IADIS International Journal on Computer Science and Information Systems
Vol. 1, No. 1, pp. 115-127
ISSN: 1646-3692

AlgAE: RAPID ANIMATION OF C++/JAVA CODE

Steven J. Zeil
Old Dominion University
Dept. of Computer Science

ABSTRACT

AlgAE is a framework for algorithm animation to support undergraduate education in programming. AlgAE is aimed at rapid development of animations of C++ or Java code extracted from the instructor's own notes or from the course's textbook. It can be used by instructors in a live classroom setting, by students studying or reviewing course material, or as part of internet-delivered courseware.
This paper presents an overview of the AlgAE architecture, the process of developing a new animation, and assesses the degree to which the AlgAE design supports its goal of facilitating rapid development of animations.

KEYWORDS

Algorithm animation, Simulation, E-learning.

1. INTRODUCTION

AlgAE (ALGorithm Animation Engine) is a framework for algorithm animation to support undergraduate education in programming. AlgAE is aimed at rapid development of animations of C++ or Java code extracted from the instructor's own notes or from the course's textbook. It can be used by instructors in a live classroom setting, by students studying or reviewing course material, or as part of internet-delivered courseware.

Figure 1 gives a snapshot of a typical AlgAE session. A graphic portrayal of the important data elements in a running algorithm (in this case, insertion of a value into a binary search tree) is shown, together with a listing of the source code being animated. The current execution point in the source code is highlighted. Not shown in this snapshot is a third pane that opens to permit interaction whenever input/output from the standard I/O streams is requested by the animated code.

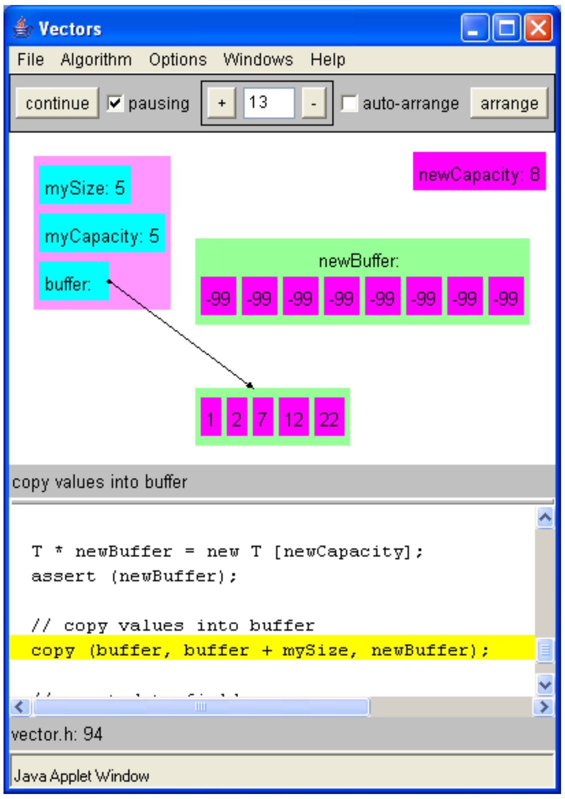
Among the design principles that underlie the AlgAE system are:

1. Animations should be tied to real, running code.
2. Animations should be tied to code from the course text and lecture notes.
3. It must be easy to create an animation from existing code.
4. The animations must be easy to run and control.
5. The viewer needs constant feedback about where the current execution point is in the algorithm.
6. One need not show many objects on the screen at once, but the objects need to be large.

To accomplish these goals, an animation designer must present AlgAE with both *structural* information about the data and algorithms being animated and *presentation* information about the desired portrayal.

A major barrier to the widespread adoption of AlgAE or any other algorithm animation package is the amount of effort required to produce new animations. This paper will assess the degree to which AlgAE has successfully managed that effort and offer suggestions as to how future animation systems could improve in this respect.

In the sections that follow, the overall architecture and design of AlgAE is described. The process involved in an instructor's creating a new animation is reviewed. Finally, the suitability of AlgAE's support for rapid development of new animations will be assessed and directions for future development outlined.



copy values into buffer

```
T * newBuffer = new T [newCapacity];
assert (newBuffer);

// copy values into buffer
copy (buffer, buffer + mySize, newBuffer);
```

vector.h: 94

Figure 1. A typical AlgAE session (implementing C++ vectors)

2. HISTORY

AlgAE evolved from a scripting language developed by the author for describing animations of Pascal data structures for use in live classroom settings. It soon became clear that scripting an animation to simulate what executing code would do was problematic.

