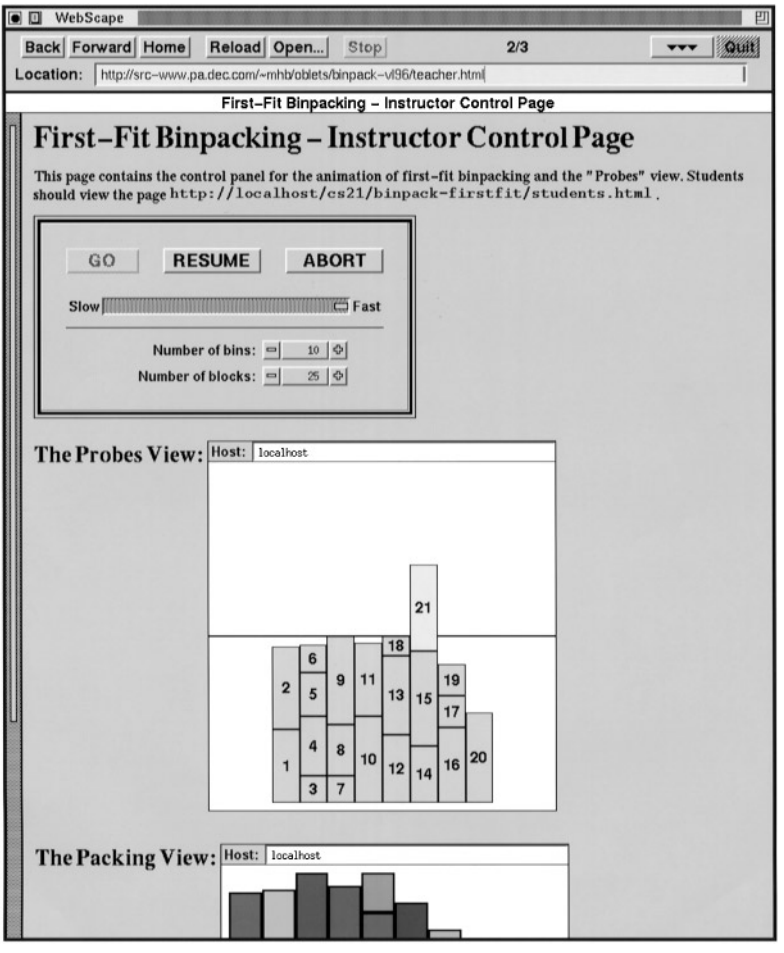


Figure 1. An instructor's workstation, ash.pa.dec.com, during a lecture on first-fit binpacking

can follow this link at any time, and following the link will not interrupt the animation (of course, the animation will not be visible until the student returns to this page).

It is important to realize that an unlimited number of students can be viewing this same page. CAT ensures that all views will be synchronized. The instructor controls how long each 'interesting event' will take using the slider in the control panel. Animations have the same duration on all computers in the classroom, regardless of the type of machine. Thus, on low-end machines, fewer frames will be displayed for a given event than on high-end machines. This is in contrast to the window multiplexing approach taken by XMX and Shared-X, where the least powerful machine determines the frame rate. In addition, our framework makes little demand on the network, because the only network traffic are the interesting events and their parameters.

Another benefit of our framework over XMX and Shared-X is that each student has interactive control over the views he is seeing. For example, he can scroll through the

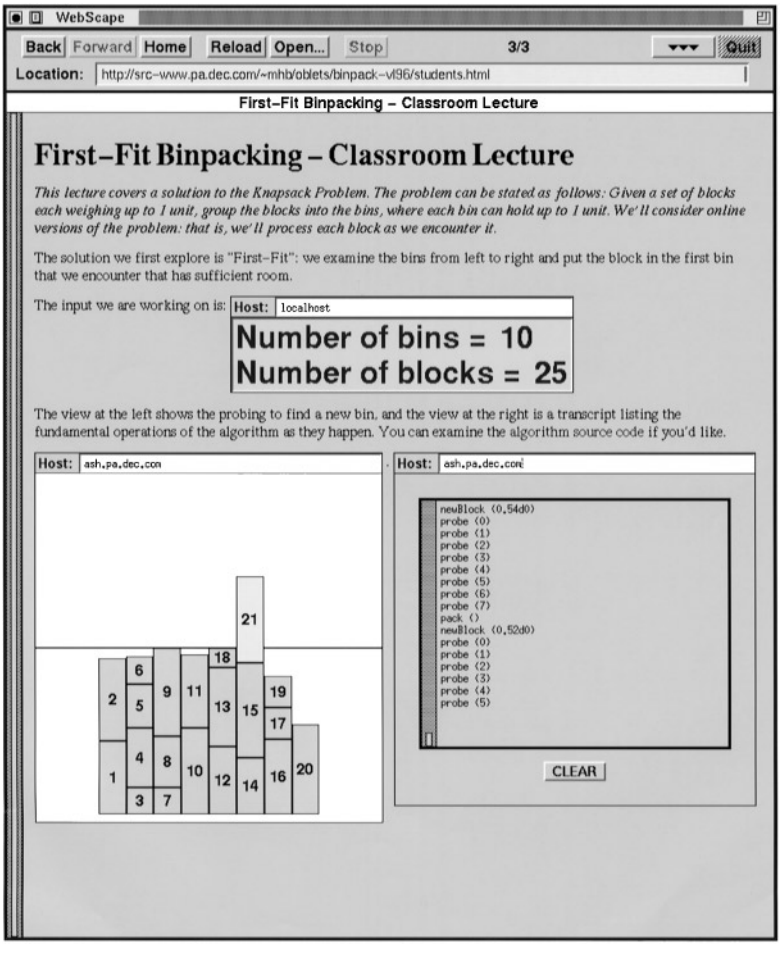


Figure 2. A student's workstation when the screen in Figure 1 was captured. Note that the student specified the host ash.pa.dec.com in order to connect to the algorithm running on that machine

'Transcript' view and clear its contents without affecting the 'Transcript' view seen by any other student. A more embellished 'Probes' view might allow a student to customize the display, for example, by adding color to the blocks. The advantage of our framework becomes even more pronounced when the views display 3D objects, as we shall see in Figures 8 and 9.

Figure 3 shows a different set of Web pages for first-fit binpacking. These pages are displayed using DeckScape [9], a Web browser that shows different Web pages in different windows. The page in the upper left contains links to five other pages, three of which are currently visible, and a control panel. This figure shows a typical configuration that a student would encounter when using CAT as an 'electronic textbook' for self-study. The Web pages are clearly different from those in Figures 1 and 2; however, the applets are *exactly* the same.

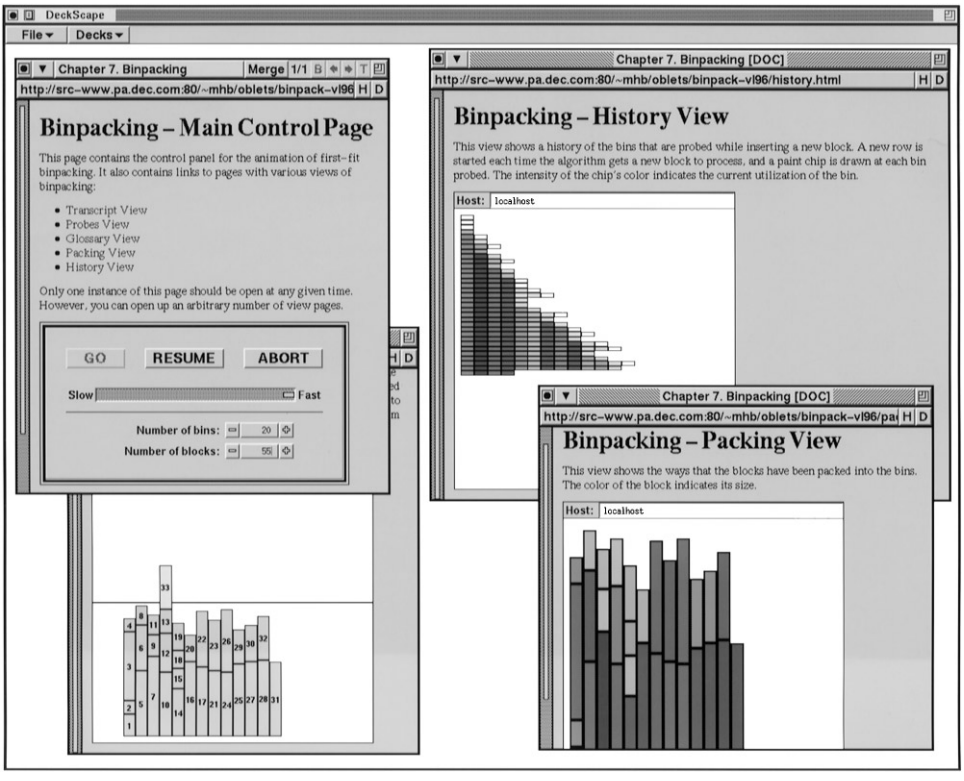


Figure 3. A user interacting with the chapter on binpacking in an 'electronic textbook'

3. Author Perspective

Our framework follows the BALSA approach: Strategically important points of an algorithm are annotated with procedure calls that generate 'interesting events'. These events are reported to an event manager, which in turn forwards them to all registered views. Each view responds to interesting events by drawing appropriate images.

The task of animating an algorithm can be divided into three steps. The first step is to identify the interesting events. The second is to implement the algorithm and annotate it with the interesting events. The final step is to implement one or more views. The algorithm and the views are implemented in Obliq, an interpreted object-oriented language [12].

In our Web-based setting, we also need to create Web pages that contain the algorithm and the views.

The remainder of this section elaborates on these steps, using the first-fit binpacking algorithm as a running example.

3.1. The Events

The 'interesting events' for binpacking algorithms are defined as follows:

```
setup     (numBins [fmt_int], numBlocks [fmt_int])
newBlock  (wt [fmt_real])
probe     (bin [fmt_int])
pack      ( )
```

An event definition consists of the name of the event, followed by a list of parameters. Each parameter is annotated with a procedure for converting its value into a string. The procedures **fmt_int** and **fmt_real** are predefined for converting integers and reals to strings. Transcript views (like the one shown in Figure 2) list the events generated by an algorithm, along with their arguments. Their code is generated automatically from the event definition file; the conversion procedures are used within that code to format the arguments of the events.

