110a

Algorithm Animation Using 3D Interactive Graphics

Marc H. Brown and Marc A. Najork

September 15, 1993

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

Algorithm Animation Using 3D Interactive Graphics

Marc H. Brown and Marc A. Najork

September 15, 1993

Publication History

This report will appear in the *Proceedings of the ACM Sixth Annual Symposium on User Interface Software and Technology*, November, 1993.

©Digital Equipment Corporation 1993

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the Systems Research Center of Digital Equipment Corporation in Palo Alto, California; an acknowledgment of the authors and individual contributors to the work; and all applicable portions of the copyright notice. Copying, reproducing, or republishing for any other purpose shall require a license with payment of fee to the Systems Research Center. All rights reserved.

Abstract

This report describes a variety of 3D interactive graphics techniques for visualizing programs. The third dimension provides an extra degree of freedom for conveying information, much as color adds to black-and-white images, animation adds to static images, and sound adds to silent animations. The examples in this report illustrate three fundamental uses of 3D: for providing additional information about objects that are intrinsically two-dimensional, for uniting multiple views, and for capturing a history of execution. The application of dynamic three-dimensional graphics to program visualization is largely unexplored. A videotape of these animations is available.

Overview

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 both for education and for research in the design and analysis of algorithms.

The first general-purpose algorithm animation systems in the early 1980's used monochrome displays. Systems such as BALSA [3] were constrained by a lack of computational power for real-time two-dimensional graphics. As computational power has increased, so has the sophistication of the graphics techniques used for animating algorithms.

In the mid-1980's, Animus [6] showed the utility of smooth transformations of 2D images, especially for looking at small examples. TANGO [11] in the late 1980's provided an elegant framework for specifying 2D animations. Color was an integral part of Zeus [1]. The Zeus system also pioneered "algorithm auralization"—using non-speech sound to convey the workings of algorithms [2]. Not surprisingly, each new advance in technology has enabled an extra level of expressiveness to be added to the visualizations.

This report describes our use of 3D interactive graphics for algorithm animation. Three-dimensional interactive graphics provides another level of expressiveness to the animations, akin to the way that smooth transitions, color, and sound have increased the level of expressiveness in the past.

We are not proposing to use 3D for showing objects that are intrinsically three-dimensional, such as the convex hull of points in 3-space. We are also not advocating the use of 3D for enhancing the beauty of a picture that is easily shown in 2D. Rather, we are using 3D to increase the quality and quantity of information conveyed in a graphical display. Specifically, we have explored three distinct uses of 3D:

- *Expressing fundamental information about structures that are inherently two-dimensional.*

  Consider how one might display the values of an N-by-M, two-dimensional matrix of positive numbers. An obvious 3D display is to draw sticks at each cell of an N-by-M grid, where the height of each stick is proportional to the value of the corresponding element. Of course there are other techniques for displaying the matrix without using 3D graphics, such as displaying a number in each cell or modifying the color, shape, or size of each cell according to the value of the corresponding element of the matrix. However, showing sticks of varying heights seems to be an extremely effective technique, perhaps because it allows direct visual comparison of the elements.

- *Uniting multiple views of an object.*

  Finding a single view of an object that reveals all of its features can be difficult, if not impossible. Therefore, presenting multiple views of that object is a helpful visualization technique. However, it can be difficult for the user to understand the relationship between the multiple views. A carefully crafted 3D view can incorporate multiple 2D views into a single image, thereby helping the user to see how the views are related.

- *Capturing a history of a two-dimensional view.*

  Often, a visualization of a program's entire execution history can be just as helpful for understanding a program's behavior as an animation of the current state of the program. When running programs on small amounts of data, a history often gives the user a context of how the algorithm has progressed each time the state has changed. When running programs on large amounts of data, a history often exposes patterns that are not otherwise observable.

In any event, identifying and quantifying the advantages and drawbacks of visualization techniques is beyond the scope of this report.

This report presents six example animations that exemplify our three distinct uses of 3D interactive graphics. The first two examples, Shortest Path and Closest Pair, use 3D for showing additional information on a structure that is inherently two-dimensional. The next three examples, Heapsort, k-d Trees, and Balanced Trees, use 3D for uniting multiple views. The final example, Elementary Sorting, uses 3D for augmenting a view with a history of how it has changed over time.

The techniques used in the Closest Pair and Elementary Sorting examples are particularly noteworthy because they can be applied to many arbitrary 2D views. We shall return to this point when describing those two examples.

The images in this report are screen dumps of views we developed using the Zeus algorithm animation system [1]. In the Zeus framework, strategically important points of an algorithm are annotated with procedure calls that generate "interesting events." These events are reported to the Zeus event manager, which in turn forwards them to all interested views. Each view consists of two windows that are installed on the user's desktop. One window displays the actual 3D image and the other window contains a control panel (see the Balanced Trees example). The control panel allows the user to change generic rendering parameters (e.g., lighting), as well as view-specific parameters (e.g., in the case of Balanced Trees, the distance between the two trees). The rendered scene can be moved, rotated, and scaled through mouse controls. In addition, one can use the mouse to specify a momentum, which will cause the scene to rotate continuously.

It is important for readers to realize that the figures in this report cannot do justice to the animations: Not only is the reader unable to manipulate the 3D scenes, but the scenes themselves are not static! They are constantly changing as the algorithm runs.

(The accompanying videotape is helpful, but a viewer of a videotape cannot manipulate the 3D scene, change the speed of the animation, vary the input data, and so on.)

