May 31, 1996

SRC Research Report **142**

Collaborative Active Textbooks: A Web-Based Algorithm Animation System for an Electronic Classroom

Marc H. Brown and Marc A. Najork

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Robert W. Taylor, Director

Collaborative Active Textbooks: A Web-Based Algorithm Animation System for an Electronic Classroom

Marc H. Brown and Marc A. Najork

May 31, 1996

Publication History

A slightly shorter version of this report will appear in the *Proceedings of the IEEE Symposium on Visual Languages (VL'96)* to be held September 3–6, 1996 in Boulder, Colorado.

Abstract

This report 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. This strategy is an improvement over the electronic classroom systems we are aware of, which simply display the same X window on multiple machines. We believe our framework generalizes to electronic textbooks in arbitrary domains.

1 Background

This report 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.

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 [13].

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 "active objects," that is, regions of the page that are drawn by programs dynamically loaded through the Web [17]. 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 [5].

The contents of our pages is similar in spirit to Gloor's CD-ROM [10], which complements the Corman, Leiserson, and Rivest *Introduction to Algorithms* textbook. 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.

We are not the first to display "animated algorithms" on Web pages: there are numerous Java applets showing algorithms in action. For example, a nice collection of sorting algorithms has been compiled by Harrison [11]. However, these animations are written in an ad hoc fashion, without the support of an algorithm animation system. As such, they lack a variety of features, which we shall discuss later.

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 [14]. Finally, there are a variety of MPEG and QuickTime movies of animated algorithms on the Web, such as our animation of Heapsort [3]; 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 [12]. 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.

Our active objects resemble Java applets in that the code for algorithms and views is downloaded over the network, and then executed by and displayed in the Web browser [4]. The innovation of CAT is that it displays multiple, simultaneously animated views of an algorithm. The views either may appear on a single page, or on separate pages. Moreover, multiple users viewing the same page will see the same animation; those views that allow customization can be customized differently by each user. Thus, when 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've 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 multiplexor, such as XMX and 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.

Another difference between CAT and Java-based applets is the level of programmer support for programming algorithm animations. Java-based algorithm animations are constructed in an ad hoc fashion. We follow the programming model pioneered by BALSA [7], 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, tailored for algorithm animation. Finally, CAT provides an algorithm-independent control panel for starting, pausing, and stopping the animation, and controlling its speed.

The rest of this report 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. Next, we describe how CAT is implemented. 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.

Figs. 1 and 2 show WebScape, a Mosaic-style Web browser. The instructor's screen, Fig. 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). Fig. 2 shows a student's screen at the same time that Fig. 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, currently displayed 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 Fig. 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 Fig. 2 contains a link to the algorithm source code. A student can follow this link at any time, and following the link will not interrupt the animation (of course the animation won't be visible until the student returns to this page).

