Web-based Algorithm Animation

Marc Najork
Compaq Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301, USA
marc.najork@compaq.com

ABSTRACT

This paper describes JCAT, a web-based algorithm animation system. JCAT combines the expressive power of web pages for publishing passive multimedia content with a full-fledged interactive algorithm animation system that includes a rich set of libraries for creating 2D and 3D animations. The paper describes in detail how an algorithm animation is authored, and it presents a sample of existing animations.

1. INTRODUCTION

Algorithm animation is concerned with illustrating the behavior of a program by visualizing the fundamental operations of the program as it runs. Such displays have proven to be quite useful for education and for research in the design and analysis of algorithms.

The 1970's saw a number of short films depicting the operations of algorithms at an abstract level [12, 3], culminating in Baecker and Sherman's seminal movie "Sorting out Sorting" [1]. The film was computer-generated, but at the time, it was not possible to do this in real time. Although "Sorting out Sorting" runs for only 30 minutes, it took over three years to make.

Baecker's work coupled with advances in graphics hardware inspired BALSA, the first real-time, interactive algorithm animation system [4]. BALSA introduced the *interesting event paradigm*, the concept of separating the algorithm from its animation, and instrumenting the algorithm with procedure calls indicating *interesting events* (e.g. the swap of two elements in an array) that drive the animation. BALSA too was constrained by the computational power and the graphics hardware available at the time; its visualizations were monochrome, and interesting events triggered abrupt changes in the display.

As hardware advanced, so did the capabilities of algorithm animation systems. Animus [10] pioneered smooth animations; TANGO [13] introduced the *path-transition paradigm*, an elegant high-level framework for specifying such animations. Zeus [6] added color graphics and computer-generated sounds to the repertoire available to the animator. Finally, Polka-3D [14] and Zeus3D [7] explored the use of 3D graphics for algorithm animation.

The advent of the web triggered the development of a series of web-based algorithm animation systems. The Mocha system [2] used a client-server approach: Users point a web browser at a web page containing a Java applet. The applet starts an algorithm on a remote server; the algorithm transmits interesting events via TCP to the applet, and the applet generates the appropriate animations. The CAT system [8], on the other hand, ran both the algorithm and the animations as applets on the user's web browser; these applets were written in a non-standard scripting language. The follow-on system JCAT [9] replaced the non-standard scripting language with Java. The most recent version of JCAT offers the full set of tools provided by our previous algorithm animation systems to the animator, including high-level 2D and 3D animation libraries and support for algorithm auralization.

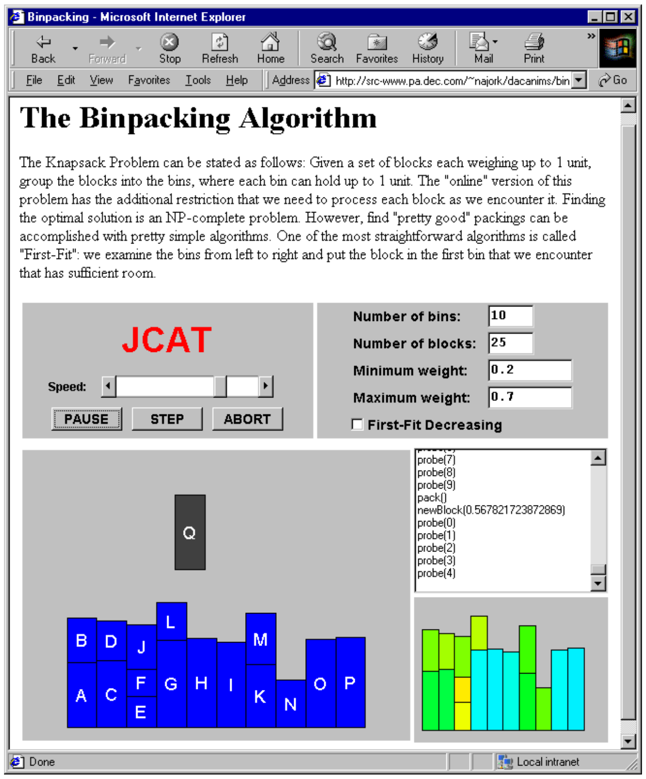
The interested reader is referred to [15] for a comprehensive coverage of algorithm animation systems.

