ObjectView: A Software Design Architecture for Breakpoint-Based Program Visualization

Christopher D. Hundhausen, Allen D. Malony

Department of Computer & Information Science
University of Oregon
Eugene, OR 97403

Keywords
algorithm visualization, breakpoint-based program visualization, instrumentation, object-oriented programming, portability, program object-to-graphics mapping, target-independent simulation-time visualization system, time of visualization.

Abstract
Algorithm visualization (AV) systems, which provide graphic depictions of the dynamic state of an executing algorithm, can prove invaluable to anyone interested in gaining insight into the dynamic behavior of computer programs. Although past researchers have taken a multitude of different approaches to designing and implementing such systems (cf. Myers 1984, Brown 1988, Stasko 1990, Naps & Hundhausen 1991, Tuchman, Jablonowski, & Cybenko 1991, Mukherjea & Stasko 1993), they have all had to address, in some way, the same four issues underlying AV system design: (1) *portability*, (2) *time of visualization*, (3) *instrumentation*, and (4) *program object-to-graphics mapping*. Targeting the specific domain of *breakpoint-based program monitoring and visualization*—a style of AV in which one can set breakpoints and examine the graphical representation of program's objects (variables, data structures, and other abstractions of program state) of interest at those breakpoints—this paper both identifies the requirements of a *target-independent* AV system architecture, and presents *ObjectView*, an architectural framework that we have developed to meet those requirements. Through its use of a distributed model, the object-oriented programming paradigm, and a portable, device-independent graphics language, ObjectView defines a framework for AV design that supports (1) a natural and non-intrusive method for identifying the program objects of interest, and for preparing those objects to be visualized; (2) a device- and program-independent technique for defining mappings between program objects and the graphics displays to which they should give rise; and (3) a run time source-level visualization environment for controlling the execution of a program, and for viewing, on demand, dynamic graphical representations of program objects.

1 Introduction

Recognizing that computer algorithms represent "complex objects whose properties can be difficult to fathom" (Brown & Sedgewick 1984, p. 177), computer scientists have inexorably sought tools that might make the dynamic behavior of algorithms more accessible. Predicated on the intuitive idea that we can gain insight into the dynamic behavior of an algorithm through a mapping from its fundamental abstractions to two- or three-dimensional graphics, *algorithm visualization* (AV) emerged in the 1980s as a tool for gaining insight into the dynamic behavior of computer algorithms. Although early work in algorithm visualization focused on developing methods for visualizing classic, programming-in-the-small algorithms (cf. Brown 1988, Stasko 1990), more recent research has turned to the problem of developing run-time systems for graphically monitoring potentially large and possibly parallel programs. Our work has been greatly influenced by two, largely independent lines of recent research.

First, in describing their *Vista* visualization system, Tuchman, Jablonowski, & Cybenko (1991) introduce the term *simulation-time visualization* to denote displays that may be selected and viewed at run-time—as an executing program unfolds. Vista exploits a compiler-generated symbol table and a distributed framework to achieve simulation-time animation with "minimal and possibly automatic instrumentation" of the target program, and without requiring the programmer to specify the visualization mappings before compilation. Although users may select a visualization mapping at runtime, Vista display methods are intimately tied to a small set of standard data views; thus, Vista discourages user exploration of alternative algorithm-to-graphics mappings.

In a similar vein, Mukherjea & Stasko (1993) describe a simulation-time visualization system called *Lens*. Like Vista, Lens exploits a compiler-generated symbol table to obviate the need for intrusive program instrumentation; moreover, a direct-manipulation, run-time environment provides users with a graphical editor, along with a suite of animation primitives, for interactively defining algorithm abstraction-to-graphics mappings. In order to determine which primitives to include in the Lens system, Mukherjea and Stasko examined a set of forty algorithms on which animations had already been defined using Stasko's *Tango* system (Stasko 1990). Accordingly, while the Lens system can prove invaluable for defining animations on algorithms that are similar to those that Mukherjea and Stasko studied (mostly sorting, searching, graph, tree, string, and graphics algorithms), a usability study that one author recently conducted on Lens (Hundhausen 1993) suggests that the Lens primitives may prove restrictive for defining animations in a more general setting.

In this paper, we introduce the concept of a *target-independent* simulation-time visualization system, and we present a software design framework called ObjectView that we have developed to meet the requirements of such a system. Section 2 precisely identifies the requirements of a target-independent simulation-time visualization system in terms of the four central issues that general algorithm visualization systems have had to address. In Section 3, we present the ObjectView architectural model for simulation-time visualization system design, and we assess its ability to meet the requirements set forth in Section 2. Using our prototype implementation of the ObjectView architecture as a basis, Section 4 illustrates the feasibility and benefits of the architecture in practice. Finally, in Section 5 we sketch out numerous future directions for the ObjectView project.