It's 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

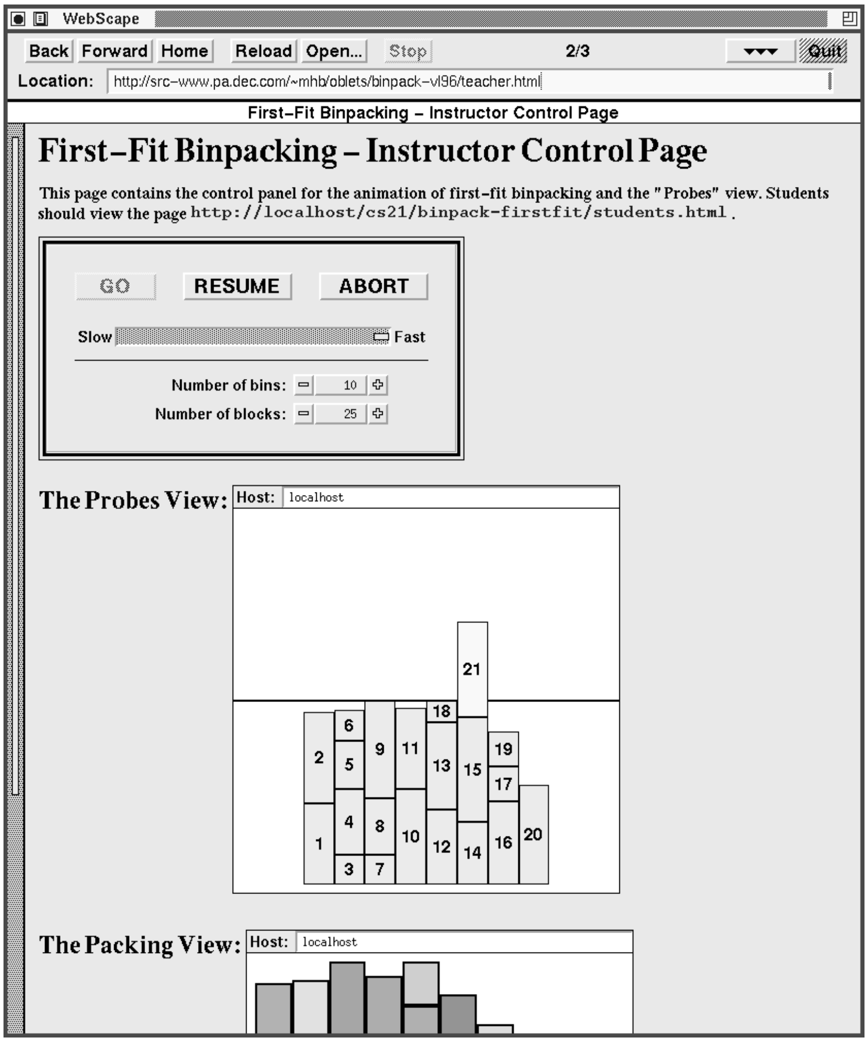


Fig. 1. An instructor's workstation, `ash.pa.dec.com`, during a lecture on first-fit binpacking.

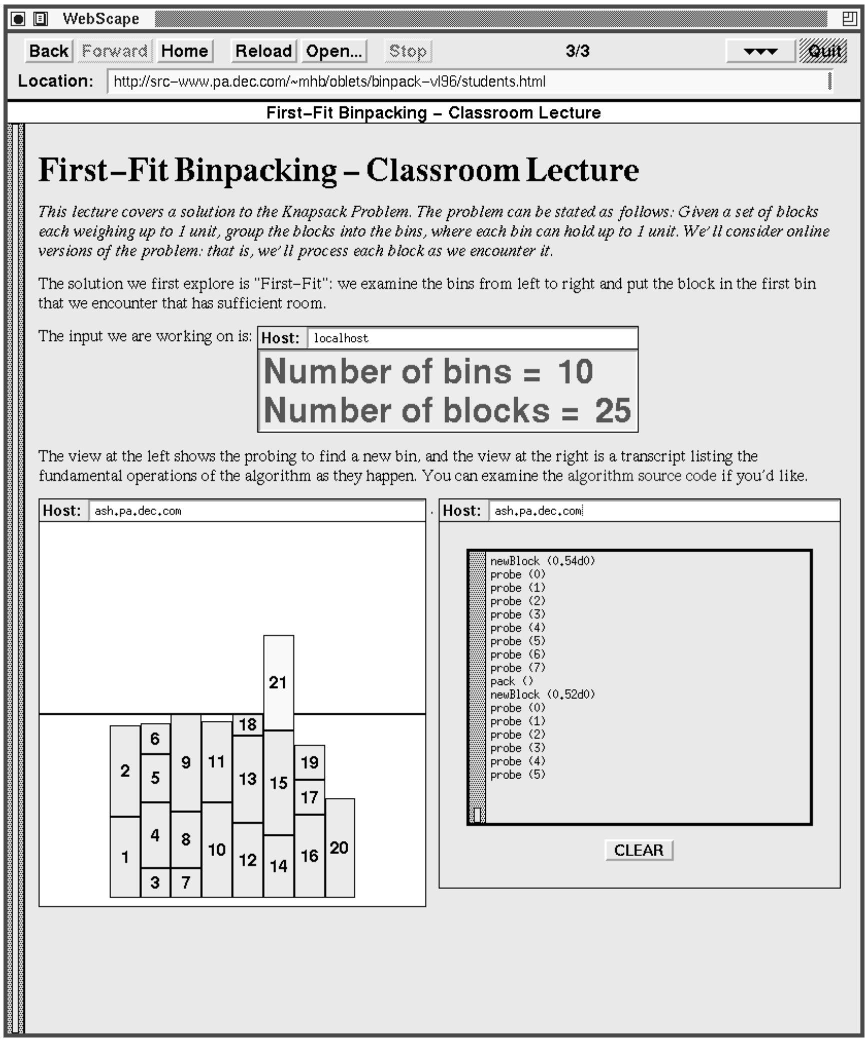


Fig. 2. A student's workstation when the screen in Fig. 1 was captured. Note that the student specified the host `ash.pa.dec.com` in order to connect with the running algorithm on that machine.

demand on the network, because the only network traffic are the parameters to each event.