2. WRITING A JCAT ANIMATION

This section describes how to write an animation using the JCAT system. It does so by fleshing out an animation of the first-fit binpacking algorithm, the "hello world" example of algorithm animation. The binpacking problem is as follows: Given a set of blocks each weighing up to 1 unit, group the blocks into the fewest bins possible, where each bin can hold up to 1 unit. The "online" version of this problem has the additional restriction that each block must be processed as it is encountered in the input stream. Although the optimal solution to this problem is NP-complete, a "pretty good" packing can be accomplished by examining the bins from left to right and putting the block in the first bin encountered that has sufficient room. This algorithm is called first-fit binpacking.

Figure 1 shows a JCAT animation of binpacking. The top-left applet is the control panel. It allows the user to start and stop the algorithm, advance the algorithm step-by-step, and adjust the speed of the animation. The control panel is algorithm-independent; this applet is used to control all algorithms in the JCAT system.

The applet at the top-right is an algorithm input panel that is used for specifying input to the algorithm. This applet is specific to each algorithm. The algorithm input panel used for binpacking algorithms allows users to specify the number of bins available for packing, the number of blocks to pack, and the range of possible weights of each block.



The Binpacking Algorithm

The Knapsack Problem can be stated as follows: Given a set of blocks each weighing up to 1 unit, group the blocks into the bins, where each bin can hold up to 1 unit. The "online" version of this problem has the additional restriction that we need to process each block as we encounter it. Finding the optimal solution is an NP-complete problem. However, find "pretty good" packings can be accomplished with pretty simple algorithms. One of the most straightforward algorithms is called "First-Fit": we examine the bins from left to right and put the block in the first bin that we encounter that has sufficient room.

Number of bins: 10
Number of blocks: 25
Minimum weight: 0.2
Maximum weight: 0.7
☐ First-Fit Decreasing

```
probe(7)
probe(8)
probe(9)
pack()
newBlock(0.567821723872869)
probe(0)
probe(1)
probe(2)
probe(3)
probe(4)
```

Figure 1: Animation of the binpacking algorithm

The three applets below the first two are *views*. The large applet on the left is the "probing view"; it shows each block as a vertical bar whose height reflects the weight of the block. As the algorithm examines the bins, the new block is graphically shown on the bin being examined. Once a bin is found with enough room for the new block, the color of the new block changes from gray to blue. The smaller applet at the lower-right is the "packing view"; it shows how the blocks have been arranged into the bins. Color is used to redundantly encode the weight of each block. The mid-right applet is a "transcript view"; it shows a textual log of the interesting events generated by the algorithm.

The framework for animating algorithms follows the model pioneered by BALSA [4]: Strategically important points of an algorithm are annotated with procedure calls that generate interesting events. These events are reported to an event dispatcher, which in turn forwards them to all registered views. Each view may respond to each interesting event by drawing appropriate images.

The task of preparing a JCAT animation consists of four parts: defining the interesting events for the algorithm; implementing the algorithm and annotating it with the events; implementing one or more views; and finally, creating Web pages that make use of the algorithm and views. The Web pages are prepared using HTML; the events, algorithm, and views are written in Java. The rest of this section shows how the binpacking animations seen before were implemented.

2.1 Defining the interesting events

The set of interesting events is specified as a Java interface. Here are the interesting events for the first-fit binpacking algorithm:

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

The `setup` event is called once when the algorithm starts, to communicate to the views how many blocks will be processed and how many bins are available. The `newBlock` event is called each time the algorithm encounters a new block, whose weight is specified as the parameter. The `probe` event is called each time the algorithm checks if the new block can be packed into the bin specified as the parameter. The `pack` event is called to indicate that the last bin probed is where the new block will be placed.

JCAT comes with a preprocessor called CATalyst that takes the interesting event interface and derives abstract classes for the algorithm and the views. The classes generated by CATalyst, along with the JCAT base classes, provide all the communication mechanisms between algorithms and views. CATalyst also generates algorithm-specific transcript views and code views (described below).

2.2 Annotating the Algorithm

The algorithm is implemented by an applet that appears as the algorithm input panel. It is a subclass of the abstract algorithm class generated by CATalyst, which is a subclass of JCAT's generic algorithm class, which in turn is a subclass of the standard Java applet class.

