SWAN: A Data Structure Visualization System [1]

Jun Yang
Hughes Network Systems
11717 Exploration Lane
Germantown, MD 20876
jyang@hns.com

Clifford A. Shaffer and Lenwood S. Heath
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061
shaffer@cs.vt.edu, heath@cs.vt.edu

1 Introduction

Swan is a data structure visualization system. It allows users to visualize the data structures and execution process of a C/C++ program. **Swan** views a data structure as a graph or collection of graphs. By "graph" we mean general directed and undirected graphs and special cases such as trees, lists and arrays.

As a part of Virginia Tech's NSF Educational Infrastructure Grant, **Swan** will be used in two ways: by instructors as a teaching tool for data structures and algorithms, and by students visualizing their own programs to understand how and why they do or do not work. To use **Swan**, a program must first be *annotated*, i.e., **Swan** calls are added to an existing program. The program is then compiled and linked with the **Swan** Annotation Interface Library (**SAIL**). The viewer then runs the annotated program.

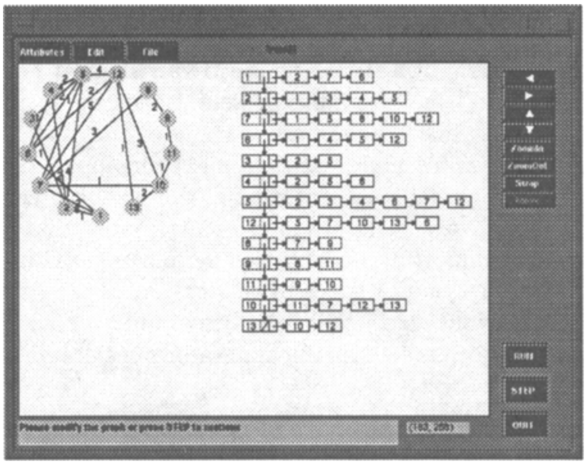
Many program visualization systems exist. See [5, 4] for examples. These have been used for teaching, presentation, and debugging purposes. The main design goal for **Swan** was to create an easy-to-use annotation library combined with a simple, yet powerful, user interface for the resulting visualization. Several features distinguish **Swan** from most other program visualization systems:

1. **Swan** provides a compact annotation library. Fewer than 20 library functions are frequently used.
2. The viewer's user interface is simple and straightforward.
3. The annotator decides the view semantics, i.e., the association between program variables and graphical elements in the views. The annotator also controls the progress of the annotated program.
4. **Swan** provides automatic layout of a graph so the annotator need only concentrate on the logical structure of the graph.
5. **Swan** allows the viewer to modify the data structure.
6. **Swan** was built on the **GeoSim** Interface Library developed at Virginia Tech, which allows **Swan** to be easily ported to X Windows, MS-DOS and Macintosh computers. It is crucial for educational software to run on a variety of operating systems that are widely used in computer science classes.

Currently, two versions of **Swan** are available: one for the X Window system and one for MS-DOS. Information about **Swan** can be obtained through the World Wide Web at URL `http://geosim.cs.vt.edu/Swan/Swan.html`.

[1] The authors gratefully acknowledge the support of the National Science Foundation under Grant CDA-9312611.

Visualization can be applied either to the physical implementation for a data structure in a program or to the abstraction represented by that data structure. For example, two views of a graph can be provided as part of an annotated minimum spanning tree algorithm. In Figure 1, the view on the right is an adjacency list representation of the graph, i.e., the program's physical implementation. The view on the left shows the logical topology of the graph, an abstraction represented by the adjacency list. Separate views of the same data structure coexist in **Swan**, each as a separate graph.



Please modify the graph or press STEP to continue

(183, 208)

Fig. 1. Two views of a graph created in an annotated minimum spanning tree algorithm

In **Swan**, information can be passed from the annotator to the viewer in the form of graphical representation of data structures. Information can also be passed from the viewer to the annotator in the form of modification requests, providing a powerful mechanism to encourage the viewer to be more active in exploring the program and gaining new insights. This capability makes **Swan** different from most program visualization systems in which the viewer can only watch the animation passively. We believe the ability to modify the program's data structures not only makes **Swan** more suitable as an instructional tool, but also shows the potential for **Swan** to be used as a graphical debugging tool at the abstract level.

2 System Components

Swan has three main components: the **Swan** Annotation Interface Library (**SAIL**), the **Swan** Kernel, and the **Swan** Viewer Interface (**SVI**). **SAIL** is a small set of easy to use library functions that allow the annotator to design different views of a program. **SVI** allows a viewer to explore a **Swan** annotated program. The **Swan** Kernel is the main module in **Swan**. It is responsible for constructing, maintaining, and rendering all views generated through **SAIL** library functions. It accepts viewer's requests through **SVI**. It is also the medium through which the annotator communicates with the viewer.