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 "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.

Fig. 3 shows a different set of Web pages for first-fit binpacking. These pages are displayed using DeckScape [6], 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 Figs. 1 and 2; however, the active objects are *exactly* the same.

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 [8].

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.

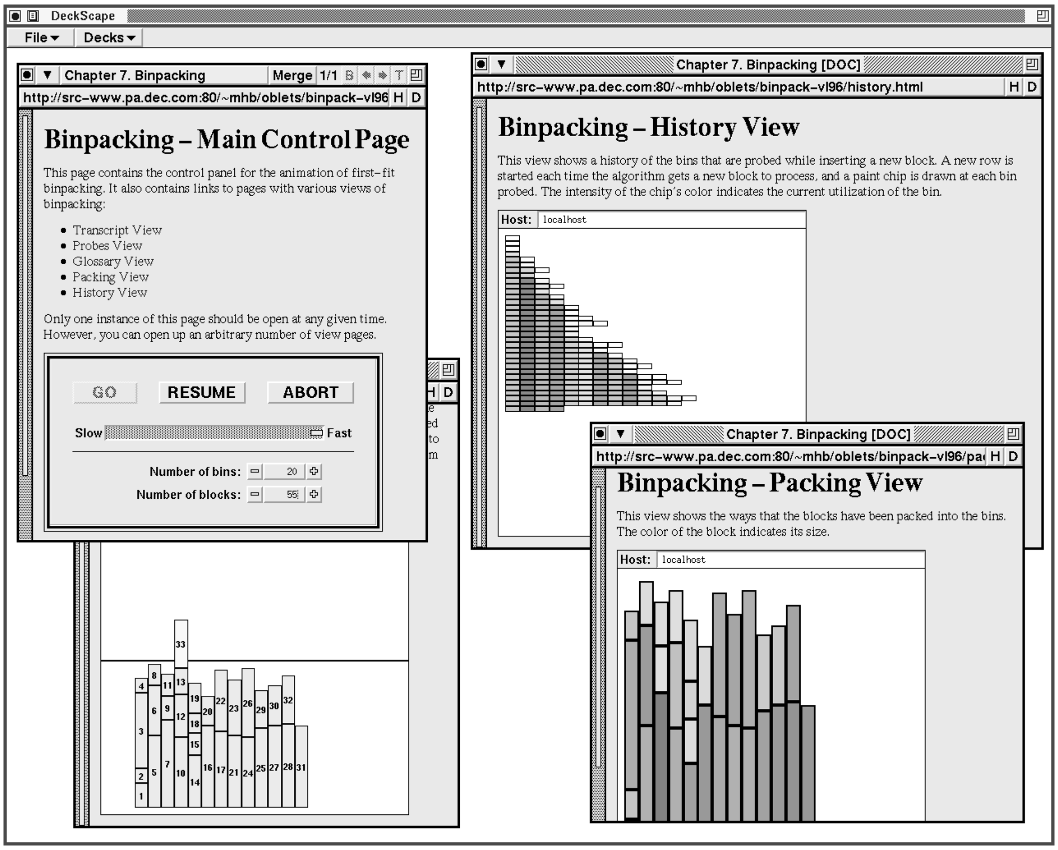


Fig. 3. A user interacting with the chapter on binpacking in an "electronic textbook".

3.1 The Events

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

```
setup    (nBins[fmt_int], nBlocks[fmt_int])
newBlock (wt[fmt_real])
probe    (b[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.

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 `b` 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'll 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, z)
     let numBins   = form_getInt(self.vbt, "bins");
     let numBlocks = form_getInt(self.vbt, "blocks");
     z.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;
       z.newBlock(amt);
       var bin = 0;
       loop
         z.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;
       z.pack();
     end;
   end
};
```

The `go` method takes two parameters: `self` refers to the object in which the method is contained, and `z` is the animation event manager 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 [9]. 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 [15].