Example 1: Shortest Path

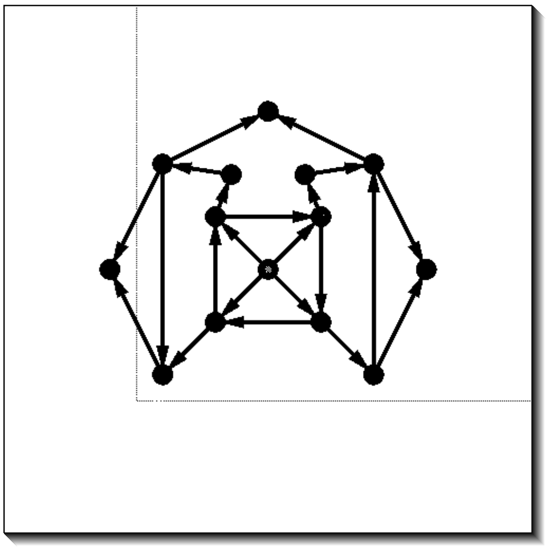
Single-source shortest-path algorithms are a family of algorithms which, given a directed graph with weighted edges, find the shortest path from a designated vertex, called the *source*, to all other vertices. The length of a path is defined to be the sum of the weights of the edges along the path.

All single-source shortest-path algorithms have certain common features: First, they assign a cost to each vertex, indicating the length of the shortest path found so far from the source to this vertex. Initially, the cost will be infinite. Then they repeatedly choose an edge e from vertex u to vertex v, test if it can lower the cost associated with v, that is, if $\textsc{cost}(v) > \textsc{cost}(u) + \textsc{weight}(e)$, and if so, indeed lower v's cost. Algorithms differ in their choice of which edge to examine next.

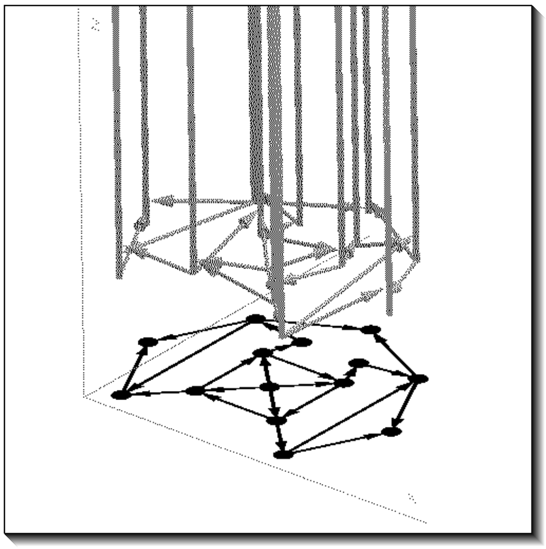
A 3D view of one such algorithm is shown in Fig. 1. The graph is drawn in the xy plane, with a green column above the node in the z dimension. The column represents the cost of each node. An edge from u to v with weight w leaves the column above u at height 0, and goes into the column above v at height w. Figs. 1a and 1b show the initial state of the algorithm, from above (Fig. 1a) and from an oblique viewing perspective (Fig. 1b). Edges are drawn in gray; a shadow of the edges is projected into the xy plane and drawn in black, along with the vertices.

Whenever an edge e from u to v is examined, a highlighted, red copy of it is lifted to the top of u's green column, hence its tip will hover over v at height proportional to $\textsc{cost}(u) + \textsc{weight}(e)$. If v's column is taller, the edge can indeed lower v's cost, so v's column is shortened, otherwise, the highlighted edge disappears. The set of highlighted lifted edges forms the shortest-path tree when the algorithm terminates. Fig. 1c shows the algorithm about halfway complete; Fig. 1d shows the algorithm upon completion, with the initial edges not drawn.

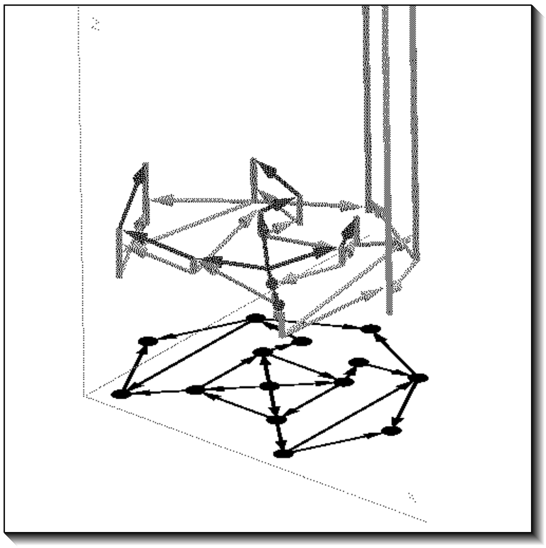
This view uses the third dimension to provide state information (namely, cost of vertices and weight of edges) about an algorithm as it operates on a data structure that uses two dimensions for placing objects. The view uses animation effects to show the fundamental operations of the algorithm: lifting an edge represents addition, lowering a highlighted edge indicates the outcome of a comparison, shortening a column shows assignment.