The **setup** event is generated once at the beginning to define the number of available bins and the number of blocks to be packed. Each block weighs between 0 and 1 units. The bins are numbered starting at 0, and each bin can hold at most 1 unit of weight. The **newBlock** event is generated whenever the algorithm gets a new block to pack; the weight of the new block is **wt**. The **probe** event is generated when the algorithm checks bin **bin** to see if the new block can be added to it. The **pack** event is generated when the algorithm decides to add the new block to the bin most recently probed.

The following regular expression defines the stream of interesting events generated by a binpacking algorithm:

```
setup (newBlock probe+ pack)*
```

When designing an algorithm animation, there is no right or wrong set of events, just as there is no right or wrong way to break a large system into procedures. We usually choose to have narrow interfaces, that is, we use as few parameters as possible for each event. As a result, some views may need to maintain state that is already being maintained by the algorithm. For example, the 'Probes' view we shall show later needs to maintain the utilization of each bin, information that is also maintained by the algorithm.

3.2. The Algorithm

In our framework, an algorithm is defined through an Obliq object named **alg**. This object must have two fields: **vbt** and **go**.

The **vbt** field is bound to an algorithm-specific input panel that will be incorporated into the control panel; in this example, the definition of the panel is loaded from the relative URL 'alg. fv'. We will look at the contents of 'alg. fv' later; for now, it suffices to say that it contains two numeric widgets, named **bins** and **blocks**, which are used for specifying the number of bins and blocks, respectively.

The **go** field is bound to a method that is called when the user hits the 'GO' button in the control panel. This method implements the algorithm as one would find it in a text book, along with the annotations for generating the interesting events.

```
let alg = {
  vbt ⇒ form_fromURL(BaseURL & ''alg.fv''),
```

```
  go ⇒
    meth (self, disp)
      let numBins = form_getInt(self.vbt, ''bins'');
      let numBlocks = form_getInt(self.vbt, ''blocks'');
      disp.setup(numBins, numBlocks);

      let totals = array_new(numBins, 0·0);
      for block = 0 to numBlocks − 1 do
        let amt = real_float(random_int(20, 90)) * 0·01;
        disp.newBlock(amt);
        var bin = 0;
        loop
          disp.probe(bin);
          if (totals[bin] + amt) ⩽ 1·0 then exit end;
          bin := bin + 1;
          if bin is numBins then exit end;
        end;
        if bin is numBins then exit end;
        totals[bin] := totals [bin] + amt;
        disp.pack( );
      end;
    end
};
```

The go method takes two parameters: self refers to the object in which the method is contained and disp is the animation control object. This object is responsible for forwarding interesting events to all registered views, and returning control to the algorithm only after all views have completed their animations.

The first few lines of the method retrieve the numbers of bins and blocks specified by the user, and then generate a setup event.

For the sake of completeness, here is the contents of 'alg.fv', which defines the algorithm-specific input panel. The user-interface specification is written using FormsVBT [1]:

```
(VBox
  (HBox
    (Text RightAlign ''Number of bins: '')
    (Numeric %bins (Min 1) =10))
  (Glue 5)
  (HBox
    (Text RightAlign ''Number of blocks:'')
    (Numeric %blocks (Min 1) =20)))
```

3.3. The Views

This section examines the 'Probes' view from before. This view uses a GraphVBT widget [13]. GraphVBT is a high-level animation package based on the metaphor of

a graph consisting of vertices and edges. Each vertex has various attributes, such as position, size, shape, color, border width and label. An edge connects two vertices and it has attributes such as color and thickness. Vertices can be repositioned, and such movement is shown by smooth animation, inspired by the Tango system [21]. Section 5.1 describes the parts of the GraphVBT interface that are used by the animations shown in this paper.

In our framework, a view is defined through an Obliq object named **view**. This object must have a field **vbt** (in this case, bound to a GraphVBT widget), and methods for each interesting event. Here is the code:

```
let view = {
  vbt          ⇒ graph_new( ),

  currVertex   ⇒ ok,
  currWt       ⇒ ok,
  lastProbe    ⇒ ok,
  currId       ⇒ 1,
  totals       ⇒ ok,

  setup ⇒
    meth (self, numBins, numBlocks)
      self.currId := 1;
      self.totals := array_new(numBins, 0·0);
      graph_clear(self.vbt);
      graph_setWorld(self.vbt, −2·0, float(numBins), 2·0, 0·0);
      let v0 = graph_newVertex(self.vbt);
      graph_moveVertex(v0, − 10·0, 1·0, false);
      let v1 = graph_newVertex(self.vbt);
      graph_moveVertex(v1, float(numBins) + 10·0, 1·0, false);
      let e = graph_newEdge(v0, v1);
      graph_setEdgeWidth(e, 0·01);
      graph_redisplay(self.vbt);
    end,

  newBlock ⇒
    meth (self, wt)
      let v = graph_newVertex (self.vbt);
      graph_moveVertex(v, −1·0, wt/2·0, false);
      graph_setVertexSize(v, 1·0, wt);
      graph_setVertexColor(v, color_named(''LightGray''));
      graph_setVertexBorder(v, 0·01);
      graph_setVertexLabel(v, fmt_int(self.currId));
      graph_setVertexLabelColor(v, color_named(''Black''));
      graph_redisplay(self.vbt);
      self.currVertex:= v;
      self.currWt := wt;
      self.currId:= self.currId + 1;
    end,
```

```
  probe ⇒
    meth (self, bin)
      let xpos = 0·5 + float(bin);
      let ypos = self.totals[bin] + self.currWt/2·0;
      graph_moveVertex(self.currVertex, xpos, ypos, true);
      graph_animate(self.vbt, 0·0, 1·0);
      self.lastProbe := bin;
    end,

  pack ⇒
    meth (self)
      self.totals [self.lastProbe] := self.totals[self.lastProbe] + self.currWt;
      graph_setVertexColor(self.currVertex, color_named(''Pink''));
      graph_redisplay(self.vbt);
    end
};
```

The **setup** method does three things: First, it initializes an array, **totals**, which holds the current utilization of the bins. Second, it initializes the GraphVBT widget, i.e. it blanks the widget's display and then defines a world coordinate system. Third, it draws a horizontal line, midway through the widget. The line indicates the maximum capacity of each bin.

The **newBlock** method creates a new GraphVBT vertex for the new block. The shape is rectangular by default, the width is set to 1, the height is set to **wt**, the color is set to a light gray and the vertex has a black border whose width is 0·01. The vertex is then positioned at the far left and made visible. Finally, we store the block's weight and a handle to the vertex (so other events can reposition and recolor the vertex).

The **probe** method moves the vertex representing the new block to the top of bin **bin**. This movement is animated smoothly; its speed is determined by the setting of the slider in the control panel. We then record the bin being probed, for use by the **pack** event.

Finally, the **pack** method increments the utilization of the bin most recently probed by the weight of the new block. It then changes the color of the vertex corresponding to the new block to pink, and updates the display.

3.4. The HTML

The CAT system consists of two components: a family of Obliq-aware Web browsers (e.g. WebScape and DeckScape), and CATalyst, an offline program that takes the author-supplied event, algorithm, and view files as input and fills in the glue that connects the algorithm and the views.

In particular, CATalyst generates 'proxy objects' for each algorithm object and view object. These proxy objects hide the intricacies of algorithm-view communication and distributed computations from the author of the algorithm animation. Given an event file 'BP.evt' and an algorithm file 'alg.obl', CATalyst produces the file 'BPalg.obl' that contains the proxy object for the algorithm. Similarly, there will be a file 'BPview.obl' generated for a view in the file 'view.obl'.

The display of an algorithm proxy object shows the control panel and the algorithm-specific input panel. The display of the view proxy object shows the view's display (e.g.

a GraphVBT widget) and a type-in field for identifying the machine on which the algorithm is running.

It is the proxy objects that are actually embedded into Web pages. We use **object**, an HTML tag proposed by the World Wide Web Consortium for inserting rectangular multimedia objects into HTML pages [22]. The markup for putting the proxy object stored at URL 'BPalg.obl' into a document is as follows:

```
⟨object codetype = "application/x-oblet" classid = "BPalg.obl"⟩
⟨/object⟩
```

The **object** tag also supports a variety of standard attributes, such as suggested dimensions, border size and alignment. If suggested dimensions are not specified, the preferred dimensions of the display of the proxy object are used.

4. System Perspective

As mentioned previously, CAT consists of two pieces: the CATalyst preprocessor and a family of Web browsers that support applets written in Obliq. The most distinguishing feature of Obliq is that it has distributed scope: objects can reside in different address spaces and on different machines and are accessed in a uniform fashion regardless of where they reside. Obliq's inherent support for distributed computations makes it easy to write applets that collaborate with one another. We call our applets Oblets (**Ob**liq app**lets**) [6].

The proxy objects we mentioned earlier for the algorithm and for the views are actually Oblets. CATalyst uses the event definition file and the user-supplied **alg** and **view** object files to generate files containing these Oblets. CATalyst also generates a 'Transcript' view from the event definition file. Figure 4 illustrates this process.