- Small inconsistencies between the scripted graphics and the actual code behavior were distracting and misleading to students.
- Updating the scripts to reflect changes in the code being taught required significant effort and self-discipline.

At the same time, students were enthusiastic about the animations, requesting the ability to run them while studying outside of the classroom.

In response to these pressures, the first version of AlgAE, permitting web-based viewing of animations, was developed in 1997 (Zeil, 1997), subsequently redesigned (Zeil, 2001), and has been used since in numerous classes, including the author's own internet-delivered distance course on data structures and algorithms, for which a catalog of available on-line animations can be found at (Zeil, 2001a).

3. THE ARCHITECTURE OF ALGAE

3.1 Features

A typical AlgAE animation presents a menu of algorithms for manipulating a common data structure. Selecting one of these starts the algorithm. At selected points in the algorithm, the program will pause and draw a picture of the current data state. Figure 1 gives an example of an AlgAE animation session in progress.

AlgAE portrays data using a simple model of labeled boxes (possibly nested) connected by arrows. Boxes are laid out automatically, taking advantage of “hints” from the animation designer as to the desired orientation of arrows representing pointer-valued data fields. Boxes in the picture can be repositioned manually at the viewer's discretion.

Pressing the `continue` button on the toolbar (or selecting `Continue` from the `Algorithm` menu) will cause execution of the algorithm to resume and move forward to the next designated pause or until the algorithm is completed.

In terms of Brown's taxonomy (Brown,1988) of algorithm animation displays, AlgAE is a *direct*, *current* portrayal. Compared to other algorithm animation systems (Brown,1988, Brown,1991, Brown,1992, Stasko,1990), the box-and-arrow portrayal by AlgAE is rather limited, but this limitation is deliberate and consistent with its goal of illustrating the behavior of detailed code.

3.2 Design Principles

- *Animations should be tied to real, running code.* Certainly, any portrayal that can be achieved with AlgAE can also be achieved via any of a number of general-purpose animation and drawing packages. However, such animations may be no easier to develop and maintain, and lose the flexibility of allowing viewers to execute with different input data values, to interactively explore the possible behaviors of an algorithm.
- *Animations should be tied to code from the course text and lecture notes.* Instructors who are facile with graphics-rich languages and packages might consider simply writing a program to draw pictures that simulate the algorithm being discussed. However, as noted in the introduction, development of an animation as a separate script or program leads to

problems of accuracy and maintenance. Moreover, it is quite possible that the initial development of such an animation would prove more time-consuming than using AlgAE.

- *It must be easy to create an animation from existing code.* This is essential, because otherwise it would be more effective for an instructor to fall back on hand-drawn illustrations. Also, texts change frequently, and so, therefore, does the code to present to the students.
- *The animations must be easy to run and control.* This is important to students running the animations outside of class who need to be able to concentrate on the content of the lesson, not the interface to the tool. Even instructors using them in-class do not need the distraction of trying to manage a complicated interface while lecturing.
- *The viewer needs constant feedback about where the current execution point is in the algorithm.* Without this, the viewer can easily lose the relationship of the data view to the code.
- *One need not show many objects on the screen at once, but the objects need to be large.* For instructional purposes, a couple dozen objects on the screen at once seem to be more than enough. Understanding is generally enhanced by the elimination of spurious details (relying upon the judgment of the instructor to determine what is spurious and what is essential). Consequently, AlgAE does not place much emphasis on speedy rendering of large numbers of objects. On the other hand, for the text in these objects and their connecting pointers to be visible in a large lecture room or on a conventional TV monitor (conventional TV bandwidth imposes some fairly severe constraints on text presentation and line drawing), objects need to present any labeling text in a large font.

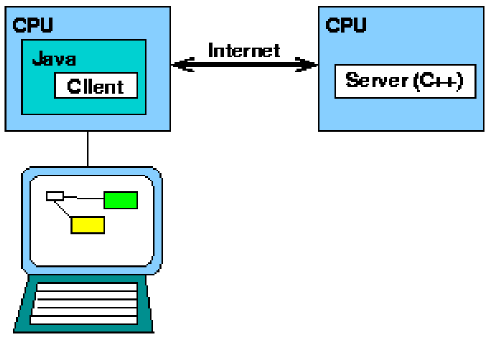


Figure 2. Client-Server for C++ animation

3.3 Web-Based Architecture

AlgAE is designed as a client-server system (Figure 2). The algorithm being animated runs on the server, which communicates with the client. The client is responsible for presenting the animation graphics and for managing interaction with the user. When Java algorithms are being animated, the client and server run together on the user's computer. But, because C++ code compiled by the instructor or animation designed might not run on a student's computer, C++ animation takes place with the C++ server running on one computer (one accessible to the instructor), communicating via a TCP/IP socket with a Java AlgAE client running on the student's computer.