2 Target-Independent Simulation-Time Visualization System Requirements

Owing primarily to a variety of different target domains and applications, past researchers have taken numerous different approaches to designing an AV system (cf. Myers 1984, Brown 1988, Stasko 1990, Naps & Hundhausen 1991, Tuchman, Jablonowski, & Cybenko 1991, Mukherjea & Stasko 1993). Nonetheless, we believe that the usability and acceptance of these systems within their respective domains has often turned on their ability to deal appropriately with four issues central to the design and implementation of any AV system: (1) *portability*, (2) *time of visualization*, (3) *instrumentation*, and (4) *program object-to-graphics mapping*. Below, we use those four issues as a framework for defining the requirements of a *target-independent* simulation-time visualization system; by "target-independence," we imply the ability of a simulation-time visualization system to be applied to different target application domains and execution environments. In Section 3, the system architecture we have designed with these requirements in mind, *ObjectView*, is presented.

Issue 1: *Portability*. The portability of an AV system reflects the extent to which it is tied to a specific computer platform, programming language, and graphics language. In general, the more portable the system, the more flexibility one has in choosing the programming and graphics language in which the algorithm and its visualization can be written, as well as the computer platform on which that algorithm can be executed and visualized.

Requirement 1: *Device and language independence.* We concur with Naps (1989), who identifies device and language independence as an important feature of an AV system. Specifically, two points are important here. First, the model underpinning the AV system should not be tied to machine-specific features such as low-level graphics libraries; instead, the conceptual model should be amenable to any graphics architecture. Second, an AV system should not impose limits on the particular programming languages that may be visualized; programmers should have the freedom to program in the language of their choice, and the AV system should provide ways of visualizing those programs regardless of the language in which they are written. While it is clear that certain program abstractions transcend programming languages—and hence are amenable to language-independent visualization—the instrumentation requirements of simulation-time visualization systems can limit language-independent capabilities, as we shall see in Section 3.

Issue 2: *Time of visualization.* An AV system can show a visualization of a program either *post-mortem* or in *real-time*. Post-mortem visualization systems usually require that programmers select the program states and events of interest *before* their program executes; the program must be re-executed for each new set of program events required. Real-time visualization, on the other hand, allows programmers to control and view program states as their program unfolds. As a result, programmers can decide, on the fly, which program states to view, and can thus alter the course of a visualization based on their dynamic observations of program behavior.

Requirement 2: *Both real-time and post-mortem visualization.* Clearly, real-time visualization gives users more flexibility in visualization specification, but requires greater, and possibly more intrusive, run-time control. Furthermore, for programs that manipulate large amounts of data, real-time visualization could become tedious and cumbersome; for these programs, post-mortem analysis of a trace file may be more manageable. Since both forms of visualization are more or less appropriate depending on the situation, we conclude that an simulation-time visualization system should be flexible enough to accommodate *both* real-time and post-mortem visualization.

Issue 3: *Instrumentation*. Instrumentation refers to the mechanism by which an AV system gains access to the dynamic state of an executing program. Often, instrumentation comes in the form of procedure calls inserted into the original source code of the program to be visualized.

Requirement 3: *Minimal intrusions into the original source code.* Brown (1988) promulgated this requirement in the first sentences of his dissertation, and it has had a powerful influence on algorithm visualization systems ever since. Indeed, an AV system that allows programmers to visualize a program without having to change it very much should be preferred to an AV system that requires programmers to transform their program into something that bears little semblance to the program they originally wrote. In short, the fewer source code intrusions that are visible to the programmer, the better.

Issue 4: *Program object-to-graphics mapping*. As pointed out in the introduction, AV requires a mapping, at one or more points in a program's execution, between program objects (variables, data structures, or other abstractions of program state), and two- or three-dimensional graphics Past researchers have addressed this issue in a variety of ways, ranging from automatic, static mappings whose location in the source code must be specified at compile-time (cf. Naps & Hundhausen 1991), to user-defined mappings that may be defined and applied dynamically at runtime (cf. Mukherjea & Stasko 1993).

Requirement 4a: *Viewers should have dynamic control of mapping.* Because one of our target domains is real-time, breakpoint-based visualization, viewers should have the ability to see the graphical representation of any visualizable object on demand—whenever the program is suspended.