In addition to the algorithm and view Oblets, there is an animation control object (the **disp** parameter to the **alg** method in Section 3.2), also generated from the event definition file. The animation control object resides on the machine where the algorithm Oblet runs, and its primary purpose is to communicate information from the algorithm to the views. To do this, the algorithm control object maintains a list of view Oblets and provides a collection of methods that correspond to the interesting events. The algorithm does not send events directly to the views, but rather to the animation control object, which forwards them to all registered views. Here is the body of a method of the animation control object corresponding to the interesting event named **foo**:

```
foo ⇒
  meth (self, arg1, arg2, . . . )
    let thrds =
      foreach v in self.views map
        fork(proc( ) v.view.foo(arg1, arg2, . . . ) end, 0)
      end;
    foreach t in thrds do join(t) end;
    . . .
  end,
```

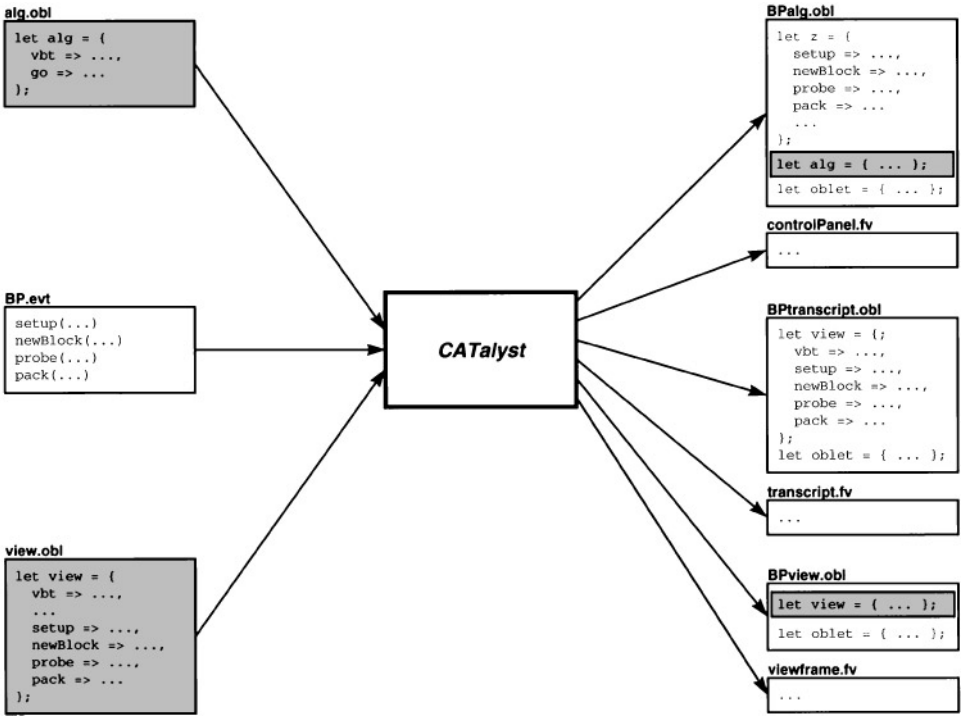


Figure 4. Inputs and outputs to the CATalyst preprocessor

The method iterates through the array of view Oblets and forks a thread per Oblet. These threads call the foo method of each user-defined view. Next, the method waits until all of the threads have completed. Before returning control to the algorithm, the method checks if the user hit the 'PAUSE' or 'ABORT' button. If the user hit the 'PAUSE' button, the method blocks until the 'RESUME' button is hit. If the user hit the 'ABORT' button, an exception is raised, which causes the algorithm to terminate. Complete details are presented in Section 4.2.

We should emphasize that the statement v.view.foo(···) is actually extremely powerful. On the surface, it appears to be rather boring: invoke the method foo of the object v.view. However, because of Obliq's distributed nature, this object does not need to reside on the same machine where the caller to foo (i.e. the animation control object) resides. In the electronic classroom setting, the algorithm Oblet and the animation control object reside on the instructor's machine and view Oblets reside on both the instructor's and the students' machines.

The remainder of this section provides more details about Oblets and about the Oblets that are generated from the user-supplied alg and view object files.

4.1. Oblets in a Nutshell

An Oblet is an Obliq program that defines a variable named oblet. This variable must contain an Obliq object with at least two fields: vbt and run. The vbt field is bound to

a widget that will be installed in the Web page when the page containing the Oblet is loaded. The run field is bound to a method that is invoked just after the vbt field is evaluated.

As mentioned above, Obliq is an inherently distributed language. Obliq objects can be distributed over heterogeneous machines across the Internet. The only statements that are specific to distribution are net_export, which exports an object to remote parties (through the mediation of a nameserver) and net_import, which imports a remote object from a nameserver. Once a remote object is imported, it is indistinguishable from a local object. Thus, from a programmer's perspective, there is no difference between local and remote objects.

Figures 5 and 6 show a simple distributed application that illustrates the fundamentals of Oblets. The application consists of two Oblets running on two different machines. One Oblet, the server, allows a user to select one of four colors (Figure 5). The other Oblet, the client, displays the name of the chosen color inside a rectangle of that color (Figure 6).

The screen dump in Figure 5 shows the Web page containing the server Oblet. The Oblet's user interface consists of four radio buttons, labeled 'Red', 'Green', 'Blue' and 'Black'. The FormsVBT description of this user interface consists of a Radio component containing a vertical arrangement of four Choice components. The Radio is named ColorChoice and the Choice components are named Red, Green, Blue and Black.

The oblet object of the server Oblet has a field named client, in addition to the required vbt and run fields. The vbt field contains a form based on the above FormsVBT description. The client field is initialized to an object with one method, changeColor, which does nothing. The run field is a method that first defines a callback procedure named cb, attaches this callback to the Radio component and finally exports the oblet object under the name ColOblet to the nameserver on machine ash.pa.dec.com.

The callback procedure cb is invoked each time the user clicks on a radio button in the server Oblet. The procedure calls form_getChoice to determine which button was pressed. This call returns the name of the Choice component, i.e. the string Red, Green, Blue or Black . The callback then calls the changeColor method of the client field, passing the selected color. Initially, this is a no-op (since the changeColor method does nothing); as we shall see, the client field will be changed to refer to the client oblet object, once the client's Oblet is created.

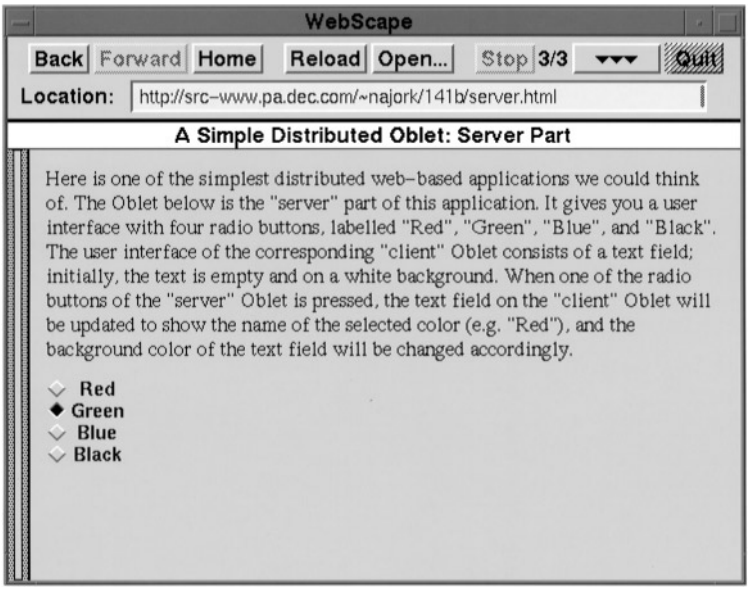
The screen dump in Figure 6 shows the Web page containing the client Oblet. The Oblet's user interface consists of a colored rectangle surrounding a string. The FormsVBT description of this user interface consists of a Text component named Col, constrained to be 200 by 100 pixels, and showing the string 'Black' in a 48.0 point font, white on a black background.

The oblet object of the client Oblet has a field named changeColor, in addition to the required vbt and run fields. The vbt field contains a form based on the FormsVBT description above. The run method imports the server's oblet object from the nameserver, and then overrides that object's client field to refer to the client oblet . That is, in the statement

```
server.client := self;
```

the object server resides on the server machine, while self resides on the client machine. After this statement is executed, the server's client field refers to an object on the client

Web page showing the server Oblet:



Description of the server Oblet's user interface:

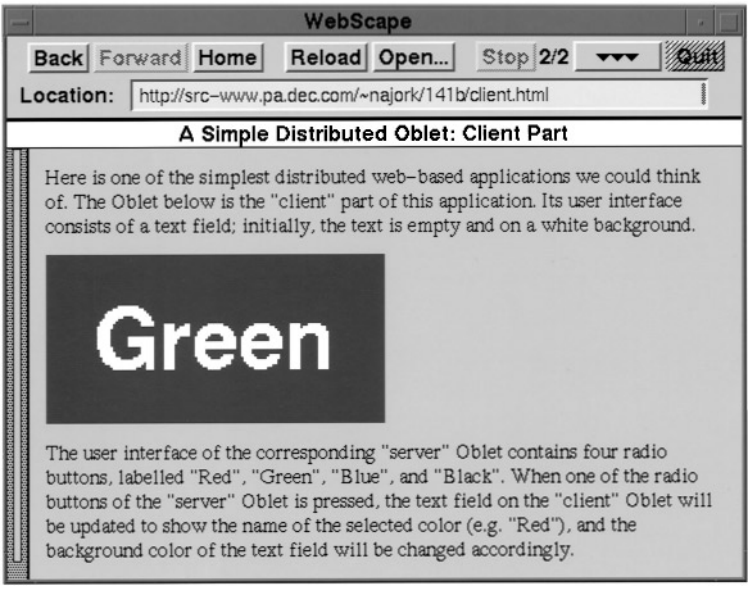
```
(Radio %Colorchoice
  (VBox
    (Choice %Red ''Red'')
    (Choice %Green ''Green'')
    (Choice %Blue ''Blue'')
    (Choice %Black ''Black'')))
```

Obliq code for the server Oblet:

```
let oblet = {
    vbt ⇒ form_fromURL(BaseURL & ''server.fv''),
    client ⇒ changeColor ⇒ meth(self, col) end},
    run ⇒
      meth(self)
        let cb = proc(fv)
          let col = form_getChoice(fv, ''ColorChoice'');
          self.client.changeColor(col)
        end;
        form_attach(self.vbt, ''ColorChoice'', cb);
      net_export(''ColOblet'', ''ash.pa.dec.com'', self);
    end
};
```

Figure 5. The 'server' Oblet; the corresponding 'client' Oblet appears in Figure 6

Web page showing the client Oblet:



Description of the client Oblet's user interface:

```
(Shape (Width 200) (Height 100) (LabelFont (PointSize 480))
  (Text %Col (BgColor ''Black'')(Color ''White'') ''Black''))
```

Obliq code for the client Oblet:

```
let oblet = {
  vbt ⇒ form_fromURL(BaseURL & ''client.fv''),
  changeColor ⇒
    meth(self, col)
      form_putColorProp(self.vbt, ''Col'', ''BgColor'', col);
      form_putText(self.vbt, ''Col'', col)
    end,
  run ⇒
    meth(self)
      let server = net_import(''ColOblet'', ''ash.pa.dec.com'');
      server.client := self;
    end
};
```

Figure 6. The 'client' Oblet; the corresponding 'server' Oblet appears in Figure 5

machine. Consequently, the statement

```
self.client.changeColor(col);
```

executed by the server's callback procedure, invokes the changeColor method on the client's machine. This method takes the obligatory self parameter and a parameter col. Since changeColor is invoked by cb, the col parameter will be the string Red, Green, Blue or Black . The changeColor method calls form_putColorProp to change the background color property of its Text component and then calls form_putText to change the string that is displayed.