The following code shows the applet that implements first-fit binpacking. The animation author implements the algorithm by subclassing the algorithm class generated by CATalyst (in this case, `BinpackingAlg`) and overriding an abstract method called `algorithm` with the algorithm in question. The algorithm code is annotated with calls to interesting event methods (marked by comments); these methods are provided by `BinpackingAlg`. Each of these methods forwards interesting events to the control panel, which in turn forwards them to the views.

```
public class FirstFit extends BinpackingAlg {
  EntryField binFld, blockFld, minFld, maxFld;

  public void init() { ... }

  protected void algorithm() {
    int numBins = binFld.getInt();
    int numBlocks = blockFld.getInt();
    double min = minFld.getDouble();
    double max = maxFld.getDouble();
    setup(numBins, numBlocks); // interesting event
    double totals[] = new double[numBins];
    for (int b = 0; b < numBlocks; b++) {
      double wt = Math.random()*(max-min)+min;
      newBlock(wt);             // interesting event
      int bin;
      for (bin = 0; bin < numBins; bin++) {
        probe(bin);             // interesting event
        if (totals[bin] + wt <= 1.0) break;
      }
      if (bin == numBins) break;
      totals[bin] += wt;
      pack();                   // interesting event
    }
  }
}
```

Since `FirstFit` is a subclass of `java.applet.Applet`, it inherits various methods that are invoked when the applet has been loaded, started, stopped, and discarded. In this example, the `init` method (elided) creates the user interface elements of the algorithm input panel seen at the top-right of Figure 1.

2.3 Implementing a View

A view is a subclass of the abstract view class generated by CATalyst (e.g., `BinpackingView`), which in turn is a subclass of `java.applet.Applet`. The abstract view class generated by CATalyst defines an empty method for each interesting event. The animation author creates a new view by subclassing the abstract view class and overriding those methods for which animation effects are desired.

The actual code for the probing view shown in the previous screen images is as follows:

```
public class ProbingView extends BinpackingView {
  GP gp = new GP();
  Vertex v;
  double currWt;
  double totals[];
  char id;
  int lastProbe;

  public void init() {
    super.init();
    add(gp);
  }
  public void setup(int numBins, int numBlocks) {
    id = 'A';
    totals = new double[numBins];
    gp.clear();
    gp.setWorld(-2.0, numBins + 1.0, 2.0, 0.0);
    gp.redisplay();
  }
  public void newBlock(double wt) {
    v = new Vertex(gp);
    v.setSize(1.0, wt);
    v.setPosition(-1.0, wt / 2.0);
    v.setColor(Color.darkGray);
    v.setBorder(0.01);
    v.setLabelColor(Color.white);
    v.setLabel(Character.toString(id++));
    gp.redisplay();
    currWt = wt;
  }
  public void probe(int bin) {
    v.move(bin, totals[bin] + currWt / 2.0);
    gp.animate(1.0);
    lastProbe = bin;
  }
  public void pack() {
    totals[lastProbe] += currWt;
    v.setColor(Color.blue);
    gp.redisplay();
  }
}
```

As mentioned, views of an algorithm implement the methods that are defined in the interesting events interface.

The body of each method is responsible for updating the screen in a way that is meaningful for the view. For example, the `probe` method smoothly slides the rectangle representing the block being processed from its current position to a position on the bin being probed, specified as a parameter to the event. In addition, it records which bin is being probed so the `pack` event can update an array that maintains the total weight of the blocks in each bin.