Requirement 4b: *Viewers should have complete flexibility in defining mapping.* An AV system should provide tools that encourage viewers to experiment with alternative ways of mapping program objects to graphics. In particular, viewers should have the ability both to define their own program object-to-graphics mappings independently of the visualization framework, and to plug a mapping into a visualization session at runtime—i.e., without having to recompile either the original source code or the visualization system. Although this requirement has been largely ignored by past visualization systems, we believe that the systems of the future should make it a top priority; indeed, recent research on the cognitive aspects of visualization suggests at least two compelling reasons for encouraging viewers to experiment freely with alternative visualization mappings:

Stasko, Badre, and Lewis (1993) argue that in order to benefit from a visualization, one "must understand both [the] mapping [from algorithm objects to computer graphics,] and the underlying algorithm on which it is based." Assuming that programmers already have an understanding of the program they wish to monitor, they maintain that an effective way of illuminating its mapping to graphics is to allow viewers to construct that mapping themselves.

Roschelle (1990) points out we should not assume that a mapping that accords well with the expert's mental model will be meaningful to everyone, as we have been prone to assume in the past. Instead, Roschelle suggests, we should shift our efforts *away from* designing visualization systems that are capable of accurately depicting algorithms as experts have come to conceptualize them, and *toward* designing systems that provide viewers with *mediational resources* with which they can experiment with their own mappings, and thereby build their own understanding of the algorithm.

3 The ObjectView System Architecture

In order to meet the design requirements of a target-independent simulation-time visualization system within the specific domain of breakpoint-based program visualization, we take an architectural approach, viewing the problem abstractly as a physical system made up of four primary components (see Figure 1): (1)*the program to be monitored*; (2) *the program monitoring environment*, with which users interact to control the execution and display of a program; (3) *the Mapper*, which maps program objects to their graphical representations; and (4) *the Viewserver*, which manages their display. Although each component constitutes a standalone and independent unit, model components must communicate extensively during a monitoring and visualization session. Consequently, message-passing plays an important role in the model. Below, we first examine the specific role of each actor in the architectural model, called *ObjectView*,[1] as well as the considerations that proved crucial to its formulation. We then look at the requirements of the message passing mechanism by which components communicate, as well as the specific messages that need to be passed among components. Finally, we consider the extent to which the ObjectView design architecture meets the requirements of a target-independent simulation-time visualization system.

3.1 The Program to be Monitored

Clearly, in order to coordinate its execution with the other components in the model, the program will need to contain some kind of instrumentation; however, in accordance with Requirement 3, we place three constraints on the nature and intrusiveness of that instrumentation. First, programmers should be required to instrument only those components of a program that they want to visualize; the "administrative" instrumentation required to interface the program with the monitoring system should be amenable to automatic generation via a preprocessor. Second, the method of identifying and instrumenting visualizable program objects should not require low-level, machine specific code; rather, the programmer should be able to express such instrumentation naturally and elegantly using the constructs of the language in which the program is written. And third, all instrumentation should be nonintrusive in the sense that program should be able to run (without recompilation) independently of the monitoring framework just as it would run without instrumentation.

[1]The name ObjectView reflects our notion of a breakpoint-based program v
system that supports graphical viewing of program objects.

Insert Figure 1 here

3.2 The Program Monitoring Environment

The program monitoring environment (the User Environment in Figure 1) provides all of the functionality that one would expect from a source-level debugger. In particular, users have the ability to (1) see their code in a scrollable window; (2) set and cancel breakpoints in that code, preferably by clicking on lines of code in the window; (3) single-step through their program; (4) tell their program to resume execution; and (5) examine the program variables and objects of their choice anytime the execution of their program is suspended.

Note that with respect to (5), an important question naturally arises: Should users have the ability to examine *every* variable in their program, as they would expect to be able to do in a debugging system, or should they be able to examine only those that they have identified as interesting—i.e., only those that they have instrumented? The answer, as we see it, depends intimately on the specific purpose for which a program is being monitored. If, for example, a program is being monitored in order to identify and eliminate a low-level bug, then clearly the ability to examine all program variables would be advantageous. If, on the other hand, a programmer is employing the monitoring system to obtain a high-level picture of the dynamic behavior of a program, then the ability to examine all variables would be, at best, unnecessary, and, at worst, a distraction.

In designing the ObjectView model, we assume the latter perspective: namely, that while it may be desired in certain situations, the ability to examine all program variables will be unnecessary in all but the lowest-level debugging activities. Furthermore, as we shall see in the next section, to supply users with such functionality would detract from the relative simplicity of our instrumentation, and thereby violate Requirement 3. While our model should be able to accommodate such functionality if the need for it arises in the future, in this paper we shall not consider the challenging issues related to giving programmers the ability to examine all program variables.