4.2. The Algorithm Oblet

The file generated for the algorithm, say 'BPalg.obl', contains three parts. The first part is the algorithm control object, disp. The second part is the algorithm code supplied by the author, i.e. the object alg shown in Section 3.2. The third part is oblet, the active object that can be embedded into a Web page. So the structure of 'BPalg.obl' is as follows:

```
let disp  = {...};
let alg   = {...};
let oblet = {...};
```

For didactic reasons, we will first look at Oblet and then at disp.

The algorithm Oblet shows the generic control panel and the algorithm-specific controls. The run method has three purposes. It exports the animation control object disp to a nameserver running on the local machine; it installs the algorithm-specific controls into the generic control panel and it defines and attaches callback procedures to the widgets in the generic control panel. The widgets are the 'GO', 'PAUSE' or 'RESUME' and 'ABORT" buttons and the speed slider. Here is the FormsVBT description of the control panel, contained in the file 'controlPanel.fv':

```
(Border
  (Rim (Pen 2)
    (Border (Pen 2)
      (Frame Chiseled
        (Rim (Pen 20)
          (VBox
            (HBox (LabelFont (PointSize 140))
              (Filter (Button %go (Text %goText ''GO'')))
              (Glue 20)
              (Filter Dormant (Button %pause (Text %pauseText ''PAUSE'')))
              (Glue 20)
              (Filter Dormant (Button %abort ''ABORT'')))
            (Glue 20)
            (HBox
```

```
                    (Text ''Slow'')
                    (Frame Lowered (Scroller (Min 10) (Max 100) (Step 1) = 50 %speed))
                    (Text ''Fast''))
                (Glue 10)
                Bar
                (Glue 10)
                (HBox Fill (Generic %algInput) Fill)))))))
```

Furthermore, the code for the algorithm's **oblet** object is the following:

```
let oblet = {
  goThread ⇒ ok,

  vbt ⇒ form_fromURL(BaseURL & ''controlPanel.fv''),
  run ⇒
    meth(self)

      let goCallback =
        proc(fv)
            let go =
                proc()
                    disp_paused := false;
                    form_putReactivity(fv, ''go'', ''dormant'');
                    form_putReactivity(fv, ''pause'', ''active'');
                    form_putReactivity(fv, ''abort'', ''active'');
                    try alg.go(disp) except thread_alerted ⇒ end;
                    form_putReactivity(fv, ''go'', ''active'');
                    form_putReactivity(fv, ''pause'', ''dormant'');
                    form_putReactivity(fv, ''abort'', ''dormant'');
                    form_putText(fv, ''pauseText'', ''PAUSE'');
                end;
            self.goThread := fork(go), 0);
        end;

      let abortCallback =
        proc(fv)
          thread_alert(self.goThread);
        end;

      let pauseCallback =
        proc(fv)
          lock disp.mu do
            if disp.paused then signal(disp.cond) end;
            let label = if disp.paused then ''PAUSE'' else ''RESUME'' end;
            form_putText(fv, ''pauseText'', label);
            disp.paused := not(disp.paused);
          end;
        end;
```

```
        let speedCallback =
          proc(fv)
            let s = form_getInt(fv, ''speed'');
            graph_setSpeed(float(110-s)*0.01);
          end;

        form_putGeneric(self.vbt, ''algInput'', alg.vbt);

        form_attach(self.vbt, ''go'', goCallback);
        form_attach(self.vbt, ''abort'', abortCallback);
        form_attach(self.vbt, ''pause'', pauseCallback);
        form_attach(self.vbt, ''speed'', speedCallback);
        speedCallback(self.vbt);

        net_export(''BP'', ''localhost'', disp);

      end
};
```

The goCallback forks a thread which invokes the go method of the user-supplied algorithm object. Before calling the go method, the paused flag is set to false, indicating that the algorithm is not paused and the 'PAUSE' and 'ABORT' buttons are activated while the 'GO' button is deactivated. As we shall see, pressing the 'ABORT' button causes the thread_alerted exception to be raised. The call to go is surrounded by an exception handler that catches this exception. After the go method completes, possibly because it was aborted by the user, the 'GO' button is again activated, the 'PAUSE' and 'ABORT' buttons are deactivated, and the thread terminates.

The abortCallback is simple: it sets the 'alert' flag of the thread in which the algorithm is running.

The 'PAUSE' button is used for pausing the algorithm and resuming it again. Initially, the algorithm is running, the button is labeled 'PAUSE', and the paused flag is false. Pressing the button causes the pauseCallback to be called. The label is changed to 'RESUME' and the paused flag is set to true. As we shall see, this will cause the algorithm thread to block on a condition variable when the next interesting event occurs. Pressing the button again causes the condition variable to be signaled (thereby resuming the algorithm thread), the label to be changed back to 'PAUSE' and the paused flag to be set to false.

The speedCallback, called when the user manipulates the speed slider, changes the speed of the animation.

Finally, here is the definition of the algorithm control object, disp, generated by CAT.

```
let disp = {
  mu ⇒ mutex( ),
  cond ⇒ condition( ),
  paused ⇒ ok,
  views ⇒ [ ]

  registerView ⇒
    meth (self, view)
      self.views := self.views @ [view];
    end,
```

```
    unregisterView ⇒
      meth (self, view)
        array_removeElement(self.views, view);
      end,

    setup ⇒
      meth (self, numBins, numBlocks)
        let thrds =
          foreach v in self.views map
            fork(proc( ) v.view.setup(numBins, numBlocks) end, 0)
          end;
        foreach t in thrds do join(t) end;
        lock self.mu do
          if self.paused then
            thread_alertWait(self.mu, self.cond)
          end
        end;
        if thread_testAlert( ) then raise(thread_alerted) end;
      end,

    newBlock ⇒ meth (self, wt) . . . end,
    probe ⇒ meth (self, b) . . . end,
    pack ⇒ meth (self) . . . end
};
```

The object contains a number of algorithm-independent fields and methods, followed by one method for each interesting event. The first fields are used to implement the 'PAUSE' and 'RESUME' functionality. The **views** field is an array of view Oblets; these Oblets are registered when a user opens a Web page containing a binpacking view and connects to the machine where **disp** resides. The **registerView** and **unregisterView** methods maintain the **views** array.

Recall that the user-defined **alg** object calls **disp.setup** for communicating information to the views. The setup method of **disp** iterates through the array of view Oblets, and forks off a thread per Oblet. These threads invoke the **setup** method on the user-defined **view** object. The **foreach . . . map . . . end** construct returns an array, which in this case contains handles to the forked threads. Next, **setup** waits until all the threads have completed.

It is worth pointing out that the Oblets contained in the **views** array may reside on different machines. The inherently distributed semantics of Obliq makes the location of an Oblet transparent to the programmer; calling **v.view.setup** works regardless of whether **v.view** is a local or remote object.

Next we handle the 'PAUSE' and 'RESUME' functionality. As we saw, pressing the 'PAUSE' button causes the **paused** flag to be set to true. The **setup** method checks if this flag is true, and if so, blocks on a condition variable. The call to **thread_alertWait** will return when the condition variable is signaled or when the thread is alerted.

Finally, we handle the 'ABORT' functionality: As we saw, pressing the 'ABORT' button causes the 'alert' flag to be set. The **setup** method checks if the flag has been set,

and if so, raises an exception. The raising of the exception will cause the algorithm to terminate, by transferring control to the exception handler in the procedure **goCallback** shown above.

The contents of the other event methods, **newBlock**, **probe** and **pack** are similar to the **setup** method, with **setup** being replaced by the names of the other events and the parameter lists being changed accordingly.

4.3. The View Oblet

The file generated for a view, say 'BPview.obl', contains two parts. The first part is the view code supplied by the author, i.e. the object **view** shown in Section 3.3. The second part is the **oblet** object, needed to make 'BPview.obl' an Oblet. The code for this object is as follows:

```
let oblet = {
  view ⇒ view,
  disp ⇒ ok,
  host ⇒ ''**unconnected**'',
  vbt ⇒ form_fromURL(BaseURL & ''viewframe.fv''),

  run ⇒
    meth (self)

      let newHost =
        proc (hostName)
          let old = self.disp;
          try
            self.disp := net_import (''BP'', hostName);
            self.disp.registerView (self);
            if old isnot ok then
              old.unregisterView(self);
            end;
            self.host := hostName;
          except net_failure ⇒
          end;
          form_putText(self.vbt, ''host'', self.host);
        end;

      let hostCallback =
        proc(fv)
          newHost(form_getText(fv, ''host''))
        end;

      form_putGeneric(self.vbt, ''contents'', view.vbt);
      form_attach(self.vbt, ''host'', hostCallback);
      newHost(''localhost'');
    end
  };
```

The view Oblet, like any Oblet, has a **vbt** field and a **run** method. In addition, it has a field **view**, which is set to the user-specified **view** object, a field **disp**, which will be bound to an animation control object, and a field **host**, the name of the machine on which **disp** resides.

The **vbt** field, the widget displayed in the Web page, consists of a type-in field named **host** for specifying the machine on which the animation control object resides and the view-specific widget (e.g. a GraphVBT widget). Here is the widget's FormsVBT expression, contained in the file 'viewframe.fv':

```
(Border (Pen 1)
  (VBox
    (HBox
      (Text "Host: ")
      (Frame Lowered
        (TypeIn (BgColor "White") %host = "local host")))
    (Bar 1)
    (Generic % contents)))
```

The **run** method installs the view-specific widget, attaches the callback **hostCallback** to **host** and calls the procedure **newHost**, which tries to import the animation control object from the nameserver on the local machine.

The **hostCallback** retrieves the contents of the type-in field and passes it to **newHost**.

The **newHost** procedure tries to import an animation control object from the nameserver on the machine **hostName**. If successful, the view is registered with the new animation control object. If the view had previously been registered with another animation control object, it will be unregistered. **newHost** then caches the name of the host. Finally, regardless of whether the import succeeded or not, the type-in field is set to the contents of the **host** field. In this way, the type-in field will always show whether the view is connected, and if so, to which machine.