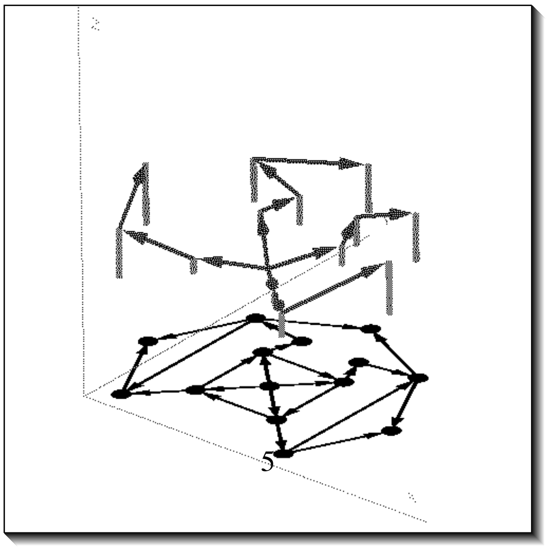
1a



1b



1c



1d

Example 2: Closest Pair

The closest-pair problem is to find the two points in a collection of n ponts that are closest to each other. An algorithm that does a pairwise comparison of all points takes $O(n^2)$ time; however, a recursive, divide-and-conquer algorithm can improve this time bound to $O(n \log n)$.

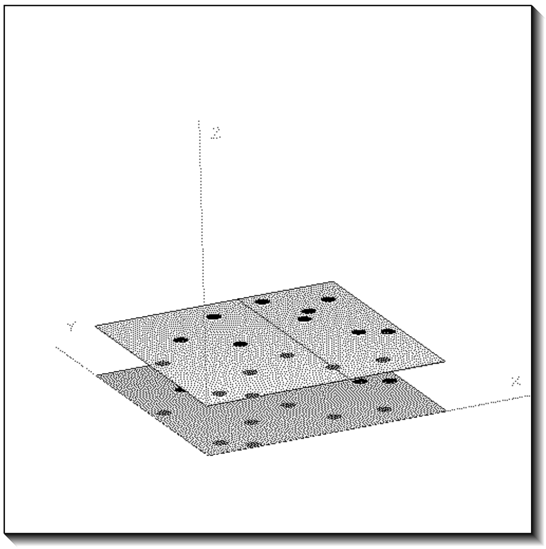
The algorithm (for points in the plane) is as follows: First we divide the plane by a line parallel to the y-axis such that each half contains the same number of points. Next, we recursively find the closest pair of points in each half. And finally, we merge the two halves, checking if there is a new pair of points (saddling the dividing line) that are closer to each other than the closest pairs in each half. The crux of the algorithm is that we need to consider only those points in each half that are fairly close to the dividing line (in x) and fairly close to the other endpoint (in y).

The 3D view of this algorithm, shown in Fig. 2, draws each half-plane in the xy plane and uses the z axis to show the division process and the induced recursion structure. In each divide step, the half-plane is lifted, split in the middle, and the two halves are moved apart. In the merge step, the halves are moved back together, the eligible points are compared pairwise, and then the merged plane is lowered. The region of interest around the dividing line is highlighted. The globally best pair found so far is also highlighted.

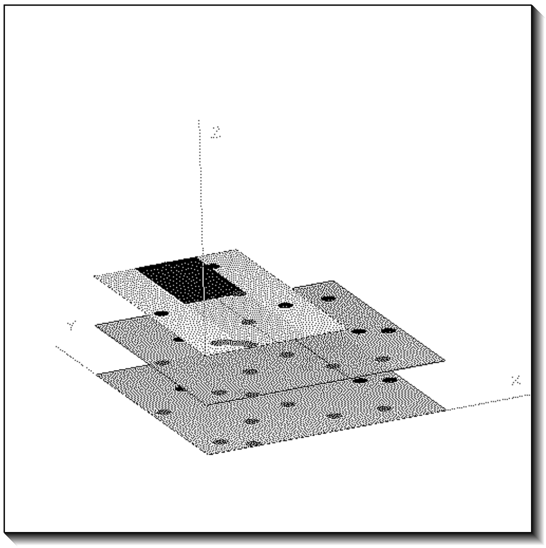
In Fig. 2, the user has specified that the half-planes should not be moved apart, and that the half-planes should be displayed almost completely opaque. Fig. 2a shows the initial splitting, Fig. 2b shows the merge of the two half-planes split by the initial left half-plane, and Fig. 2c shows the algorithm deep in recursion as it processing the initial right half-plane.

As in the Shortest Path example, the third dimension is used to display additional information (in this case, the recursion structure) about an algorithm that operates on two-dimensional data, and animation effects are used to show operations crucial to the algorithm.

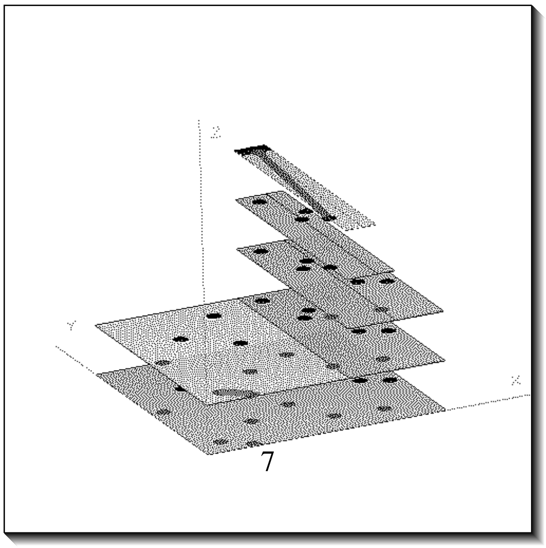
The visualization technique here is an example of a general-purpose way to integrate a visualization of a program's calling structure with the contents of its data structures. We believe that it can be applied to other views, although we have not explored this yet.