The class `GP` is a rich, high-level 2D animation package

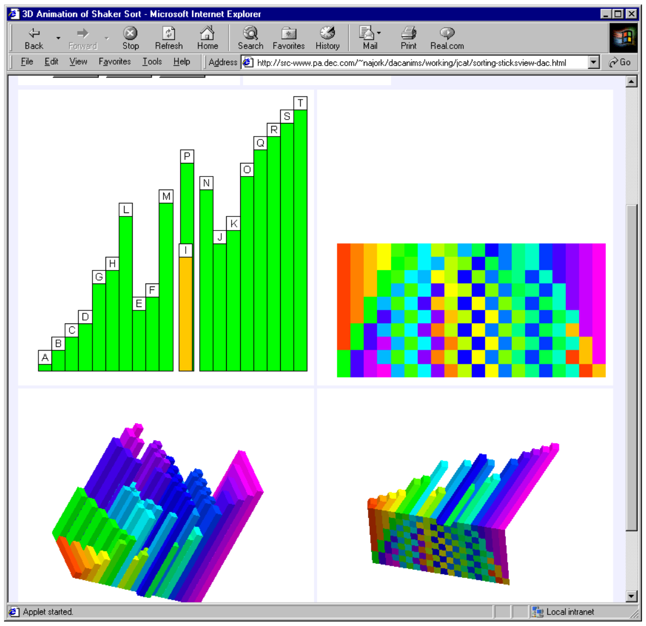


Figure 2: Animation of Shaker sort

based on the metaphor of a graph consisting of vertices and edges. Each vertex has various attributes associated with it, such as position, size, shape, color, border width, and label. A vertex can be surrounded by colored highlights, and a highlight can be moved between vertices. An edge connects two vertices and has attributes such as color, thickness, curvature, and arrowheads. The `GP` package also provides colored polygons, specified by a sequence of vertices. Vertices can be repositioned, and such movement can be shown by smooth animation. When a vertex is moved, all highlights, edges, and polygons associated with it are smoothly moved as well. `GP` was inspired by Stasko's TANGO system [13].

3. A GALLERY OF JCAT ANIMATIONS

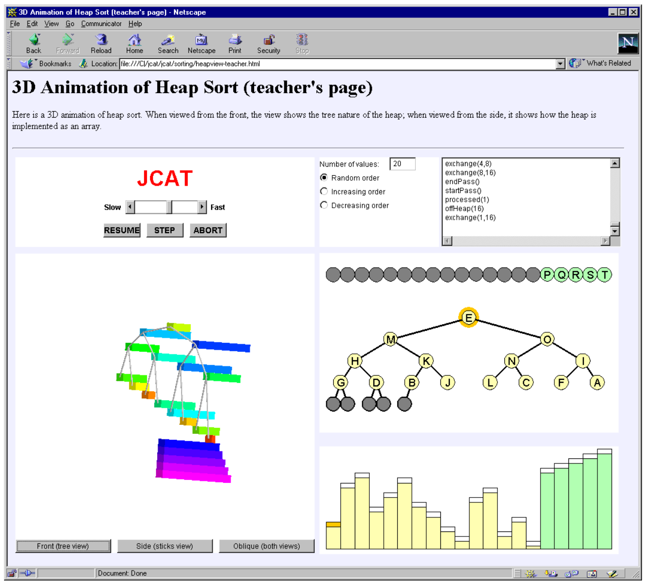
This section describes some some of the animations we have built using the JCAT system. In the process, it describes a number of algorithm-independent features provided by JCAT, such as code views and storyboard view.

3.1 Shaker sort

Figure 2 shows an animation of Shaker sort, one of the many array-based sorting algorithms. The figure shows four views of the algorithm; the control panel and the input panel are scrolled out of sight.

The top-left view is the "sticks view", a common 2D representation of an array of elements: each stick represents an element in the array, and the height of the stick is proportional to the value of the element. Whenever two elements of the array are exchanged, the corresponding sticks trade places in a smooth animation. When the array is completely sorted, the sticks will be arranged from short to tall, from left to right.

The top-right view is the "chips view", which captures a history of the contents of the array being sorted. A chip represents an element of the array. As in the "sticks view", the horizontal position of a chip indicates its position in the array. However, values of array elements are expressed


3D Animation of Heap Sort (teacher's page) - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: file:///C|/jcat/jcat/sorting/heapview-teacher.html
What's Related
JCAT
Slow
Fast
RESUME
STEP
ABORT
Front (tree view)
Side (sticks view)
Oblique (both views)
Document: Done

3D Animation of Heap Sort (teacher's page)

Here is a 3D animation of heap sort. When viewed from the front, the view shows the tree nature of the heap; when viewed from the side, it shows how the heap is implemented as an array.

Number of values: 20
[✓ Random order] Increasing order Decreasing order

```
exchange(4,8)
exchange(8,16)
endPass()
startPass()
processed(1)
offHeap(16)
exchange(1,16)
```

Figure 3: Animation of Heapsort

through color rather than height. This frees up one dimension, which has been used to capture a history of the algorithm's execution. Whenever the algorithm completes an iteration of its outer loop, a copy of the topmost row of chips is drawn on top. Changes to array elements affect the chips in the topmost row only. When the array is sorted, the topmost row will show the color spectrum from red to violet.