To run a C++ animation from the Web, the user asks a web browser to display a page containing a call to a simple CGI script on some other machine (the “host”). This script launches the server, giving it a template for an HTML page in which to write the selected port number. The HTML page is then sent back to the user's computer. This HTML page contains an applet call to invoke the AlgAE client program to run on the user's computer, communicating via the selected port number. At the end, then, both a client and a server are running, on different computers, but sharing a common port number for their communication.

3.4 AlgAE Servers

The server in an AlgAE animation is actually rather simple. It contains conventional code for communicating via a TCP/IP socket. A simple communications protocol allows the transmission of descriptions of sets of objects for portrayal in the animation. Each object is described as 1) a unique identifier, 2) a set of text strings to appear inside the object's portrayal, 3) a list of identifiers of other objects contained inside this one, and 4) a list of identifiers of objects pointed to from this one. All objects to be portrayed in an animation must be subclasses of an AlgAE class `Visible` that, as described later, provides operations for extracting this information.

Other information that can be transmitted via the protocol include input and output text (AlgAE provides functions for prompting the student for input values during an animation, and the standard I/O streams are also automatically redirected through the client), and information/status messages to be presented to the student.

A list is maintained of ``root" objects to be shown in the animation. When a pause/frame location is reached, this list is walked and a description of each such root object and of each object reachable from the roots via the contained/pointer identifier lists is transmitted to the client.

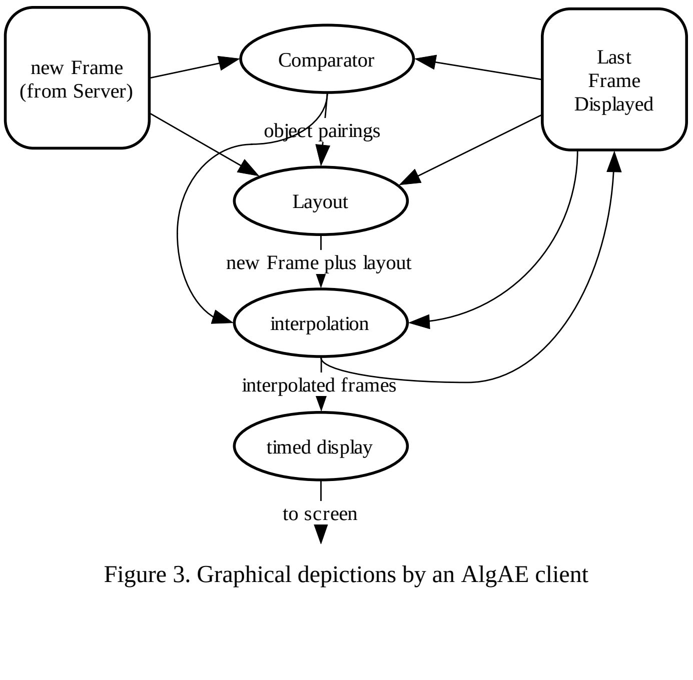


Figure 3. Graphical depictions by an AlgAE client

3.5 AlgAE Clients

The AlgAE client runs as a Java applet. It contains code to connect to a server and to accept *frames* (a set of object descriptions from a single simultaneous moment in the algorithm execution) from that server. These frames are then rendered graphically as shown in Figure 3.
A newly arrived frame is compared to the prior frame to identify pairs of unchanged objects. This information is passed to the layout manager, which assigns (x,y) positions for new objects and for objects whose size/connectivity have changed. Unchanged objects may also be assigned new positions if they are connected to objects that have moved.