The **newHost** procedure assumes that the calls to **registerView** and **unregisterView** will never fail. However, it would be straightforward to handle those failures by catching the appropriate exceptions.

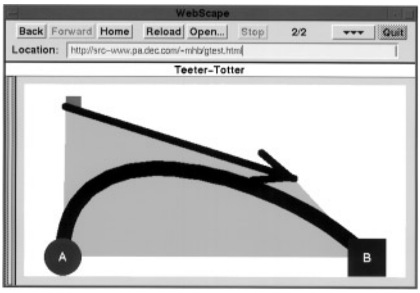
5. The Animation Libraries

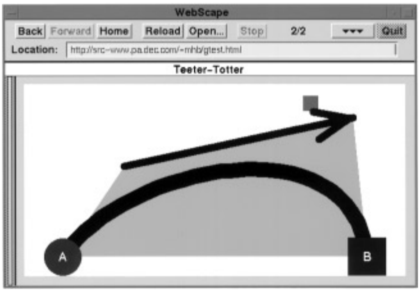
This section describes the 2D and 3D animation libraries provided by CAT. The screen dumps in Figures 7–9 show a number of CAT animations that were built using these libraries.

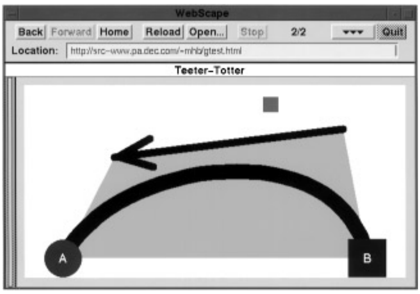
5.1. The 2D Animation Library

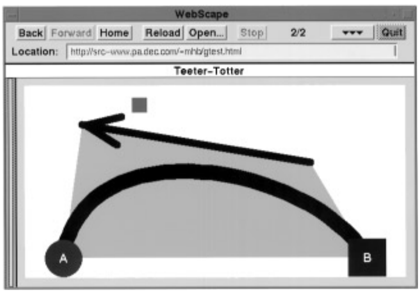
The 2D animation library (GraphVBT [13]) is based on the metaphor of a graph consisting of vertices, edges, vertex highlights and polygons. Vertices are colored rectangles or ellipses with optional label fields and highlights. Edges connect vertices; they can be straight or curved and can have arrowheads at either or both ends. Polygons fill regions defined by a sequence of vertices.

```
let oblet = {
  vbt ⇒ graph_new(),

  run ⇒
    meth(self)
      let newVertex =
        proc(x, y, clr, label, shape)
          let v = graph_newVertex(self.vbt);
          graph_moveVertex(v, x, y, false);
          if shape isnot ok then
            graph_setVertexColor(v, color_named(clr));
            graph_setVertexSize(v, 0·1, 0·1);
            graph_setVertexLabel(v, label);
            graph_setVertexShape(v, shape);
          end;
          v;
        end;


      graph_setWorld(self.vbt, 0·0, 1·0, 0·05, 0·55);
      let X = [false, true];
      let C = [[0·3, 0·3], [0·1, 0·1]];
      let D = [[0·9, 0·1], [0·7, 0·3]];
      let a = newVertex(0·1, 0·5, "red", "A", "ellipse");
      let b = newVertex(0·9, 0·5, "blue", "B", "rectangle");
      let c = newVertex(C [0] [0], C [0] [1], ok, ok, ok);
      let d = newVertex(D [0] [0], D [0] [1], ok, ok, ok);


      let p = graph_newPolygon([a, b, d, c]);
      graph_setPolygonColor(p, color_named("green"));

      let el = graph_newEdgeBezier(a, b, c, d);
      graph_setEdgeWidth(e1, 0·04);
      let e2 = graph_newEdge(c,d);
      graph_setEdgeWidth(e2, 0·02);

      let h = graph_newVertexHiLi(d);
      graph_setVertexHiLiColor(h, color_named("brown"));
      graph_setVertexHiLiBorder(h, 0·02, 0·02);


      var i = 1;
      loop
        foreach v in [c, d] do
          graph_setEdgeArrows(e2, X[i], not(X[i]));
          graph_moveVertexHiLi(h, v, true);
          graph_moveVertex(c, C [i] [0], C [i] [1], true);
          graph_moveVertex(d, D [i] [0], D [i] [1], true);
          i: = 1 − i;
          graph_animate(self.vbt, 0·0, 2·0);
        end
      end
    end
};
```









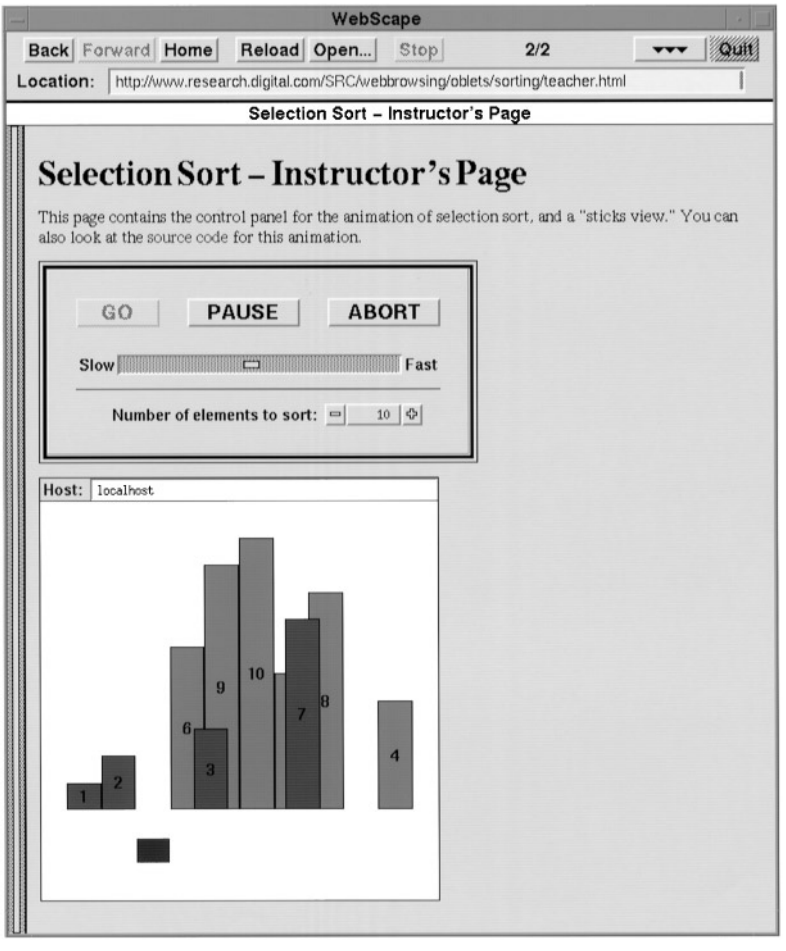


Figure 7. An animation of Selection sort. This classical view of sorting algorithms shows each element of the array as a stick whose height corresponds to its value. When the screen dump was captured, the algorithm was in the process of swapping two elements

Vertices are drawn in front of edges, edges in front of vertex highlights and vertex highlights in front of polygons. The various attributes of the objects can be changed and the library takes care of showing the changes with smooth transformations.

The Oblet on the previous page illustrates the elements of the 2D animation library. The Oblet consists of four vertices, two edges, one vertex highlight and one polygon. The vertex at the bottom-left is a red ellipse labeled 'A', and the vertex at the bottom-right is a blue rectangle labeled 'B'. There are two additional vertices that are invisible (that is, their width and height are both zero). These vertices, positioned in the upper-left and upper-right of the Oblet, are used for four reasons: a highlight moves back-and-forth between them, they are the endpoints of an arrow, the vertices of the polygon and the control point of the Bezier curve connecting the two vertices at the bottom.

Each iteration of the animation takes 2 seconds, and four actions happen: the arrow head changes direction, the highlight moves from one control point to the other and the

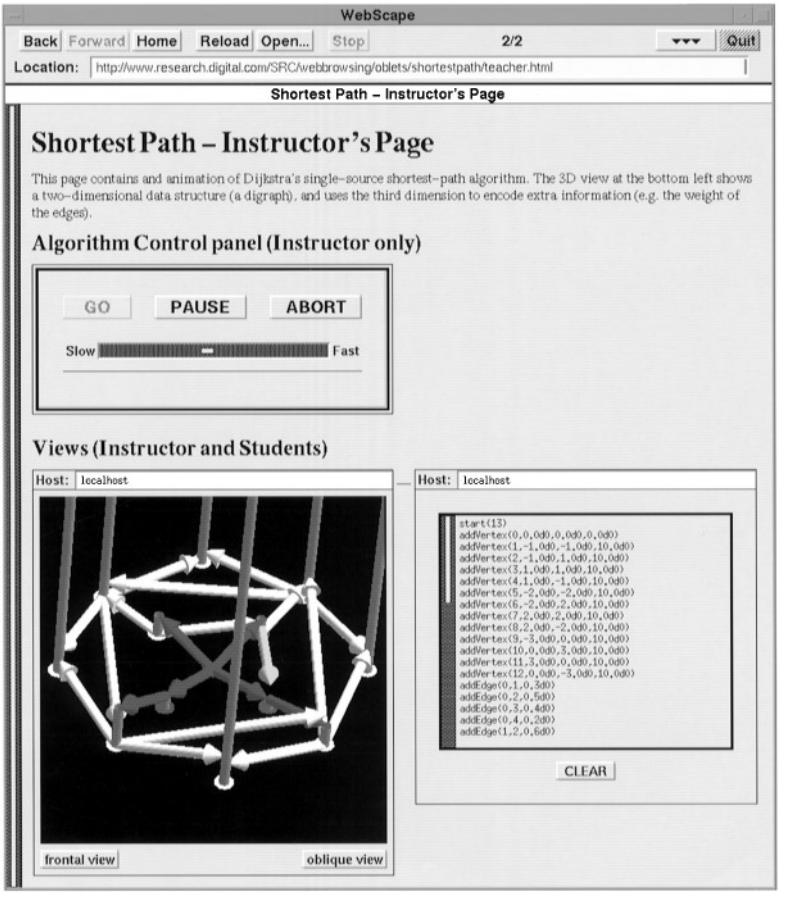


Figure 8. An animation of Dijkstra's algorithm for finding the shortest path from a source vertex to all other vertices in a weighted, directed graph. Graphs are naturally visualized as 2D structures; we use the third dimension to encode additional information such as the weight of each edge (height difference between the two ends of each arrow) and the shortest distance found so far from the source to each vertex (height of the green column)

two control points move in complementary directions (one control point moves up and out, while the other moves in and down). Moving the control points also changes the position of the arrow and the shapes of the polygon and the Bezier curve.