2a



2b



2c

Example 3: Heapsort

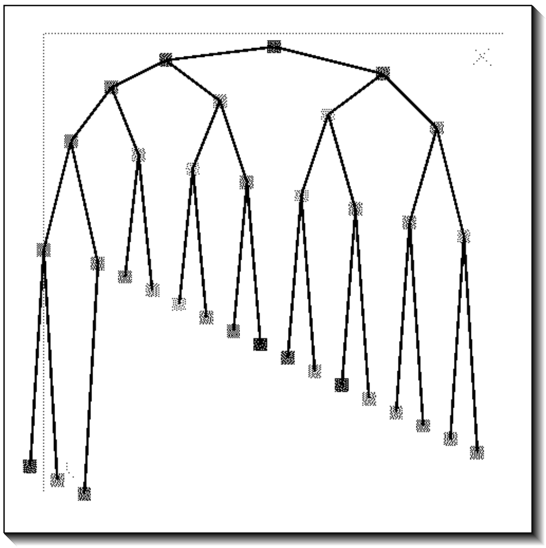
Heapsort works in two phases. First, it arranges the elements being sorted into a *heap*, a complete binary tree in which the value of each node is larger than the values of each of its children. Second, it repeatedly removes the root (i.e., the largest value among the elements) from the heap, sets it 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 $2i + 1$.

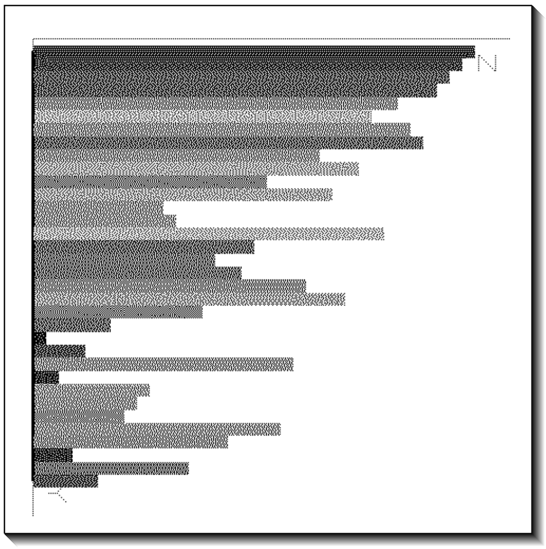
The 3D view in Fig. 3 exposes both of these properties. When viewed from the front as in Fig. 3a, we see the heap configured as a traditional tree (drawn in the xy plane). Each node in the tree is an element of the array being sorted, and has depth (in the z dimension) proportional to its value. Thus, nodes at the top of the tree are longer (or deeper) than those near the leaves. When the tree is viewed from the side as in Fig. 3b, we see a classical sticks view of sorting algorithms (cf. Fig. 6). Fig. 3c shows the same structure from an oblique viewing angle. Notice the relationship between the two representations. Fig. 3d shows the algorithm when it's almost completed.

The value of elements are also encoded by colors along the spectrum: large elements are displayed in red and small elements in blue. Color is not crucial in the sticks representation, because the value of a stick is encoded by its length, but it is quite helpful in the tree view.

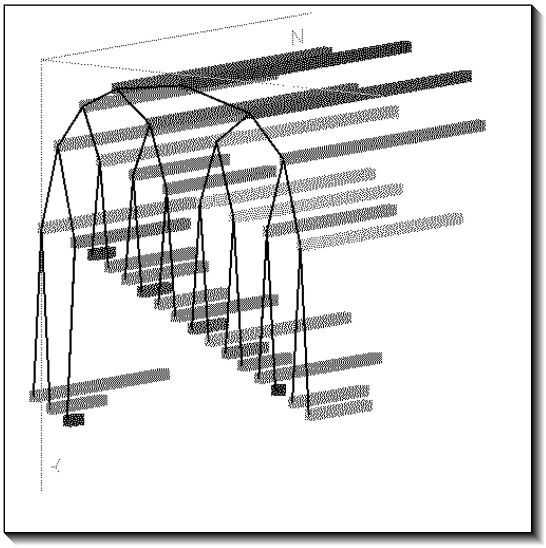
Of course, it is possible to show the two perspectives—the tree and the array—as separate views, each in its own window, without using 3D graphics. However, the viewer must mentally integrate the different views in order to understand them as a whole. The 3D view alleviates this problem.



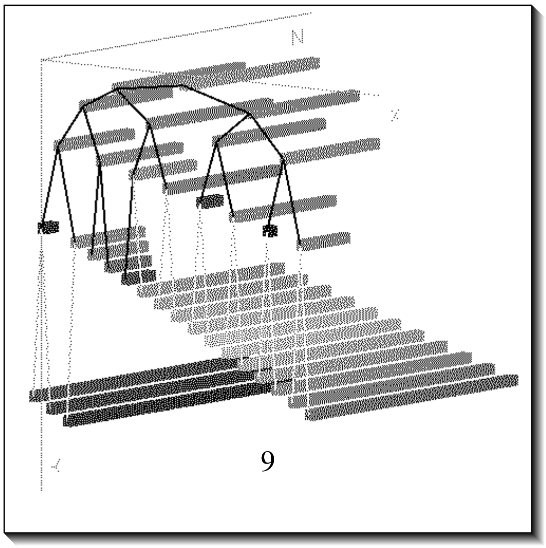
3a



3b



3c



3d

Example 4: k-d Trees

k-d trees are a special kind of search tree, useful for answering range queries about a set of points in k-space. The algorithm for the two-dimensional case (i.e., $k = 2$) with points in the xy plane is as follows: The algorithm selects any point and draws a line through it parallel to the y axis. This line partitions the plane vertically into two half-planes. Another point is selected and is used to horizontally partition the half-plane in which it lies. In general, a point that falls in a region created by a horizontal partition will divide this region vertically, and vice versa.