All views in **Swan** are composed of **Swan** graphs. A **Swan** graph has a set of **nodes** and **edges**. A **Swan** graph is defined by the annotator via its nodes and edges. A **Swan** graph has default display attributes for its nodes and edges, which are used by **Swan** to render the corresponding graphical objects. Nodes and edges can have their own individual display attributes that can override the graph's default values. The annotator uses **SAIL** to annotate the program; the viewer investigates the annotated program through **SVI**.

The topology of a graph is stored in the **Swan** Logical Layer. The **Swan** Logical Layer contains all the internal representations of graphs created by the annotated program. For each graph, a standard adjacency list representation is used to store all the graph's nodes and edges. After appropriate layout algorithms are applied, a physical representation of the layout is kept in the **Swan** Physical Layer. Every **Swan** graph has *physical* attributes, that affect its graphical display. The most important attribute is the position of the graph and the positions of all of the nodes and edges in this graph, that is, the *layout* of the graph. Several graph layout algorithms have been implemented in **Swan** to deal with different types of graphs so that the annotator does not need to spend time on layout himself.

Events generated by interactions between the viewer and **Swan** are sent to the **Swan** Event Handler. A **Swan** annotated program runs as a single thread process. The events generated from the **Swan** Viewer Interface are stored in an event queue. Initially the annotated program has control of the process. Whenever a **SAIL** function is invoked, **Swan** will process all events in the event queue. At this point, **Swan**'s Event Handler takes control. After the **SAIL** function completes, control is returned to the annotated program.

There are three basic states in **Swan** when it is active: **Run**, **Step** and **Pause**. Essentially, the process may run continuously (i.e., in **Run** state) or step by step (i.e., in **Step** state). "Step" here refers to the execution of a code segment ending at the next breakpoint set by the annotator. **Swan** lets the annotator decide the size of the step because it is difficult, if not impossible, for **Swan** to identify the interesting events in the annotated program.

The viewer interacts with an annotated program through the **Swan** Viewer Interface (**SVI**) as shown in Figure 1. The **SVI** main window contains a control panel and three child windows: the **display window**, **I/O window** and **location window**. The display window contains the graphs output by **Swan**. From here the viewer can get information about nodes and edges, or pan and zoom over the graph display area. The I/O window is used by the annotator and the **Swan** system to display one-line messages and get input from the viewer.

The **Swan** Annotation Interface Library (**SAIL**) is a set of easy to use functions for annotating a program so that its significant data structures and the manner in which the data structures change during the execution of the program can be visualized. Given an appropriate description of the data structures used in a program, **Swan** is able to display them using different graphical elements as specified by the annotator.

3 SwanGraph Layout Algorithms

A graph in **Swan** consists of a set of layout components. Each layout component has nodes and edges. When the graph is displayed, the layout of nodes and edges is determined by the type of the layout component they belong to. There are several algorithms implemented in **Swan** to lay out different kinds of graphs automatically. New graph layout algorithms can be integrated into **Swan** easily.

Linked lists and arrays are examples of layout components. Layout components allow an annotator to build a more complicated structure than the simple linked list or array. In **Swan**, a node in a layout component may be a parent node of another layout component. Therefore, a simple linked list can be recursively expanded to represent relatively complex structures.

Swan also contains an algorithm to draw rooted trees, based on the aesthetic criteria suggested by Bloesch [1]. These criteria include aligning sibling nodes horizontally; centering parent nodes between their leftmost and rightmost children; keeping edges from crossing; and good horizontal and vertical separation.

Swan implements two algorithms to draw general undirected graphs. The first distributes nodes along the circumference of a circle evenly (Figure 1). Edges are drawn as straight lines between its two end nodes. The second algorithm implements Kamada and Kawai's algorithm [3], which is a force-directed placement method. Here, the total balance of a layout is considered to be more important than simply reducing the number of edge crossings. **Swan** also includes a hierarchical layout algorithm for digraphs based on the procedures of Eades and Sugiyama [2].

References

1. A. Bloesch, "Aesthetic Layout of Generalized Trees", *SOFTWARE — Practice and Experience*, Vol. 23(8), August 1993, pp. 817-827.
2. P. Eades and K. Sugiyama, "How to Draw a Directed Graph", *Journal of Information Processing*, Vol. 13, No. 4, 1990, pp. 424-437.
3. T. Kamada and S. Kawai, "An Algorithm for Drawing General Undirected Graphs", *Information Processing Letters*, Vol. 31, April 1989, pp. 7-15.
4. G.-C. Roman and K.C. Cox, "A Taxonomy of Program Visualization Systems", *IEEE Computer*, Vol. 26, No. 12, 1993, pp. 11-24.
5. R. Tamassia and I.G. Tollis, Eds., *Graph Drawing'94*, Lecture Notes in Computer Science 894, Springer, Berlin, 1994.