5.2. The 3D Animation Library

The 3D animation library (Obliq-3D [19]) is founded on three basic concepts: *graphical objects* for constructing scenes, time-variant *properties* for animating various aspects of a scene and *callbacks* for providing interactive behavior.

Graphical objects include geometric shapes (spheres, cones, cylinders and the like), light sources, cameras, groups for composing complex graphical objects out of simpler ones, and roots (which are a subtype of groups) for displaying graphical objects on the screen.

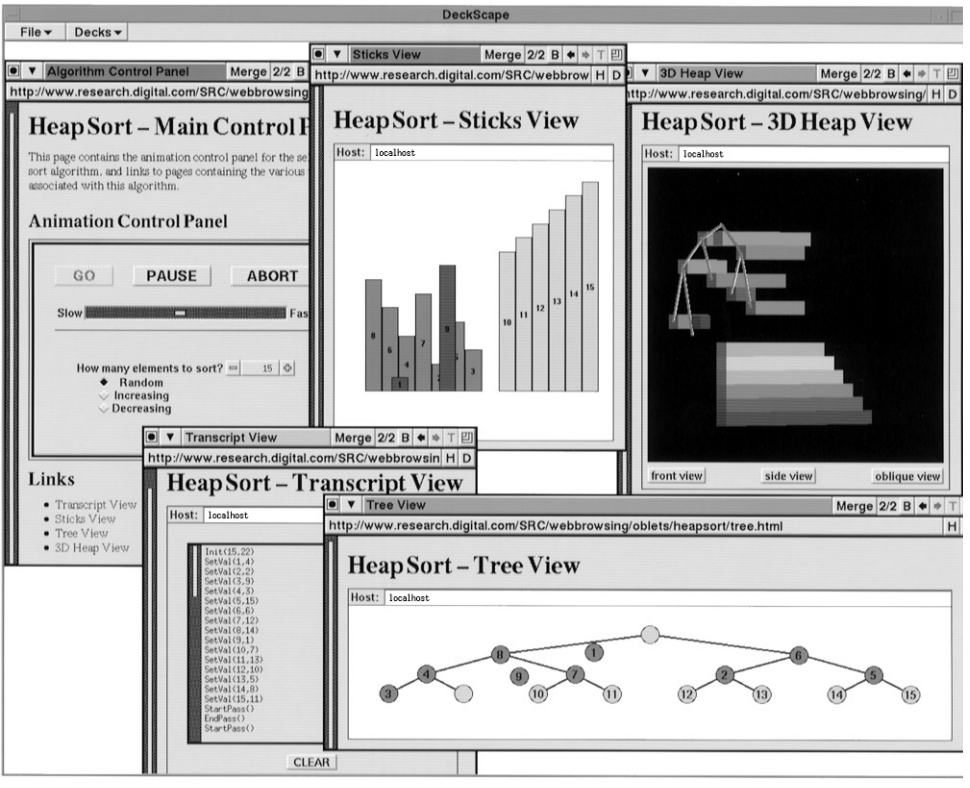


Figure 9. An animation of Heapsort. Heaps are binary trees in which each node is larger than its children. Heaps are commonly implemented as arrays. The 'Tree' view illustrates the tree nature of the heap and the 'Sticks' view shows its implementation as an array (see Figure 7). The '30 Heap' view combines these two representations into a single 3D view: When viewed from the front, we see the tree; when viewed from the side, we see the array and when viewed from an oblique angle, we see both representations at once.

Properties describe attributes of graphical objects, such as their color, size, location or orientation. A property consists of a *name* that determines what attribute is affected and a *value* that determines how it is affected. Property values are not simply scalar values, but rather functions that take a time and return a value. Thus, property values form the basis for animation.

The following is the code for an Oblet that displays a yellow sphere:

```
let oblet = {
  vbt ⇒ anim3D_new( ),
  run ⇒
  meth(self)
    let root = RootGO_NewStd(self.vbt);
    let ball = SphereGO_New([0, 0, 0], 0·5);
    root.add(ball);
    SurfaceGO_SetColor(ball,''yellow'');
  end
};
```

The procedure anim3D_new creates a widget capable of displaying 3D scenes. In this example, the widget is assigned to the Oblet's vbt field, but it can also be inserted into a FormsVBT form, as it was done in Figures 8 and 9.

The call to RootGO_NewStd(self.vbt) creates a root object that is associated with the 3D widget stored in self.vbt. The procedure also initializes the root object with some reasonable defaults: It attaches a camera object to the root object, adds two light objects to the scene and attaches callback objects that allow the user to interactively manipulate (e.g. rotate and scale) the scene. From here on, all objects that are added to the root object will be displayed by the 3D widget as seen through the camera object.

The call to SphereGO_New([0, 0, 0], 0·5) creates a sphere object that is centered at the origin of the coordinate system and has a radius of 0·5. The sphere object is then added to the root object, thereby becoming a part of the scene and appearing in the 3D widget.

Finally, the call to SurfaceGO_SetColor(ball, "yellow") changes the surface color property of the sphere object from its default value (white) to yellow. In this example, we passed a string constant to SurfaceGO_SetColor; in general, however, we can pass a property value object that maps times to colors. For example, assume we replace the call to SurfaceGO_SetColor with the following three lines:

```
let fractional = proc(r)r − float(floor(r)) end;
let pv = ColorProp_NewAsync(meth(self, time) color_hsv(fractional(r), 1, 1) end);
SurfaceGO_SetColor(ball,pv);
```

The first line defines a function that takes a real number and returns its fractional part; the second line creates a color property value pv whose value depends on the current time, and the third line changes the surface color property of the sphere object. The effect of these lines is that the sphere changes its color depending on the current time.

There are four types of property values: constant property values (such as the object created by passing the string 'yellow' to SurfaceGO_SetColor), asynchronous property values (objects whose value depends on the current time), dependent property values (objects whose value depends on other property values) and synchronous property values (objects that store a batch of animation requests and process them once a trigger object is signaled).

It is worth noting that most of the parameters that define this scene—the center and radius of the sphere, the location and settings of the light sources and the cameras, the background color of the root object and so on—are represented as property values and can therefore be animated. This uniformity, together with the small number of basic concepts and the provision of sensible defaults wherever possible, all contribute to making Obliq-3D easy to use.

Figure 8 shows a 3D animation of Dijkstra's shortest path algorithm; Figure 9 shows various animations of heapsort, one of them being 3D. These 3D views were originally part of the Zeus3D system [3]. An earlier paper [19] gives the full Obliq-3D source code for a non-Oblet version of the heapsort 3D animation.

6. A Java-based Reimplementation

The JCAT system [8] is an implementation of CAT in Java. By replacing Obliq with Java, JCAT can be run on the major Web browsers. Java does not have Obliq's language-level support for applets on different machines to communicate with one another; however, its Remote Method Invocation (RMI) package provides library-level support for constructing remote objects.

By design, CAT and JCAT are virtually indistinguishable to the end user and very similar to the animation author.

To the end user, the only difference is the browser in which the pages are viewed. In CAT, users were limited to our in-house browsers; in JCAT, users can view animations using any Java-enabled browser[a].

From the animation author's perspective, the systems are quite similar as well. Both follow the 'interesting events' paradigm; both require the author to write an events file, an algorithm and some views; both use a preprocessor to convert the algorithm and views into applets and both use standard HTML tags to embed the applets into Web pages. The main difference is that the algorithm and the views are written in Java rather than Obliq. Also, JCAT currently supports only 2D animation.

At the system level, the primary difference between CAT and JCAT is that Java does not have language-level support for distributed computation. However, the Java Developer's Kit (JDK) 1.1 contains the Remote Method Invocation (RMI) package, an object-oriented version of remote procedure calls, which allows applets running on different machines to communicate. The process of creating remote Java objects is more tedious than in Obliq; however, the JCAT preprocessor automatically generates all the code related to remote objects, thereby hiding the tedium from the animation author.

The remainder of this section expands on the differences between CAT and JCAT from the animation author's point of view.

6.1. The Interesting Events

In JCAT, interesting events are specified as a Java interface. Here is the Java interface that specifies the interesting events for binpacking algorithms:

```
public interface Binpacking {
    void setup(int numBins, int numBlocks);
    void newBlock(double wt);
    void probe(int bin);
    void pack( );
}
```

In the corresponding CAT event definition file (shown in Section 3.1), each parameter of the interesting events was annotated with a procedure for converting its value into

[a] The collaborative features of JCAT are available only if the browser supports Java RMI. At the time of this writing, RMI is supported only by Sun's HotJava browser; Netscape and Microsoft have announced that RMI-enabled versions of their browsers will be available by the summer of 1997.

a string. In JCAT, this is not necessary, because Java provides a standard mechanism for converting values of any type into strings.

JCATalyst, the JCAT preprocessor, takes this interface as its input and generates classes that implement the mechanisms that make the location of the algorithm and view applets transparent to the animation author. It also generates a class that implements a transcript view applet.