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,
  totals       => ok,

  setup =>
    meth (self, nBins, nBlocks)
      self.totals := array_new(nBins, 0.0);
      graph_clear(self.vbt);
      graph_setWorld(self.vbt, -2.0, float(nBins), 2.0, 0.0);
      let v0 = graph_newVertex(self.vbt);
      graph_setVertexSize(v0, 0.0, 0.0);
      graph_moveVertex(v0, -10.0, 1.0, false);
      let v1 = graph_newVertex(self.vbt);
      graph_setVertexSize(v1, 0.0, 0.0);
      graph_moveVertex(v1, float(nBins)+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_setVertexSize(v, 1.0, wt);
      graph_setVertexColor(v, "VeryLightGray");
      graph_setVertexBorder(v, 0.01);
      graph_moveVertex(v, -1.0, wt/2.0, false);
      graph_redisplay(self.vbt);
      self.currVertex := v;
      self.currWt := wt;
    end,

  probe =>
    meth (self, b)
      let xpos = 0.5 + float(b);
      let ypos = self.totals[b] + (self.currWt/2.0);
      graph_moveVertex(self.currVertex, xpos, ypos, true);
      graph_animate(self.vbt, 0.0, 1.0);
      self.lastProbe := b;
    end,

  pack =>
    meth (self)
      let b = self.lastProbe;
      self.totals[b] := self.totals[b] + self.currWt;
      graph_setVertexColor(self.currVertex, "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, that is, 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 a handle to the vertex and to the block's weight (so other events can reposition and recolor the vertex).

The `probe` method moves the vertex representing the new block to the top of bin `b`. 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

CAT uses the file of interesting events to generate a number of auxiliary objects, such as the animation control object we mentioned before.

In particular, CAT generates "proxy objects" for each algorithm 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," CAT creates 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 `insert`, an HTML tag proposed by the World Wide Web Consortium for inserting multimedia objects into HTML pages [16]. The markup for putting the proxy object stored at URL "BPalg.obl" into a document is:

```
<insert code="BPalg.obl" type="application/x-oblet">
</insert>
```

The insert 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

At the heart of CAT is a family of Web browsers that support active objects 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 active objects that collaborate with one another. We call our active objects Oblets (**Ob**liq app**lets**).

The proxy objects we mentioned earlier for the algorithm and for the views are actually Oblets. CAT uses the event definition file and the user-supplied alg and view object files to generate files containing these Oblets. CAT also generates a "Transcript" view from the event definition file.

In addition to the algorithm and view Oblets, there is an animation control object (the z 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 each interesting event. Here is the body of a method 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,
```

The method iterates through the array of view Oblets, and forks a thread per Oblet. These threads call the foo method on 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` on 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.

Figs. 4 and 5 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 (Fig. 4). The other Oblet, the client, displays the name of the chosen color inside a rectangle of that color (Fig. 5).

The screen dump in Fig. 4 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`s 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 Fig. 5 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 points, 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

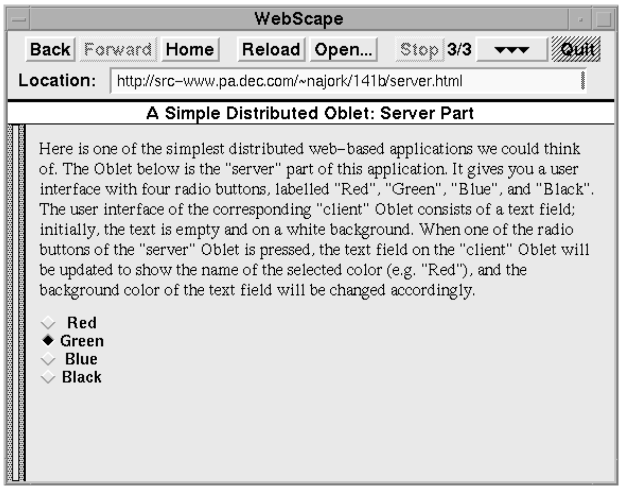
```
server.client := self;
```

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 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.

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
};
```

Fig. 4. The "server" Oblet; the corresponding "client" Oblet appears in Fig. 5.

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
};
```

Fig. 5. The "client" Oblet; the corresponding "server" Oblet appears in Fig. 4.

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, `z`. The second part is the algorithm code supplied by the author, that is, 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 z     = {...};
let alg   = {...};
let oblet = {...};
```

For didactic reasons, we will first look at the Oblet, and then the animation control object.