The lower-left view is a 3D version of the sticks view. As in the 2D view, exchanging two elements in the array is shown with a smooth animation of the corresponding sticks trading places. Whenever the algorithm completes an iteration of its outer loop, the frontmost row of sticks is duplicated. As the algorithm continues, changes are made to the frontmost row of sticks only. In this way, the 3D sticks view combines the elegance of the sticks view with the chips view's capacity for capturing history.

The lower-right view is a variation of the 3D sticks view. In this view, the current contents of the array are displayed by the row of sticks in the rear. After each iteration of the main loop, the sticks stamp their color onto a plane, which is then pulled forward. Thus, the history plane looks exactly like the chips view.

3.2 Heapsort

Figure 3 shows an animation of Heapsort. Heapsort works in two phases: First, it arranges the elements being sorted into a heap, a complete binary tree in which the value of each node is larger than the values of each of its children. Second, it repeatedly removes the root (i.e., the largest value among the elements) from the heap, sets it aside, and reestablishes the heap property, doing so until the heap is empty. Heaps can be implemented as arrays by placing the root node at position 1, and for each node at position i, placing its left child at position 2i and its right child at position 2i+1.

The web page contains six applets: the control panel; the algorithm input panel; a "transcript view;" a "tree view" that shows the heap as a binary tree; a "sticks view" that shows how the heap is implemented as an array; and a 3D

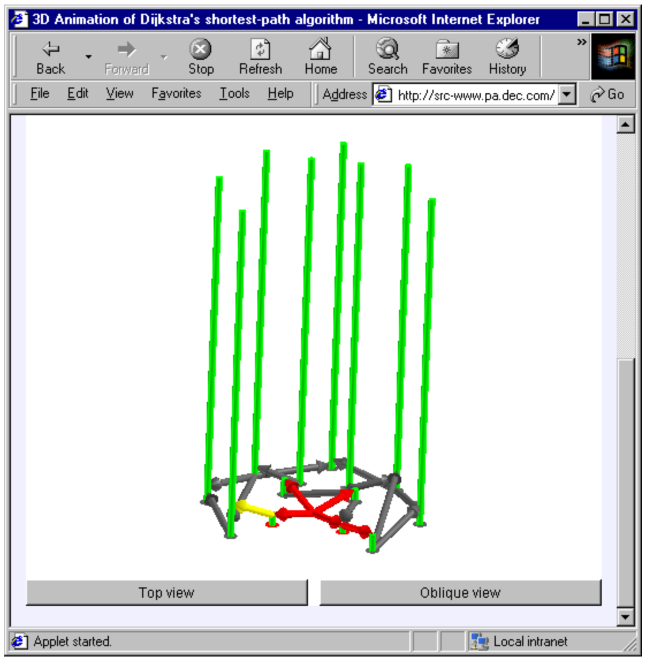

3D Animation of Dijkstra's shortest-path algorithm - Microsoft Internet Explorer
Back Forward Stop Refresh Home Search Favorites History
File Edit View Favorites Tools Help
Address http://src-www.pa.dec.com/
Go
Top view
Oblique view
Applet started.
Local intranet

Figure 4: Animation of Dijkstra's shortest-path algorithm

view that combines the tree view and the sticks views. In the tree view and sticks view, color is used to distinguish which elements are on the heap and which have been removed.