6.2. The Algorithm

An algorithm (in this case, FirstFitAlg) is a subclass of the abstract algorithm class generated by JCATalyst (in this case, BinpackingAlg), which in turn is a subclass of Applet. The following code shows the first-fit binpacking algorithm (the Obliq version was given in Section 3.2). For the sake of simplicity, we have omitted the code for laying out the input control panel's user interface.

```
import java.awt.*;

public class FirstFitAlg extends BinpackingAlg{
  TextField numBinsField = new TextField(''10'');
  TextField numBlocksField = new TextField(''20'');

  public void init( ) {
    super.init( );
    //. . . LayoutManager-related code omitted
    add(new Label(''Number of bins: ''));
    add(numBinsField);
    add(new Label(''Number of blocks: ''));
    add(numBlocksField);
  }
  public void algorithm(BinpackingDisp disp) {
    int numBins = Integer.parseInt(numBinsField.getText( ));
    int numBlocks = Integer.parseInt(numBlocksField.getText( ));
    disp.setup(numBins, numBlocks);
    double totals [ ] = new double [numBins];
    for (int block = 0; block < numBlocks; block++ ) {
      double amt = 0·2 + 0·7 * Math.random( );
      disp.newBlock(amt);
      int bin;
      for (bin = 0; bin < numBins; bin++ ) {
        disp.probe(bin);
        if (totals [bin] + amt ⩽ 1·0) break;
      }
      if (bin = = numBins) break;
      totals [bin] += amt;
      disp.pack( );
    }
  }
}
```

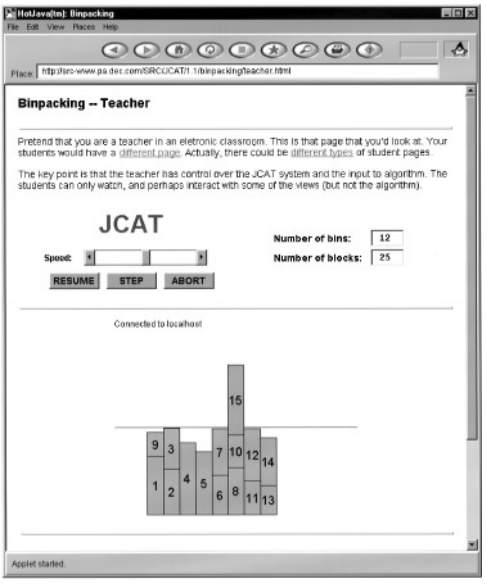
The init method, common to all applets, is called by the Web browser once the applet has been loaded. In this case, init configures the input control panel to contain type-in fields for the number of bins and the number of blocks.

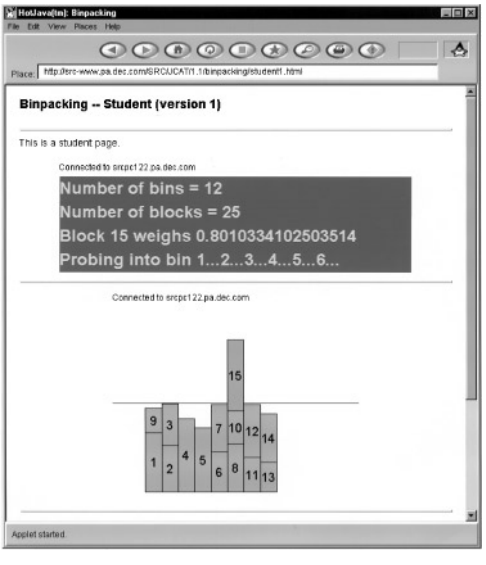
The disp parameter of the algorithm method contains the animation control object that notifies local and remote views of interesting events.

6.3. A view

A view in JCAT is an applet with additional methods corresponding to the interesting events. The 'Probes' view shown in Figure 10 is implemented by a class ProbingView, which is a subclass of BinpackingView, a class generated by JCATalyst. BinpackingView in turn is a subclass of Applet and an implementor of the Binpacking interface. The code of ProbingView is as follows:

```
import gp.*;
import java.awt.*;

public class ProbingView extends BinPackingView {
  GP        gp = new GP( );

  Vertex    currVertex;
  double    currWt;
  int       lastProbe;
  int       currId;
  double    totals[ ];

  public void init( ) {
      super.init( );
      add(gp);
  }
  public void setup(int numBins, int numBlocks) {
      currId = 1;
      totals = new double[numBins]; // all elems are 0 by default
      gp.clear( );
      gp.setWorld( − 2·0, numBins + 1·0, 2·0, 0·0);
      Vertex v0 = new Vertex(gp);
      v0.setPosition(− 10·0, 1·0);
      Vertex v1 = new Vertex(gp);
      v1.setPosition(numBins + 10·0, 1·0);
      Edge e = new Edge(v0, v1);
      e.setWidth(0·01);
      gp.redisplay( );
  }
  public void newBlock(double wt) {
      Vertex v = new Vertex(gp);
      v.setPosition( − 1·0, wt/2·0);
      v.setSize (1.0, wt);
      v.setColor(Color.lightGray);
      v.setBorder(0.01);
```





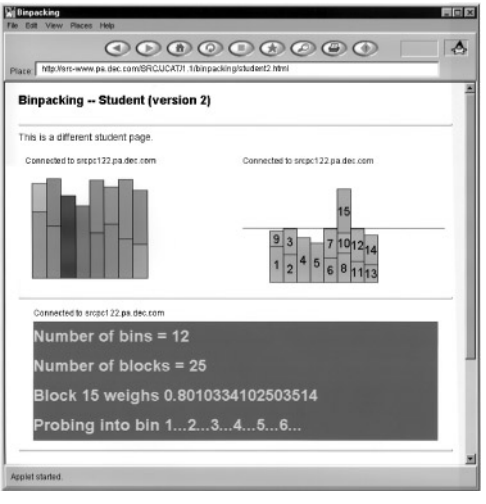
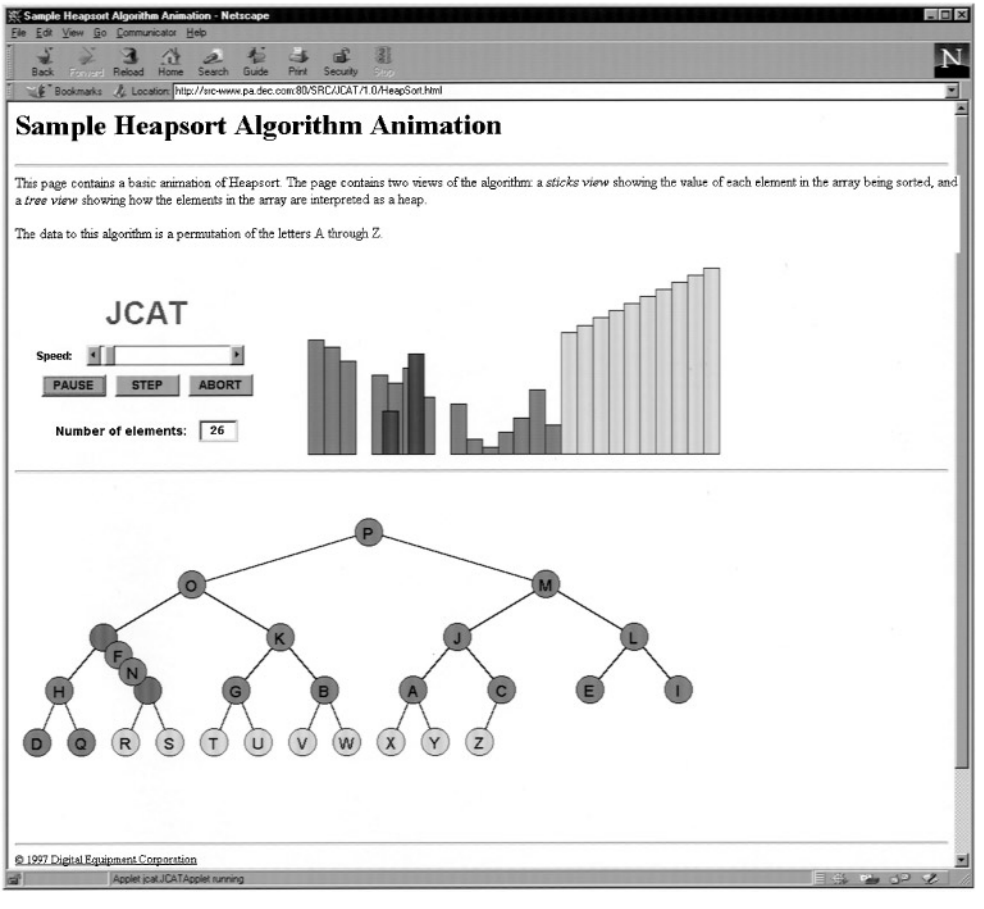


Figure 10. This figure shows a snapshot of a JCAT session involving an instructor and two students. The upper screen dump shows the instructor's Web browser. It contains three applets: the animation control panel, an algorithm-specific control panel and a 'Probes' view. The two lower screen dumps show the browsers of two students. The student on the left is visiting a page with two applets: a 'Glossary' view and the 'Probes' view. The student on the right is visiting a page with three applets: a 'Packing' view, the 'Probes' view and the 'Glossary' view

Figure 11. This animation shows two different views of heapsort: a view that shows the conceptual structure of the heap (bottom) and a view that shows the implementation of the heap as an array (middle right). The screen dump was captured while the 4th (the key F) and the 9th element (the key N) in the array were exchanging. The exchange is shown with smooth animations in both views. Colors are used consistently in both views to distinguish the heap elements from those elements of the array that are already sorted and no longer part of the heap

```
        v.setLabel(currId + " ");
        v.setLabelColor(Color.black);
        gp.redisplay( );
        currVertex = v;
        currWt = wt;
        currId ++ ;
    }
    public void probe(int bin) {
        currVertex.move(0.5 + bin, totals[bin] + currWt/2.0);
        gp.animate(1·0);
        lastProbe = bin;
    }
    public void pack( ) {
        totals[lastProbe] += currWt;
        currVertex.setColor(Color.pink);
        gp.redisplay( );
    }
}
```