AlgAE uses motion to focus the viewer's attention on the changes that have occurred. Consequently, the old and new frames are next subjected to an interpolation process to produce a sequence of frames in which changes from the old frame to the new occur gradually. In the current version of AlgAE, only object movement (caused by the layout manager's reactions to connectivity changes when pointers/references are altered) is interpolated. The technique is valuable enough, however, that other potential interpolations (e.g., smooth changes of pointer arrows, color changes, etc.) could be added in the future.
Finally, the sequence of interpolated frames is passed to a timer-driven display that renders the sequence over the period of approximately one second, halting at the new frame originally received from the server.

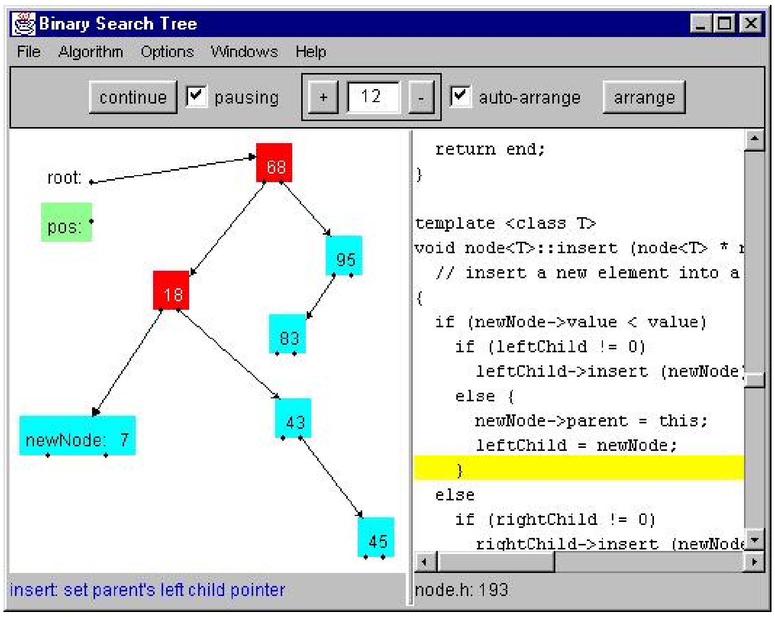
4. PREPARING AN AlgAE ANIMATION

An AlgAE animation begins with code taken from the instructor's notes or the course textbook, containing a group of one or more related algorithms (e.g., the primary operations on a particular abstract data type). A simple skeleton is written to

- declare any shared data objects to be manipulated by these operations
- invoke each algorithm to be animated inside a parameterless “menu function” that applies the algorithm to the shared data
- provide a `main` function to launch the AlgAE framework, passing it the list of menu functions together with identifying names to appear in the user interface.

At this point, the program could be run via the web, exercising the data structure with standard input and output routed through a window on the viewer's machine. However, the algorithms would not pause inside their code bodies and no pictures of the data would be drawn.

To actually produce a viewable animation, the designer must add code supplying both *structural* and *presentation* information. The *structural* information describes which data objects should be drawn and how they map onto the boxes-and-arrows model. The *presentation* information indicates where to pause the animation, what colors to employ, and what highlighting and other enhancements to add.



```
  return end;
}

template <class T>
void node<T>::insert (node<T> * r
  // insert a new element into a
{
  if (newNode->value < value)
    if (leftChild != 0)
      leftChild->insert (newNode)
    else {
      newNode->parent = this;
      leftChild = newNode;
    }
  else
    if (rightChild != 0)
      rightChild->insert (newNode
```

insert: set parent's left child pointer

node.h: 193

Figure 4. AlgAE session for binary search trees

4.1 Structural Information

Perhaps surprisingly, the structural information is quite straightforward and can often be completed with only a few minutes of programming time.

Each data type to be drawn during an animation must be made a subclass of `Visible`. (In Java, where multiple inheritance is not permitted, classes must merely implement an interface similar to what is described here.) `Visible` subclasses acquire a few new responsibilities:

- `Visible` objects must be given a default color as part of their initialization. (This would actually qualify as presentation information.)
- Each `Visible` object must provide an operation to indicate what text, if any, should be written inside its box.
- Each `Visible` object must provide an operation to "touch" any other `Visible` objects that should be portrayed as nested inside it.
- Each `Visible` object must provide an operation to "touch" any pointers to be portrayed as arrows from it to other `Visible` objects. A hint may be given for each such pointer indicating the desired direction from which that pointer emerges.

Figure 5 illustrates the code added to the binary tree nodes portrayed in Figure 4.