This division process induces a binary tree structure: The first point becomes the root, and each point falling into the left half-plane is inserted into the left subtree, and each point falling into the right half-plane is inserted into the right subtree. For points that divide regions horizontally, the points in the upper half-plane are inserted into the left subtree whereas points in the lower half-plane are inserted into the right subtree. Thus, nodes at even levels in the tree divide the set of points into left and right half-regions, and nodes at odd levels divide a region into upper and lower half-regions.

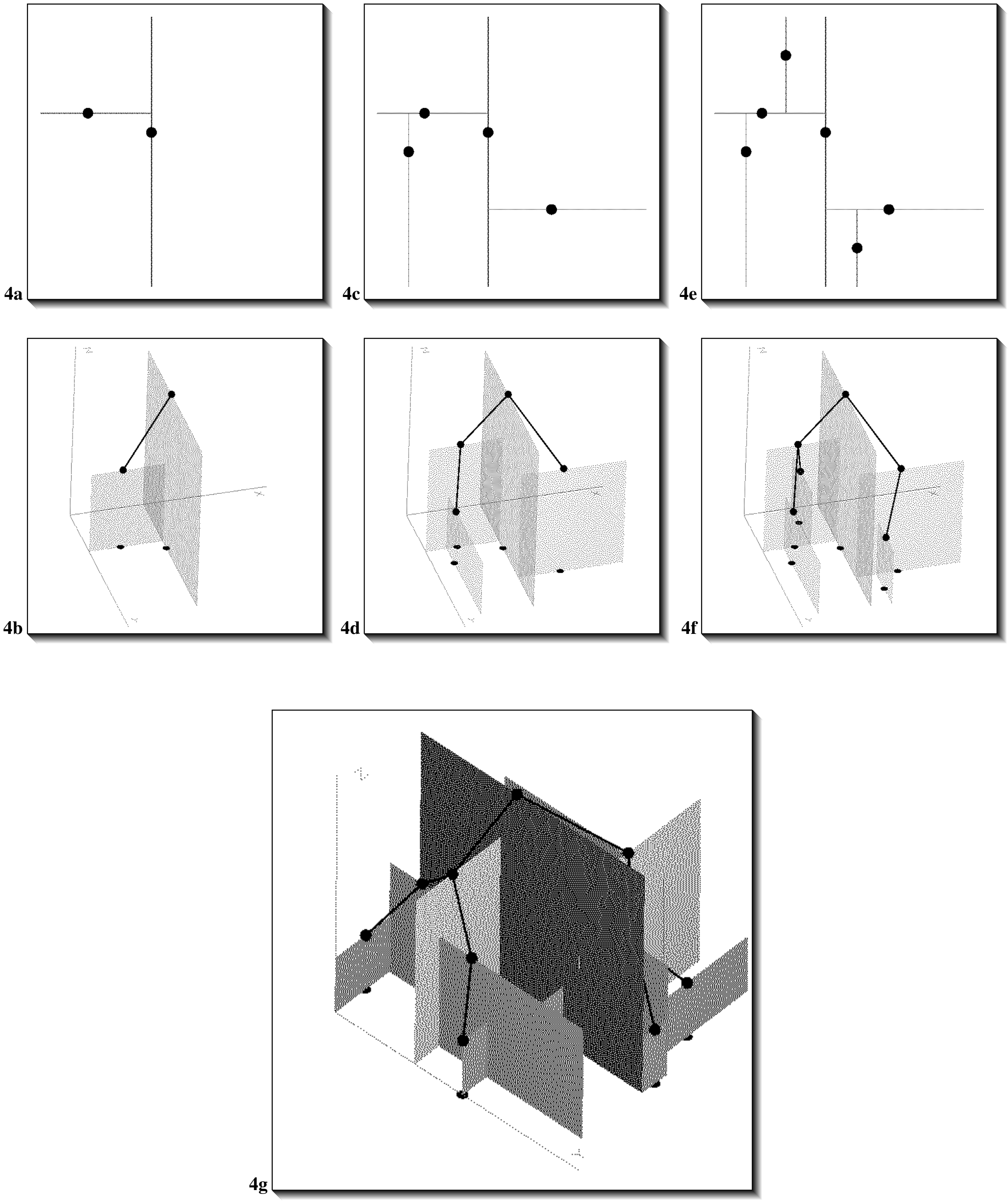
There are two obvious views of this algorithm: a view of the partitioning of the plane, and a view of the binary tree that is induced. The 3D view shown in Fig. 4 merges and unites these two views.

The points in the plane are drawn as circles in the xy plane, and the partitionings caused by them are drawn as transparent walls extended in the z dimension. On top of each wall and above each point is a sphere, representing the corresponding node in the 2-d tree. Therefore, the height (as well as color) of each wall reflects the node's level in the tree. The tree edges are represented as lines connecting related nodes.