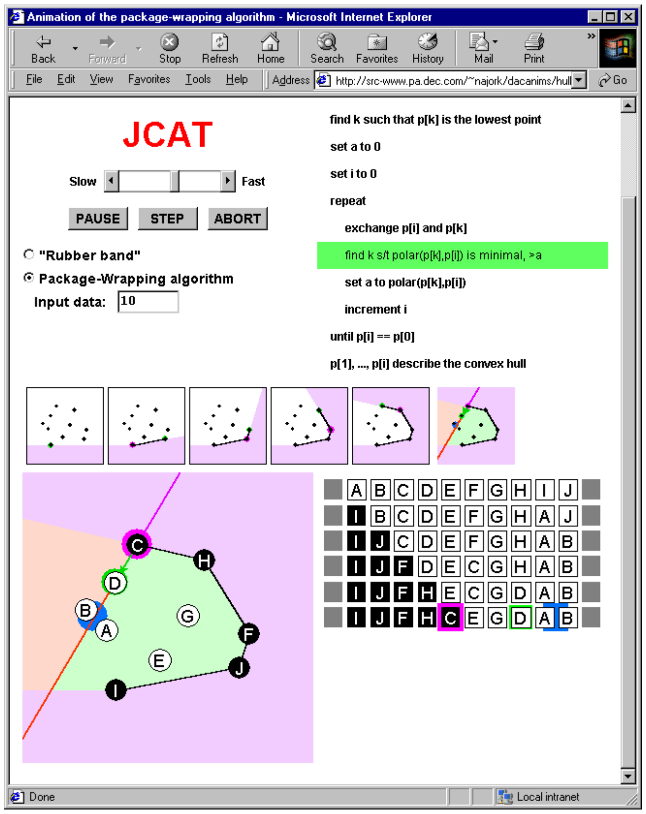
The 3D view shows a binary tree whose nodes are rods that extend in the z-dimension. As in the sticks view, the length of each rod is proportional to the value of the corresponding heap element. In addition, the rods are color-coded in the same way as in the chips view. It might appear as if color were redundant since the values are already encoded by their lengths of the rods; however, it is quite helpful when the tree is viewed from the front. Siblings in the binary tree are not drawn at the same y value; rather, right nodes are slightly lower than their left siblings. More precisely, the y position of each rod corresponds to the index of the corresponding array entry. The effect of this layout is that the tree, when viewed from the side, shows the familiar sticks view (but rotated 90 degrees).

The 3D view does not contain any more information than is contained in the two 2D views. However, integrating the two 2D views requires cognitive effort, while obtaining the same information from the single 3D view leverages the viewer's perceptual system and thereby lessons the cognitive load.

3.3 Shortest-Path

Figure 4 shows an animation of Dijkstra's shortest-path algorithm. Given a directed graph with weighted edges, this algorithm finds the shortest path from a source vertex to all other vertices. The length of a path is defined to be the sum of the weights of the edges along the path.

To do this, Dijkstra's algorithm associates a cost with each vertex, indicating the length of the shortest path found so far from the source to this vertex. Initially, this cost will be infinite. It also maintains a bit per vertex indicating whether the vertex has been processed or not. The algorithm then


Animation of the package-wrapping algorithm - Microsoft Internet Explorer
Back
Forward
Stop
Refresh
Home
Search
Favorites
History
Mail
Print
File Edit View Favorites Tools Help
Address http://src-www.pa.dec.com/~najork/dacanims/hull
Go
JCAT
Slow
Fast
PAUSE
STEP
ABORT
"Rubber band"
Package-Wrapping algorithm
Input data: 10
Done
Local intranet

```
find k such that p[k] is the lowest point
set a to 0
set i to 0
repeat
    exchange p[i] and p[k]
    find k s/t polar(p[k],p[i]) is minimal, >a
    set a to polar(p[k],p[i])
    increment i
until p[i] == p[0]
p[1], ..., p[i] describe the convex hull
```

```
A B C D E F G H I J
I B C D E F G H A J
I J C D E F G H A B
I J F D E C G H A B
I J F H E C G D A B
I J F H C E G D A B
```

Figure 5: Animation of the package-wrapping algorithm

repeatedly selects the unprocessed vertex u with the minimal cost, marks it as processed, and lowers the cost of each neighboring vertex v to the cost of u plus the weight of the edge (u, v), provided that this value is indeed lower than the current cost of v.

In the view, each vertex is shown as a disk, and the current cost of the vertex is represented by as a green column protruding from the disk whose height is proportional to the cost. Edges are shown as arrows. Each edge leaves a vertex at height 0, and enters the other vertex at a height proportional to its weight. Unprocessed edges are shown in grey, the edge that is currently being examined is shown in yellow, and processed edges are shown in red.