- `writeText` is used to provide any textual content to be placed inside the boxes drawn by AlgAE for objects of the class.

```
class node : public Visible
{
public:
  node (const T& v, node<T>* par, node<T>* left, node<T>* right)
    : value(v), parent(par), leftChild(left), rightChild(right),
      Visible(AlgAE::Cyan)
    { }

  // operations
  node<T> * copy ( node<T> *);
  void release ();
  int count (const T & testElement) const;
  void insert (node<T> *);
  int size () const;

  // Data fields
  T value;
  node<T> * parent;
  node<T> * leftChild;
  node<T> * rightChild;

  virtual void touchAllPointers() const
  {
    if (showParents) touch(parent, AlgAE::N);
    touch(leftChild, AlgAE::SSW);
    touch(rightChild, AlgAE::SSE);
  }
  virtual void touchAllComponents() const
  {}
  virtual void writeText(std::ostream& out) const
  {
   out << setw(3) << value;
  }
};
```

Figure 5. Adding structural information to a C++ class

- `touchAllPointers` is used to indicate that the object is to be portrayed as having pointers/arrows to other `Visible` objects. It does this by calling the `touch` function on each such pointer. An optional direction value (named for the points of the compass) may be supplied to indicate the preferred orientation of the emerging arrows. Judicious use of this parameter goes a long way toward giving the AlgAE-generated pictures the appropriate "look-and-feel" for any particular data structure.
- `touchAllComponents` is used to indicate that the object is to be portrayed as a compound object containing other `Visible` objects inside it.

Once the `Visible` inheritance has been arranged, the designer must call a `Visible::show()` function on one or more objects to indicate that they (and all their touched components and all objects reachable from them via touched pointers) should be drawn.

Using an inheritance-based mechanism to portray data carries one significant drawback. In both C++ and Java, many of the simple data primitives such as the numeric types are not full-fledged objects and so cannot be modified to become subclasses of `Visible`. AlgAE provides a library of `Visible` "ghost" types that can be used to portray these primitives (e.g., Figure 6). This is not an entirely satisfactory solution, however, because it does not easily extend to arrays of primitives, pointers to primitives, and other common patterns of data structure construction.

4.2 Presentation Information

The most critical part of the presentation information is where the algorithms should be paused. This is indicated by adding calls to `AlgAE::Frame(`*message*`)` at each desired breakpoint (e.g., as shown in Figure 7). Each time a `Frame` call is reached, the running algorithm on the server pauses, the AlgAE system code walks the `Visible` data structures using the "touch" functions described above, and a description of the current data state sent to the client. The client determines an appropriate layout and renders the data state as a boxes-and-arrows picture. Changes in the state since the previous rendering are highlighted by using motion to smooth out the transition from one rendering to the next. The *message* string from the `Frame` call appears in the client display's status line.

At this point, the designer has a complete animation that will show the data state at each indicated breakpoint.

The results are sometimes less than esthetically pleasing, however. AlgAE provides a number of operations for highlighting objects, changing colors, adding additional positioning hints, and otherwise decorating a basic animation (e.g., Figure 7). The author's experience is that the bulk of the time spent developing a new animation is typically in the addition of these additional decorations.

5. EVALUATION AND FUTURE DIRECTIONS

Students, particularly those enrolled in the author's internet-based distance learning courses, have been enthusiastic about AlgAE in each semester's student evaluation of teaching surveys. This suggests that the animations do fulfill a useful role in teaching.

AlgAE has been available via anonymous ftp (Zeil,2001a) since 2001. Although little effort has been made to publicize it, it has been downloaded many times. Very few people, however, have responded to the author's request for URLs of completed animations. It's hard to avoid a conclusion that AlgAE has not been successful at facilitating the rapid development of new animations.

In searching for reasons why this might be true, two problems become immediately apparent. First, the steps required to capture structural information for non-class types can often lead to awkward, time-consuming programming "workarounds"., e.g., the "ghost" classes described earlier. Second, the time spent coding highlighting and other presentation decorations often exceeds the time to get a simple working animation. For example, in the example shown in Figure 7, the presentation information actually constitutes more code than is present in the algorithm being animated.

```
{
  int i; // can’t make this Visible
  VisibleInt ghostOfi (&i);
  ghostOfi.show();
   ≈
```