When viewed from the top and with the tree edges hidden, as in Figs. 4a, 4c, and 4e, we see the traditional view of the partitioning of the plane. Exposing the tree edges would show the 2-d tree in a representation that a graph-theorist would be comfortable with: as a connected, acyclic graph. However, when viewed from the side (and with the walls mostly transluscent), as in Figs. 4c, 4d, and 4f, we see a tree more familiar to the computer scientist: each node is below its parent.

Figs. 4a and 4b show the state of the algorithm after it has processed the first two points. Figs. 4c and 4d show the state after the algorithm has processed the third and fourth points, and Figs. 4e and 4f, after the algorithm has processed the fifth and sixth points. Finally, Fig. 4g shows the state of the algorithm after all points have been processed, with opaque partitioning walls. Notice how the 3D view merges the traditional plane-partitioning view and the induced 2-d tree view.

It is disconcerting to see edges of the tree overlapping. (Moreover, the left and right children are not necessarily drawn to the left and right of their parent!) Fortunately, when the tree is rotated in real-time about the z axis, it appears to have depth. The real-time animation provides the viewer with the visual clues needed to understand the overlaps.

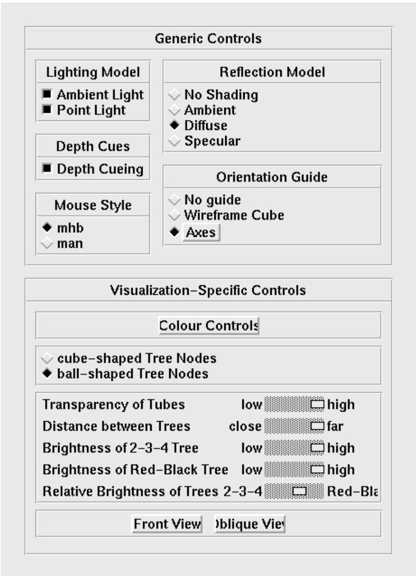

4a
4c
4e
4b
4d
4f
4g

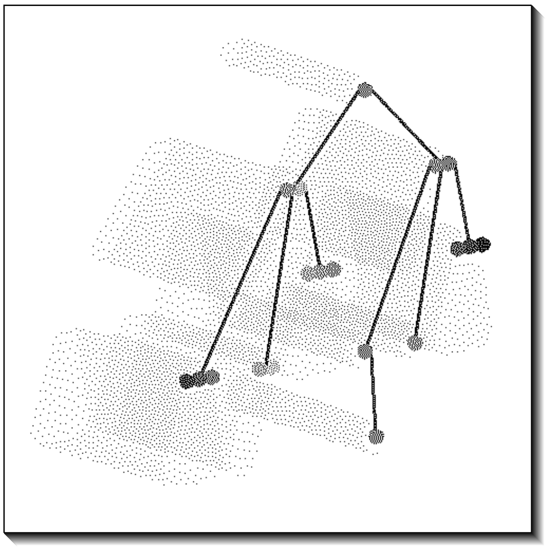
Example 5: Balanced Trees

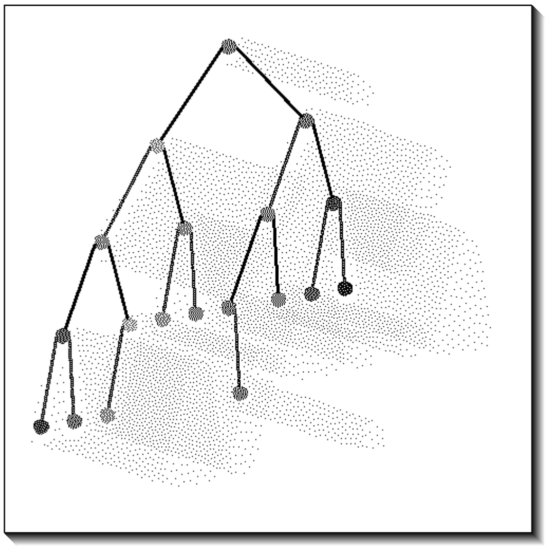
A 2-3-4 tree is a balanced search tree in which nodes can contain 1, 2, or 3 keys and can have 2, 3, or 4 children. Inserting keys into a node might eventually cause it to overflow, which results in the node being split into multiple nodes. Performing the split operation judiciously will keep the tree balanced. Unfortunately, 2-3-4 trees are cumbersome to implement, mainly due to their irregular structure. Therefore, it is common to implement 2-3-4 trees as Red-Black trees. These are ordinary binary search trees with an extra bit (the "color") attached to each node.

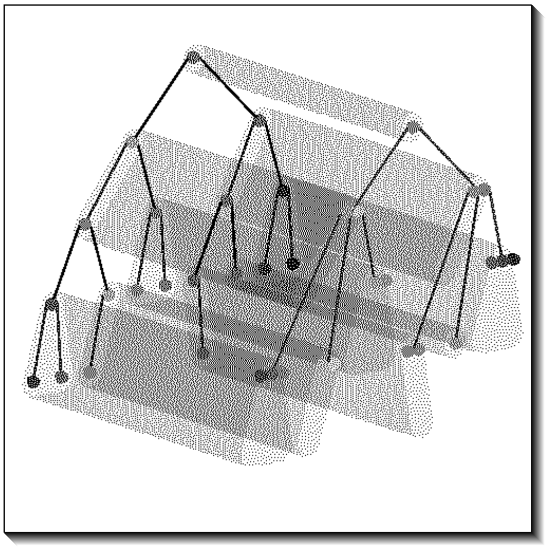
The 3D view shown in Figs. 5a, 5b, and 5c illustrates the mapping between a 2-3-4 tree and a Red-Black tree. The two trees are drawn with one in front of the other. More precisely, each tree is drawn in the xy plane, and the trees have different values of z. Each node in the 2-3-4 tree is associated with its corresponding nodes in the Red-Black tree by enclosing the nodes in both trees into a horizontal transparent envelope, and thus grouping them together.