3.3 The Mapper

The Mapper is responsible for converting execution-time representations of a *particular* visualizable program object to its graphic representation. Thus, although not visible in Figure 1, a separate mapper may exist for *each* class of visualizable objects. In devising this component of the model, we had to pay particular attention to Requirement 4b: *Viewers should have complete flexibility in defining mappings*. The Mapper responds to that requirement by decoupling the graphics routine used to specify a program object-to-graphics mapping from both the program in which those objects reside, and the environment in which those objects are ultimately displayed.

3.4 The Viewserver

The Viewserver manages the display of the graphic representations of program objects produced by the Mapper. In order to meet requirement 4a— *dynamic control of visualization*—the Viewserver must provide for a way to interface with a user-defined mapping routine at runtime, as well as a way for users to request the graphical depiction of any visualizable program object whenever the program is suspended.

Insert Figure 2 here

3.5 Message Traffic

Figure 2 shows the ObjectView physical model augmented with the messages by which model components communicate. Although we have attempted to give messages descriptive names, further comments on each message are appropriate. In the following subsections, we group messages according to their source and destination model components.

3.5.1 User Environment to Instrumented Program

These messages inform the program of user actions that affect the program's execution. *setBreak* and *cancelBreak* tell the program's *breakpoint manager* to set or cancel a breakpoint at the specified line. *step* tells the program to execute a specified number of lines, while *go* tells the program to execute to the next breakpoint. Finally, *halt* informs the program that the user does not wish to continue monitoring the program; accordingly, the program should stop execution immediately.

3.5.2 Instrumented Program to User Environment

Two types of messages need to be sent from the program to the User Environment: those that indicate which objects are currently in scope, and those that indicate what execution state the program is presently in. The *inScope* and *outOfScope* messages inform the User Environment that a particular visualizable object has just come into or left the current scope. The *suspendedAt* and *algComplete* messages, on the other hand, let the User Environment know how far along the program is in its execution: *suspendedAt* says the program is suspended and waiting at a particular line number, while *algComplete* says that the program has completed execution.

3.5.3 User Environment to Viewserver

Within the model, visualizable program objects are grouped into *classes*, with a single *class viewer* responsible for displaying all visualizable objects belonging to a particular class. Assuming that the appropriate class viewer has been opened, users may choose to view any visualizable object that is within the current scope of the program. Thus, the main messages from the User Environment to the Viewserver are sent both in response to user requests to open or close a class viewer (*openClassViewer* and *closeClassViewer*), and in response to a visualizable object's entering or leaving the current scope of the program (*addInstance* and *removeInstance*). Notice that the latter pair of messages can only be sent if the user has opened a class viewer on the class to which the visualizable objects belong.

The *suspendUserEvents* and *resumeUserEvents* messages tell the Viewserver whether it should accept user events (i.e. mouseclicks). The User Environment sends a *suspendUserEvents* message immediately before the program begins or resumes execution; a *resumeUserEvents* message is sent when the program has suspended and control has been transferred to the User Environment. Finally, the *closeVS* message informs the Viewserver that either the user has chosen to abort the monitoring session, or the program has run to completion; in either case, the Viewserver must shut itself down.

3.5.4 Viewserver to User Environment

The *classViewerClosed* is sent in response to the user's request (via a mouseclick) to close a class viewer on a particular class. Since users interact with the Viewserver directly—and, more specifically, with class viewers—the User Environment has no way of knowing that the user decided to close a class viewer without the explicit notification that the *classViewerClosed* message provides.

3.5.5 Viewserver to Instrumented Program

The *show* message is potentially sent in response to two distinct events: (1) when the user chooses to open a view a particular visualizable object in a class on which a class viewer has been opened; and (2) when the User Environment informs the Viewserver (with a *resumeUserEvents* message) that the program has suspended. In the former case, the *show* message is used to generate a picture of the object whose visualization the user requested. And in the latter case, the *show* message is used to *update* the displays of all objects currently being displayed, so that they reflect the current state of the program.

3.5.6 Instrumented Program to Mapper

The *mapsnapshot* message is always sent in response to a *show* message, which, as described above, requests that a particular visualizable object send a current graphical snapshot of itself to the Viewserver. The *mapsnapshot* message commences the message passing necessary to fulfill that request by passing a textual representation of the visualizable object of interest (i.e., whose graphical representation was requested by the *show* message) to the Mapper associated with that object.