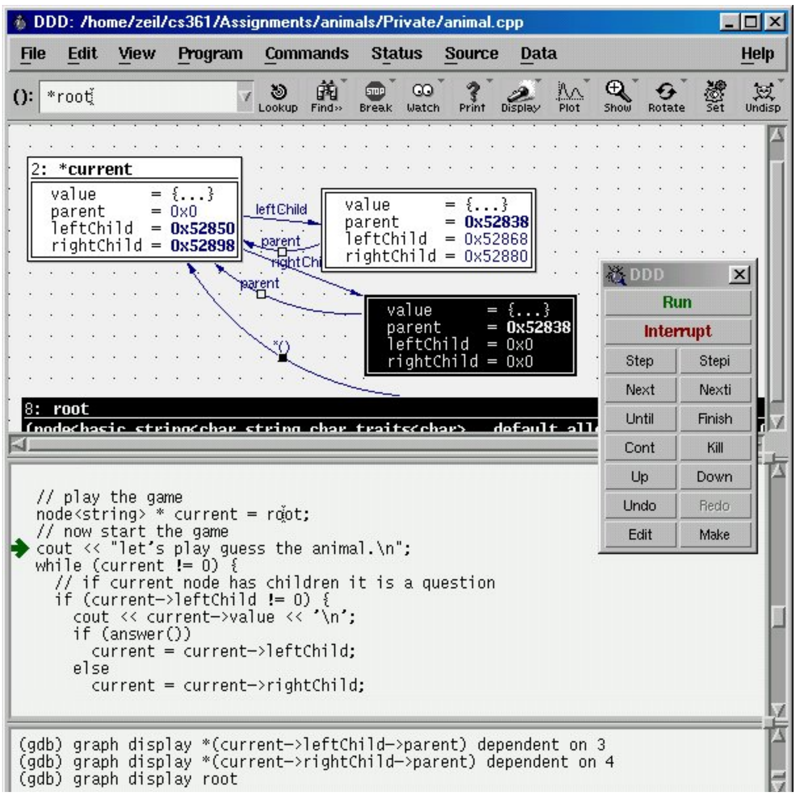
Figure 6. Showing a primitive

5.1 Deriving Structural Information from Debuggers

A possible approach to minimizing the amount of programming required to do AlgAE-like animations is to obtain the structural information required for an animation from typical run-time debuggers. Intuitively, the identification of pause/frame locations for an animation is similar to the setting of a breakpoint in a debugger. The further idea that a debugger can supply information required about the algorithm's data state becomes apparent from considering, first, the relatively simple nature of the information required and, second, the fact that typical debuggers allow evaluation of functional expressions, identification of data types, and the revelation of the data structures contained within each type. In fact, the author's inspiration for this approach came from comparing the displays of the Data Display Debugger (DDD) (Rojansky, 1997) to the displays of AlgAE. A typical DDD session is shown in Figure 7. The graphical depiction of data in this session is unmistakably similar to the connected-labeled-boxes depiction rendered by AlgAE. This suggests that much of the AlgAE server could be redesigned simply as a bridge mediating communications between an AlgAE client and a native debugger running the algorithm of interest.

```
void inOrderTraverse ( const node<int> * T )
{
  if (T != 0)
    {
      T->highlight(AlgAE::SSW); algae->FRAME("go to the left");
      inOrderTraverse(T->leftChild);
      T->unHighlight(AlgAE::SSW);
      cout << T->value << " " << flush;
      T->highlight(AlgAE::Yellow); algae->FRAME("in-order"); T->unHighlight();
      T->highlight(AlgAE::SSE); algae->FRAME("go to the right");
      inOrderTraverse(T->rightChild);
      T->unHighlight(AlgAE::SSE);
    }
}
```

Figure 7. Adding presentation info to an algorithm



```
2: *current
  value      = {...}
  parent     = 0x0
  leftChild  = 0x52850
  rightChild = 0x52898

  value      = {...}
  parent     = 0x52838
  leftChild  = 0x52868
  rightChild = 0x52880

  value      = {...}
  parent     = 0x52838
  leftChild  = 0x0
  rightChild = 0x0

8: root
```

```
// play the game
node<string> * current = root;
// now start the game
cout << "let's play guess the animal.\n";
while (current != 0) {
  // if current node has children it is a question
  if (current->leftChild != 0) {
    cout << current->value << '\n';
    if (answer())
      current = current->leftChild;
    else
      current = current->rightChild;
```

```
(gdb) graph display *(current->leftChild->parent) dependent on 3
(gdb) graph display *(current->rightChild->parent) dependent on 4
(gdb) graph display root
```

Figure 8. DDD session

Such a server would not necessarily sacrifice much in the way of portability. DDD, for example, is actually a generic front end to any of a number of different native debuggers including the GNU Debugger gab for C and C++, the Sun debugger jdb for Java, the Perl debugger, and others. Similarly, the emacs debug mode defines a common front-end interface to a substantial range of native debuggers. Together, these suggest that the structural information required by AlgAE could be supplied by a common interface to a range of different debuggers.