This view, like the others, is somewhat hard to appreciate fully as a static image. Spinning the scene very slowly helps the viewer to see the mapping. The controls for manipulating this 3D view are as follows:




5a


5b


5c

Example 6: Elementary Sorting

Perhaps the most famous algorithm animation is the "sticks" view of sorting algorithms, shown in Fig. 6a. This view, introduced in Baecker's seminal 1981 film *Sorting Out Sorting*, shows the array of elements as a row of sticks. The height of each stick is proportional to the corresponding element in the array, so when the sort is completed, the sticks are arranged from short to tall, from left to right.

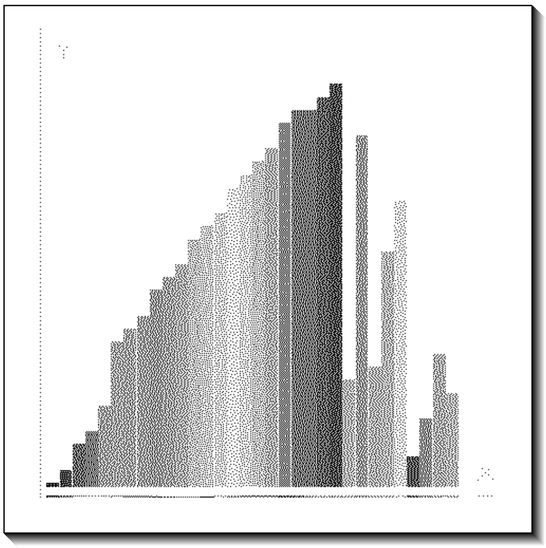
This view is superb for understanding the dynamics of many sorting algorithms, especially when the algorithm runs on small amounts of data. However, this view does not provide any history of the execution. We see the current state only. However, if we consider the sticks as being drawn in the xy plane, we can see an execution history by drawing the sticks at increasing values of z as the algorithm progresses. That is, we stack the new row of sticks in front of the old ones. This results in a 3D solid.

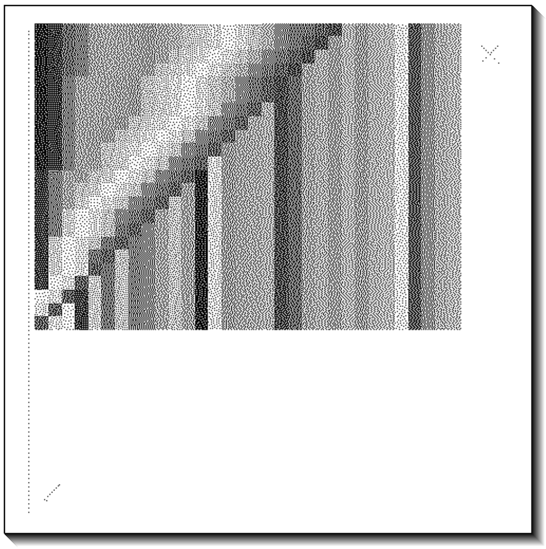
In order to emphasize the importance of the current row of sticks, we chose to flatten all previous sticks and to encode by color the value of the corresponding array element. In addition, we keep the current row fixed at $z = 0$, and move the stack of flattened sticks forward at each step. This results in a horizontal plane of "paint chips" giving a complete history of the algorithm. (Another way to think of the "chips" view is as the sticks stamping their color onto the chips plane, which is pulled forward as execution progresses.)

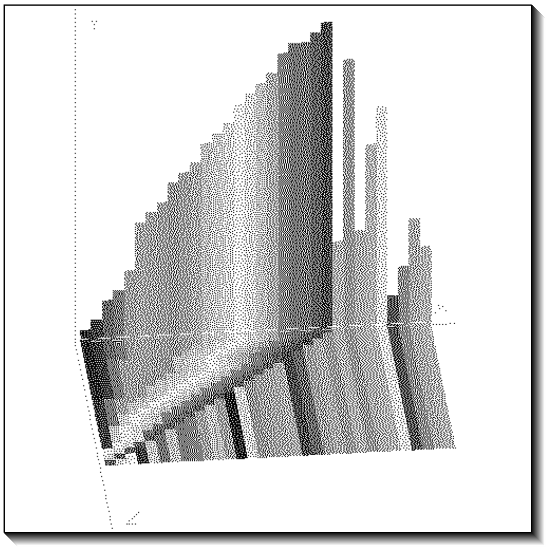
Fig. 6b shows the same scene as in Fig. 6a, but viewed from above. Fig. 6c shows the same scene again, but viewed from an oblique viewing perspective. Notice how we can see both the current contents of the array and the history of the algorithm's execution in the 3D view.