When the algorithm marks a vertex as processed, the color of the vertex in the view is changed from gray to red. When the algorithm considers an edge from u to v, the edge is highlighted in yellow, and lifted by the current cost of u. Hence, after the lifting is complete, the startpoint of the edge coincides with the tip of the green column above u. If the endpoint of the lifted edge is lower that the green column above v, this column is lowered to the endpoint, reflecting the lowering of the cost of v by the algorithm, and the color of the edge changes from yellow to red. Otherwise, the yellow edge simply disappears. When the algorithm is finished, the graph has been replaced by the shortest-path tree, and the columns above each vertex reflect the length of the shortest path from the source to that vertex.

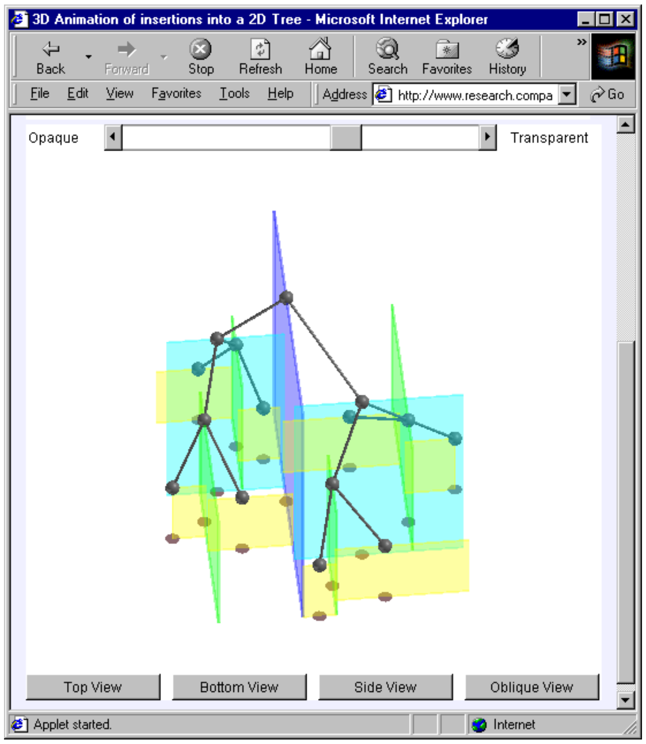

3D Animation of insertions into a 2D Tree - Microsoft Internet Explorer
Back
Forward
Stop
Refresh
Home
Search
Favorites
History
File Edit View Favorites Tools Help
Address http://www.research.compa
Go
Opaque
Transparent
Top View
Bottom View
Side View
Oblique View
Applet started.
Internet

Figure 6: Animation of construction of a 2D tree

3.4 Package-Wrapping

Figure 5 shows an animation of the package-wrapping algorithm for computing the convex hull of a set of points in the plane. The figure contains the control panel, the input panel, and four views of the algorithm.

The view at the top-right is a "code view"; it shows a pseudo-code representation of the algorithm, with a highlight on the line that is currently being executed. The pseudo code was provided by the animator, but the code view applet was created automatically by the CATalyst preprocessor.

The lower-left view is the "geometry view"; it shows how the segments of the hull are wrapped around the points in the plane.

The view in the middle is a "storyboard view". It shows snapshots of the geometry view taken at key points in time. The rightmost frame in the storyboard view is "live"; it shows the same animations as the full-sized geometry view. When a key event occurs (in this case, when a new point becomes part of the convex hull), the frame freezes, and a new live frame appears to its right.

The lower-right view shows the main data structure of the algorithm, namely, an array of points. The points are shown as squares, the array is represented by a row of squares. The view captures a history of how the array evolves by taking snapshots of the row representing the array and arranging the snapshots vertically, similar in spirit to the chips view used in the animation of Shaker sort.

3.5 Construction of a 2D Tree

Figure 6 shows an animation of the construction of a 2D tree, a space-partitioning tree that contains points in the plane. This data structure provides support for efficient range searching. A 2D tree is essentially a binary search tree with 2D points in the nodes, using the x and y coordinates of the points as keys in a strictly alternating sequence.

When viewed from the top, the scene shows the points in the plane. The walls indicate the partitioning imposed by nodes in the 2D tree. The large blue wall corresponds to the root of the tree; the two cyan walls correspond to its children, and so on. The edges of the 2D tree are displayed as black lines. This representation of a tree resembles the standard layout of a spanning tree, but it does not reveal parent-child relationships. When viewed from the side, the space partitioning tree has the standard binary tree layout, where parent nodes are located above their children. The drawback of this viewing angle is that we can no longer see the points in the plane. When viewed from an oblique angle (as in Figure 6), the visualization simultaneously shows the points in the plane, the walls partitioning the plane that are induced by the nodes of the 2D tree, and the 2D tree.