An animation server that worked as a debugger front end, extracting a structural description of the data state from the debugger, would avoid the inheritance/primitive problems of the current version of AlgAE. In fact, it would eliminate the need for all the highlighted code in Figure 5.

5.2 Interactive Specification of Presentation Information

As noted earlier, presentation information accounts for the bulk of the programming effort in a typical AlgAE animation development. The possibilities here are, however, easily enumerated:

1. The algorithm designer must select, from among all possible breakpoints, the ones at which the animation should pause and redraw the data state.

2. At each possible breakpoint (not necessarily limited to the ones at which the animation will pause), the designer may
    - provide a name or expression for some data object to be shown in the animation's data display or for an already displayed object that should be removed from the display.
    - provide labels for one or more displayed objects or for connectors between objects.
    - designate one or more displayed objects or connectors between objects to be highlighted or to lose highlighting.
    - designate explanatory or status messages to appear in a status line.
3. For each data type being portrayed, the designer may need to indicate
    - the default background color to be used in portraying objects of that type
    - the data members or expression values to be portrayed as internal components of the object
    - the data members or expression values to be interpreted as connectors from that object to others, and the desired direction, if any, from which that connector should be drawn as leaving the object. (Such directions are an important presentation clue. For example, the (local) pointer connectivity is the same for nodes in a binary search tree and in a doubly-linked list. Their traditional presentations differ mainly in the directions in which those pointers emerge from each node.)

It seems likely that this information could be supplied interactively by the instructor or animation designer when running the animation in a special training mode. The training mode would look nearly identical to the normal mode (Figure 1) except that at each statement the animation would pause and allow the designer to click on displayed items and specify, via a pop-up dialog box, the presentation information to associated with that item whenever an animation reaches that location.

Coupling such a “training” client to a debugger-based server would remove all but the most trivial requirements for additional coding to produce a new animation, bringing AlgAE’s goal of rapid, easy development into reach.

6. CONCLUSIONS

AlgAE has proven to be a valuable instructional tool, but still requires substantial effort to produce a new algorithm animation. The client-server model appears to be sound, allowing animation of actual running code native to one machine to be viewed from a variety of other machines. Reducing the effort required to develop new animations appears possible, but will require a substantial redesign of both the client and server. A promising approach would be to build upon conventional programming language debuggers for extraction of structural information, while allowing interactive training sessions by the instructor to capture presentation information.

REFERENCES

Brown, M. H., 1988, Perspectives on algorithm animation, Proceedings *of ACM CHI'88 Conference on Human Factors in Computing Systems*, pp. 33–38.

Brown, M. H., 1991, ZEUS: A system for algorithm animation and multi-view editing, *Proceedings of the 1991 IEEE Workshop on Visual Languages*, Oct. 1991, pp. 4–9.

Brown, M. H., 1992, *ZEUS: A system for algorithm animation and multiview editing*, Tech. Rep. 75, Digital Equipment Corporation, Systems Research Centre, 28 Feb. 1992.

Rojansky, S., 1997, Linux apprentice: DDD — database display debugger, *Linux Journal*, 42, Oct. 1997.

Stasko, J. T., 1990, Tango: A framework and system for algorithm animation, *Computer* 23, 9, Sept. 1990, 27–39.

Zeil, S. J., 1997, *AlgAE (ALgorithm Animation Engine) Programmers' Manual*, Tech. Rep. TR–97–35, Old Dominion University, Sept. 1997.

Zeil, S. J., 1999, *CS361 Algorithm Animations and Demonstrations*, http://www.cs.odu.edu/~zeil/animations.html.

Zeil, S. J., 2001, *AlgAE (ALgorithm Animation Engine) 2.0 Programmers' Manual*, Tech. Rep. TR–01–03, Old Dominion University, May 2001.

Zeil, S. J., 2001a, *AlgAE download*, ftp://ftp.cs.odu.edu/pub/zeil/algae/algae_2.0.tar.gz.