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 `z` 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)))))))
```

And here is the code for the algorithm's `oblet` object:

```
let oblet = {
  goThread=> ok,

  vbt => form_fromURL(BaseURL & "controlPanel.fv"),

  run =>
    meth(self)

      let goCallback =
        proc(fv)
          self.goThread := thread_fork(proc()
            z.paused := false;
            form_putReactivity(fv, "go",    "dormant");
            form_putReactivity(fv, "pause", "active");
            form_putReactivity(fv, "abort", "active");
            try alg.go(z) 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, 0);
        end;

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

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

    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, `z`, generated by CAT.

```
let z = {
  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, nBins, nBlocks)
      let thrds =
        foreach v in self.views map
          fork(proc() v.view.setup(nBins, nBlocks) end, 0)
        end;
      foreach t in thrds do join(t) end;
      if thread_testAlert() then raise(thread_alerted) end;
      lock self.mu do
        if self.paused then
          thread_alertWait(self.mu, self.cond)
        end
      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 `z` resides. The `registerView` and `unregisterView` methods maintain the `views` array.

Recall that the user-defined `alg` object calls `z.setup` for communicating information to the views. The setup method of `z` 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.

The next line handles the "ABORT" functionality: As we saw, pressing the "ABORT" button causes the "alert" flag to be set. `setup` 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.

Finally, we handle the "PAUSE" and "RESUME" functionality: As we saw, pressing the "PAUSE" button causes the `paused` flag to be set to true. `setup` 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.

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, that is, 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,
  z    => ok,
  host => "** unconnected **",
  vbt  => form_fromURL(BaseURL & "viewframe.fv"),

  run =>
    meth (self)

      let newHost =
        proc (hostName)
          let old = self.z;
          try
            self.z := net_import ("BP", hostName);
            self.z.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 `z`, which will be bound to an animation control object, and a field `host`, the name of the machine on which `z` 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). 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 ="localhost")))
    (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.

5 Conclusion

This report 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 level 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 technique is applicable to other domains amenable to computer animation and simulation.

References

[1] Gideon Avrahami, Kenneth P. Brooks, and Marc H. Brown. A Two-View Approach To Constructing User Interfaces. *Computer Graphics*, **23**(3):137–146, July 1989.

[2] Marc H. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing, *IEEE Workshop on Visual Languages*, pp. 4–9, Oct. 1991.

[3] Marc H. Brown and Marc A. Najork. "An MPEG Movie of a 3D Animation of Heapsort."
`http://www.research.digital.com/SRC/zeus/heapsort3D.2.mpg`

[4] Marc H. Brown and Marc A. Najork. Distributed Active Oblets. *Computer Networks and ISDN Systems*, **28** (1996) 1037–1052.

[5] Marc H. Brown and Marc A. Najork. Distributed Active Oblets (video). Research Report #141b, DEC Systems Research Center, Palo Alto, CA, May 1996.

[6] Marc H. Brown and Robert A. Shillner. DeckScape: An Experimental Web Browser. *Computer Networks and ISDN Systems*, **27**(1995) 1097–1104.

[7] Marc H. Brown and Robert Sedgewick, A System for Algorithm Animation, *Computer Graphics*, **18**(3):177–186, July 1984.

[8] Luca Cardelli. A Language with Distributed Scope. *Computing Systems*, **8**(1):27–59, Jan. 1995.

[9] John D. DeTreville. The GraphVBT Interface for Programming Algorithm Animations, *IEEE Symp. on Visual Languages*, pp. 26–31, Aug. 1993.

[10] Peter A. Gloor, Scott Dynes, and Irene Lee. *Animated Algorithms – A Hypermedia Learning Environment for Introduction to Algorithms.* MIT Press, Cambridge, MA, 1993.

[11] Jason Harrison. "Sorting Algorithms Demo."
`http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html`

[12] Bertrand Ibrahim. "World Wide Algorithm Animation."
`http://www.oac.uci.edu/indiv/franklin/doc/ibrahim/paper.html`

[13] Andrea Lawrence, Albert Badre, and John Stasko. Empirically Evaluating the Use of Animations to Teach Algorithms. *IEEE Symp. on Visual Languages*, pp. 48–54, Oct. 1994.

[14] John T. Stakso. "Interact with XTango Animations."
http://www.cc.gatech.edu/stasko/cgi-bin/xtangoanim

[15] John T. Stasko, TANGO: A Framework and System for Algorithm Animation, *IEEE Computer*, 23(9):27–39, Sept. 1990.

[16] World-Wide Web Consortium. "HTML3 Linking and Embedding Model."
http://www.w3.org/hypertext/WWW/TR/WD-insert-951221.html

[17] World-Wide Web Consortium. "Consortium Announces Active Object Agreement."
http://www.w3.org/pub/WWW/MarkUp/19951201_Insert_Press