Arguably, it is disconcerting to see the edges of the tree overlapping; moreover, the left and right children are not necessarily drawn to the left and right of their parent! However, we found that when the tree is interactively rotated about the z-axis, it appears to have depth. The rotation provides the viewer with the visual cues needed to understand the overlaps and perceive the tree's depth. This example demonstrates that it is crucial for the viewers of 3D animations to be able to interactively manipulate the scene.

4. CONCLUSION

This paper describes JCAT, a Web-based algorithm animation system. JCAT augments the expressive power of web pages (which can contain passive multimedia such as text, images, or movies) with interactive animations of algorithms.

Unlike traditional algorithm animation systems, JCAT does not require any software installation. Animations are immediately accessible simply by pointing a standard Web browser to a page containing an animation. At first blush, this might seem a minor detail, but we believe it makes a significant difference. Having built several stand-alone systems, we learned that installation overhead does deter many potential users. By making the installation completely automatic, we create a seamless user experience. We believe that this technology will expose a much wider audience to algorithm animation.

You can experience the examples shown in this paper by visiting `http://research.compaq.com/SRC/JCAT`.

5. ACKNOWLEDGMENTS

The research described in this paper was done in collaboration with Marc Brown. Some of this work has been described previously [9].

6. REFERENCES

[1] R. Baecker and D. Sherman. Sorting Out Sorting. 16mm color sound film, 30 minutes, Dynamic Graphics Project, Univ. of Toronto, 1981.

[2] J. Baker, I. Cruz, G. Liotta, and R. Tamassia. Algorithm animation over the World Wide Web. In *Proc. of the International Workshop on Advanced Visual Interfaces*, pp. 203-212, May 1996.

[3] K. Booth. PQ Trees. 16mm color silent film, 12 minutes, 1975.

[4] M. Brown. A System for Algorithm Animation. In *Proc. of ACM SIGGRAPH'84*, pp. 177-186, July 1984.

[5] M. Brown. ZEUS: A System for Algorithm Animation and Multi-View Editing. In *Proc. of the 1991 IEEE Workshop on Visual Languages*, pp. 4-9, Oct. 1991.

[6] M. Brown and J. Hershberger. Color and Sound in Algorithm Animation. *Computer* **25**(12):52-63, Dec. 1992.

[7] M. Brown and M. Najork. Algorithm Animation Using 3D Interactive Graphics. In *Proc. of the 1993 ACM Symposium on User Interface Software and Technology*, pp. 93-100, Nov. 1993.

[8] M. Brown and M. Najork. Collaborative Active Textbooks: A Web-Based Algorithm Animation System for an Electronic Classroom. In *Proc. of the 1996 IEEE Symposium on Visual Languages*, pp. 266-275, Sept. 1996.

[9] M. Brown, R. Raisamo, and M. Najork. A Java-based implementation of Collaborative Active Textbooks. In *Proc. of the 1997 IEEE Symposium on Visual Languages*, pp. 372-379, Sept. 1997.

[10] R. Duisberg. Animated Graphical Interfaces Using Temporal Constraints. In *Proc. of the ACM CHI'86 Conference on Human Factors in Computing Systems*, pp. 131-136, April 1986.

[11] J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Terävirta, and P. Vanninen. Animation of User Algorithms on the Web. In *Proc. of the 1997 IEEE Symposium on Visual Languages*, pp. 360-367, Sept. 1997.

[12] F. Hopgood. Computer Animation Used as a Tool in Teaching Computer Science. In *Proc. of the 1974 IFIP Congress*, pp. 889-892, 1974.

[13] J. Stasko. TANGO: A Framework and System for Algorithm Animation. *Computer* **23**(9):27-39, Sept. 1990.

[14] J. Stasko and J. Wehrli. Three-Dimensional Computation Visualization. In *Proc. of the 1993 IEEE Symposium on Visual Languages*, pp. 100-107, Aug. 1993.

[15] J. Stasko, J. Domingue, M. Brown, and B. Price (eds.). Software Visualization: Programming as a Multimedia Experience. MIT Press, 1998.