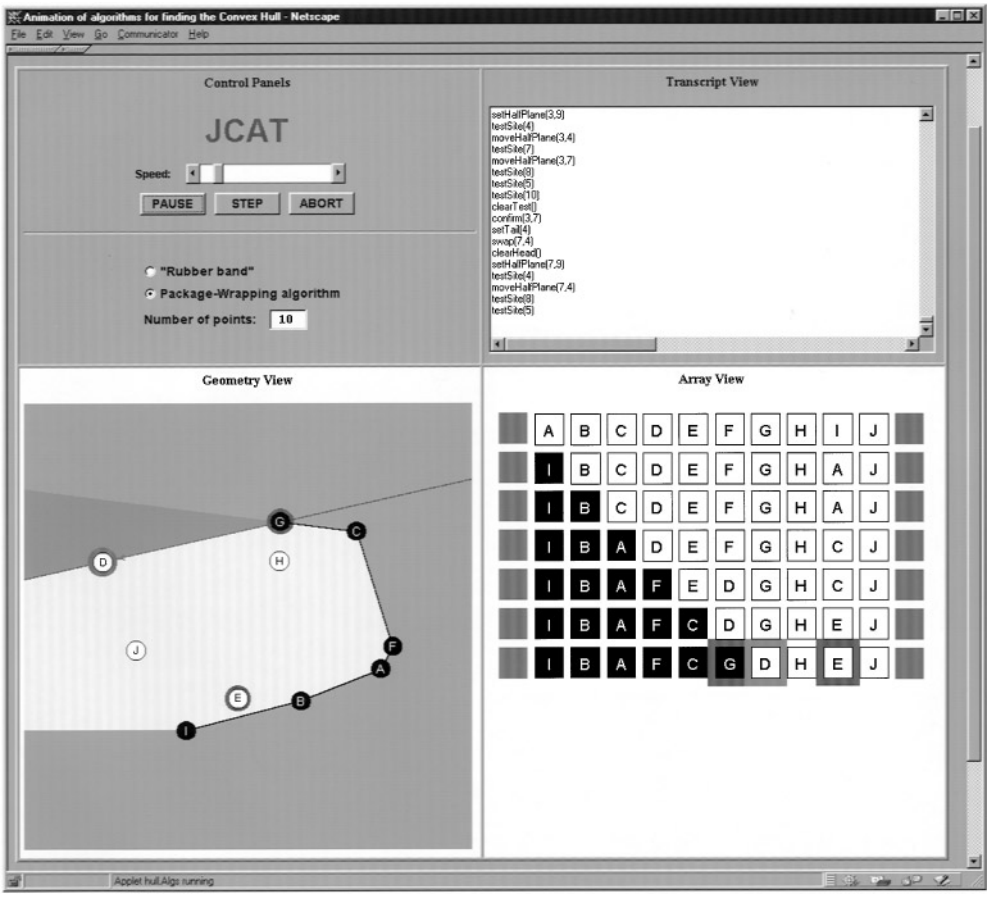


Figure 12. An animation of the package-wrapping algorithm for computing the convex hull of a set of points in the plane. The lower left view shows how the segments of the hull are wrapped around the points in the plane; the lower right view shows the underlying main data structure of the algorithm

As mentioned, views implement the methods that are defined in the interesting events interface. This view overrides the **setup**, **newBlock**, **probe** and **pack** methods defined in the **Binpacking** interface. The body of each method is responsible for updating the screen in a way that is meaningful for the view. For example, **newBlock** creates a gray rectangular vertex representing a block and also saves the weight of the block; **probe** smoothly slides the vertex representing the block being processed from its current position to the bin being probed and also saves the bin, and **pack** changes the color of the rectangle to indicate that the current block has been packed and also updates an array maintaining the total weight of the blocks in each bin.

The package **gp** is a Java-based version of the 2D animation library described in Section 5.1. The current version of Java does not support 3D graphics; we intend to incorporate 3D animations once this changes.

Figures 11–13 show snapshots of some of the JCAT animations we have built. You can experience the actual animations by visiting the JCAT home page at **http://www.research.digital.com/SRC/JCAT**.

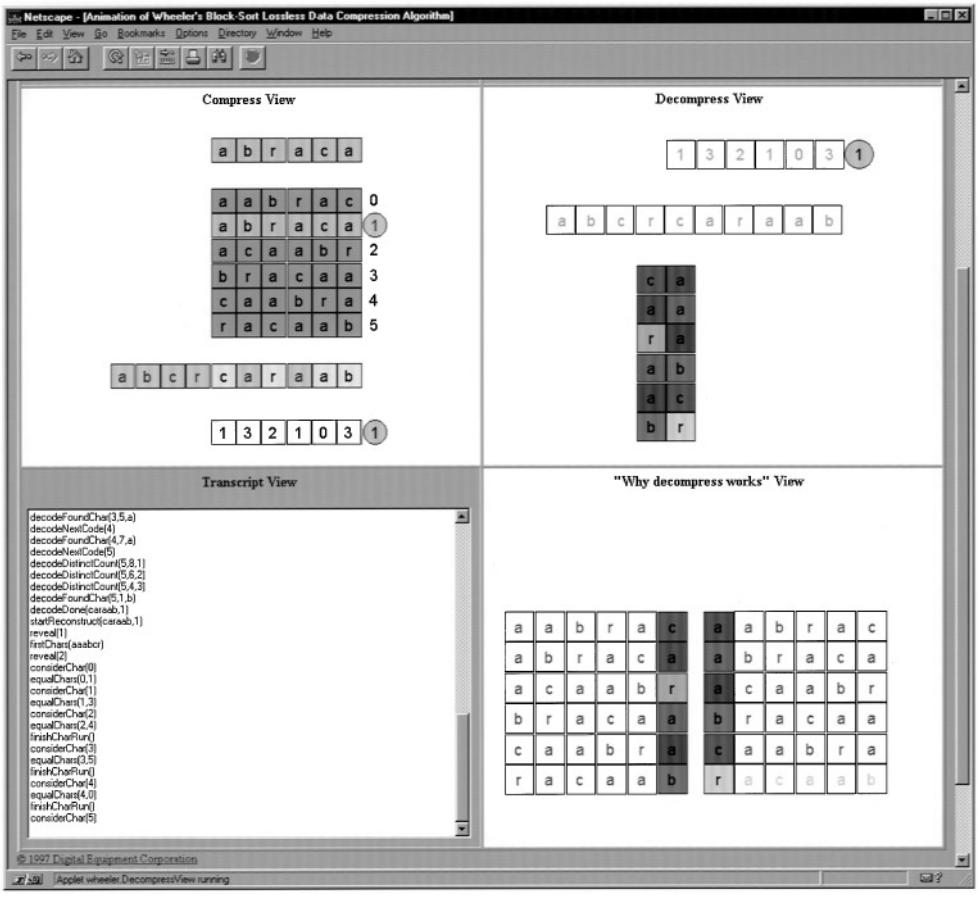


Figure 13. An animation of David Wheeler's block-sorting lossless data compression algorithm [11]. The upper left view illustrates the compression phase; the upper right view illustrates the decompression phase; the lower right view provides further insight into why the decompression phase works

7. Conclusions

This paper has described CAT, a Web-based algorithm animation system.

CAT improves on classical algorithm animation systems (e.g. BALSA, TANGO, Zeus) by combining the power of Web pages for publishing passive multimedia information with interactive algorithm animations.

CAT improves on previous Web-based algorithm animations (e.g. Java applets) in that the same running algorithm can be viewed on multiple machines. This feature makes CAT particularly well-suited for an electronic classroom setting. Moreover, we provide the same amount of high-level support for producing algorithm animations as is found in bona fide algorithm animation systems. CAT improves on Gloor's Hypercard-based electronic textbook for the same reasons.

Finally, CAT improves on existing electronic classroom software by supporting a higher-level notion of collaboration than the standard technique of multiplexing X windows.

Although we have presented CAT in the context of algorithm animation, we believe that the system is suitable to other domains amenable to computer animation and simulation.

References

1. G. Avrahami, K. P. Brooks & M. H. Brown (1989) A two-view approach to constructing user interfaces. *Computer Graphics* **23,** 137–146.
2. J. E. Baker, I. F. Cruz, G. Liotta & R. Tamassia (1996) Algorithm animation over the World Wide Web. In *International Workshop on Advanced Visual Interfaces (AVI '96)*, pp. 203–222.
3. M. H. Brown & M. A. Najork (1993) Algorithm animation using 3D interactive graphics. *ACM Symposium on User Interface Software and Technology*, pp. 93–100.
4. M. H. Brown & M. A. Najork (1995) Algorithm animation at SRC. http://www.research.digital.com/SRC/zeus/.
5. M. H. Brown & M. A. Najork (1995) An MPEG movie of a 3D animation of heapsort. http://www.research.digital.com/SRC/zeus/heapsort3D.2.mpg.
6. M. H. Brown & M. A. Najork (1996) Distributed active objects. *Computer Networks and ISDN Systems* **28,** 1037–1052.
7. M. H. Brown & M. A. Najork (1997) Distributed applets (video and extended abstract). In: *ACM Conference on Human Factors in Computing Systems* (*CHI'97*), *Extended Abstracts*, pp. 204–205.
8. M. H. Brown, M. Najork & R. Raisamo (1997) A Java-based implementation of collaborative active textbooks. To appear in *IEEE Symposium on Visual Languages*, September 1997.
9. M. H. Brown & R. A. Shillner (1995) DeckScape: an experimental web browser. *Computer Networks and ISDN Systems* **27,** 1097–1104.
10. M. H. Brown & R. Sedgewick (1984) A system for algorithm animation. *Computer Graphics* **18,** 177–186.
11. M. Burrows & D. J. Wheeler (1994) A block-sorting lossless data compression algorithm. SRC Research Report 124, Digital Equipment Corporation, Systems Research Center.
12. L. Cardelli (1995) A language with distributed scope. *Computing Systems* **8,** 27–59.
13. J. D. DeTreville (1993) The GraphVBT interface for programming algorithm animations. *IEEE Symposium on Visual Languages*, pp. 26–31.
14. J. Erickson (1997) Computational geometry interactive software. http://www.cs.duke.edu/~jeffe/compgeom/demos.html.
15. P. A. Gloor, S. Dynes & I. Lee (1993) *Animated Algorithms—A Hypermedia Learning Environment for Introduction to Algorithms*. Cambridge, MA, MIT Press.
16. A. Hausner (1996) Algorithm animation. http://www.cs.princeton.edu/~ah/alg_anim.
17. B. Ibrahim (1994) World wide algorithm animation. *Computer Networks and ISDN Systems* **27,** 255–265.
18. A. Lawrence, A. Badre & J. Stasko (1994) Empirically evaluating the use of animations to teach algorithms. *IEEE Symposium on Visual Languages*, pp. 48–54.
19. M. A. Najork & M. H. Brown (1995) Obliq–3D: a high-level, fast-turnaround 3D animation system, *IEEE Transactions on Visualization and Computer Graphics* **1,** 175–193.
20. J. T. Stasko (1994) Interact with XTango animations. http://www.cc.gatech.edu/stasko/cgi-bin/xtangoanim.
21. J. T. Stasko (1990) TANGO: a framework and system for algorithm animation. *IEEE Computer* **23**, 27–39.
22. World-Wide Web Consortium (1997) Inserting objects into HTML. http://www3.org/pub/WWW/TR/WD-object.