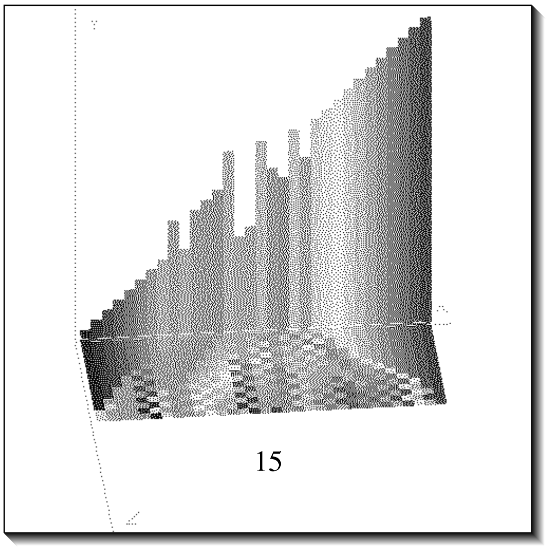
Fig. 6d uses the same view and viewing angle to display Shakersort. In the BALSA system, the chips view was a separate view of sorting algorithms, one among a dozen or so views. The chips view of Shakersort was instrumental in providing the insight that led to Janet Incerpi's Ph.D. thesis on the worst-case analysis of Shellsort (Brown University, 1985). The insight suggested by the picture of Shakersort is the "zipper" effect: in one left-to-right pass, many elements are moved one position to the left, only to be moved back to their previous position on the subsequent right-to-left pass.

This digression is important because the 3D visualization technique of stacking a 2D view along the z axis as the algorithm progresses is general purpose, and can be applied to many 2D views. It is reasonable to imagine that other hidden properties of algorithms will be exposed by examining 3D history of 2D views.


6a


6b


6c


6d

Related Work

The scientific visualization community routinely uses 3D interactive graphics. Systems like AVS [12] support 3D visualizations of domain-independent data. The Information Visualization project at Xerox PARC [4] has stimulated a flurry of interest in developing 3D views that show classical types of data organization (e.g., a tree) traditionally shown in 2D. Both scientific visualization and information visualization typically concentrate on a given set of numeric or relational data. We are concerned with visualizing the behavior of programs, which typically operate in subtle ways on abstract and complex combinatorial structures.

There are a few recent examples of using 3D in program visualization. Lieberman [8] describes a 3D view of the execution of Lisp programs. The view shows the code for an expression in the xy plane on a block with some depth in the z dimension. As the program executes, each subexpression causes a new block to be displayed in front of the caller's block. When an expression is evaluated, its block (the frontmost) is removed. Koike's VOGUE system [7] provides a 3D visualization of class libraries: A conventional class hierarchy tree is drawn in the xy plane. Behind each node (in the z axis) are "floating" nodes for methods. Finally, Reiss has developed a 3D variation of a program call graph [9], where the z coordinate (and the actual contents) of each node reflects some attribute of the corresponding procedure.

In algorithm animation per se, Cox and Roman [5] recently showed a view of a shortest-path algorithm similar to that shown in Fig. 1. Their work was developed independently of ours. Also, in an unpublished videotape, Stasko at Georgia Tech shows a clever 3D animation of Quicksort using Polka [10]. The front view is a traditional "dots" view (i.e., just the tips of the sticks in Fig. 6); as elements are exchanged, a trail is maintained whose depth is proportional to the number of exchanges that have happened so far. Viewing the scene from the side provides a history.

Summary

The potential use of 3D graphics for program visualization is significant and mostly unexplored. The examples in this report use 3D graphics for expressing additional information geometrically about a two-dimensional structure, integrating two nominally 2D views, and capturing a history of execution. Our use of 3D graphics is not to enhance the beauty of a program visualization; it provides additional, fundamental information.

Two of the examples stand out as being instances of general-purpose visualization techniques: In the Closest Pair example, we discussed using 3D for combining program control information with arbitrary 2D views of program data structures. In the Elementary Sorting example, we discussed using 3D for capturing a history of an arbitrary 2D view.

A great deal of experimentation is needed to better understand the strengths and weaknesses of using 3D interactive graphics for animating algorithms, and to develop a collection of 3D visualization techniques and metaphors to augment those that have been developed for using 2D, color, and sound.

References

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

[2] Marc H. Brown and John Hershberger. Color and Sound in Algorithm Animation. *Computer*, 25(12):52–63, December 1992.

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

[4] George G. Robertson, Stuart K. Card, and Jock D. Mackinlay. Information Visualization Using 3D Interactive Animation. *Communications of the ACM*, 36(4):56–71, April 1993.

[5] Kenneth C. Cox and Gruia-Catalin Roman. Abstraction in Algorithm Animation. *1992 IEEE Workshop on Visual Languages* (September 1992), 18–24.

[6] Robert A. Duisberg. Animated Graphical Interfaces Using Temporal Constraints. *ACM CHI '86 Conf. on Human Factors in Computing* (April 1986), 131–136.

[7] Hideki Koike. An Application of Three-Dimensional Visualization to Object-Oriented Programming. *Advanced Visual Interface '92*, Rome, Italy.

[8] Henry Lieberman. A Three-Dimensional Representation for Program Execution. *1989 IEEE Workshop on Visual Languages* (October 1989), 111–116.

[9] Steven P. Reiss. A Framework for Abstract 3-D Visualization. *1993 IEEE Symposium on Visual Languages* (August 1993), 108–115.

[10] John T. Stasko and Joseph F. Wehrli. Three-Dimensional Computation Visualization. *1993 IEEE Symposium on Visual Languages* (August 1993), 100–107.

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

[12] Craig Upson, et. al. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, July 1989.