3.5.7 Mapper to Viewserver

The Mapper responds to an *updateView* message by mapping the textual snapshot associated with the *updateView* message to an appropriate graphical depiction, which it passes to the Viewserver via the *updateView* message. Note that the Viewserver should be able to convert the *graphicalsnapshot* parameter directly to graphics that can be displayed within the Viewserver.

3.5 Discussion

Does the architecture presented above meet the requirements of a *target-independent* simulation-time visualization system? Below we examine the architecture vis-á-vis each requirement.

Requirement 1: *Device and language independence.* That we were able to describe the architecture independently of any specific computer platform clearly indicates the architecture's device independence; indeed, the architecture's only operating system requirement—an environment that supports multiple processes and a message passing style of inter-process communication—is widely available and popular. Similarly, assuming that the required message passing machinery is in place, we see no inherent reason why programs written in any programming language could not be integrated into the ObjectView framework; however, as we shall see in the following section, if we consider the elegance and ease with which programs to be visualized can be instrumented, then object-oriented languages, which support an object-based, message passing style of computation, are clearly more amenable to the architecture than procedural or functional languages.

Requirement 2: *Both real-time and post-mortem visualization.* It should be clear that the architecture facilitates real-time visualization; it does this by allowing *show* messages, which give rise to a corresponding graphical depiction in the Viewserver, to be issued anytime the executing program is suspended—i.e., in real-time. While it may not be readily apparent that the architecture also supports post-mortem visualization, we contend that post-mortem visualization fits quite naturally with the architectural framework. Without modification of the underlying architecture, *graphicalsnapshot* parameters (accompanying the *updateView* message) can be directed to a trace file instead of, or in addition to, the Viewserver. A standalone, post-mortem AV system similar in spirit to the GAIGS system described in (Naps & Hundhausen 1991) can then be used to replay trace files created by the redirected *updateView* messages.

Requirement 3: *Minimal intrusions into the original source code.* Recall that this requirement does not impose a limit on the *total* number of source code intrusions required to instrument a program; rather, it limits the number of intrusions that are *visible* to the programmer. Therefore, instrumentation that can be done automatically and invisibly is not addressed by this requirement. As suggested by Figure 2, upon execution of each line or block of code, the breakpoint manager is responsible for checking whether a breakpoint exists, and for transferring control of the program to the User Environment (via a *suspendedAt* message) if one does. We see, then, that some kind of control statement is required immediately before each line or block of executable code in order to interface the program with the ObjectView framework. Notice that such control statements amount to nothing more than a call to a procedure capable of checking the current line number against a table of breakpoints. Because this instrumentation is quite uniform, requiring nothing more than a knowledge of the number of the line that is about to be executed, it lends itself nicely to automation via a preprocessor; thus, it is invisible to the programmer and therefore nonintrusive.

One other kind of instrumentation is needed to identify visualizable objects, and to interface those objects with the architectural framework. In particular, visualizable objects must be capable of (1) informing the User Environment that they are in scope or out of Scope (cf., the *inScope* and *outOfScope* messages in Figure 2), and of (2) responding to *show* messages with an appropriate textual representation of themselves (cf., *mapSnapshot* message in Figure 2). As it requires only a knowledge of when an object enters and leaves scope, the instrumentation needed to handle (1) can, in the worst case, be automated by a preprocessor, and, in the best case, be obtained "for free" via the native constructs of the programming language (as we shall see in Section 4.2); in either case, it will not be intrusive according to our definition. The instrumentation required to handle (2) involves both interrogating a visualizable object's current state, and mapping that state to a textual representation; unfortunately, such instrumentation is intrusive: programmers must write it by hand. Nonetheless, by successfully eliminating all but one form of potentially intrusive instrumentation, the ObjectView architecture arguably meets the requirement of minimal intrusiveness.

Requirement 4a: *Dynamic control of mapping.* Within the architecture, a *show* message, which invokes a program object-to-graphics mapping, may be sent from the Viewserver anytime the program is suspended. Since viewers can have complete control of when and which show messages are sent (e.g., through a user interface to the Viewserver), the architecture supports the dynamic control of mapping by viewers.

Requirement 4b: *Complete flexibility in defining mapping.* Recall that by "flexibility in defining a mapping," we mean that viewers should have the ability to define their own mappings without having to recompile any component of the AV system. Since the Mapper—responsible for rendering a textual description of a visualizable object into a corresponding graphical depiction—constitutes a *distinct* component in the architecture, viewers have the ability (assuming they are provided with appropriate tools for interfacing with the message passing system) not only to write their own mapping routines, but also to plug such routines in at runtime without recompilation of the entire system.

4 The ObjectView Design Architecture in Practice

Although we have already shown that ObjectView holds the promise of meeting the requirements of a *target-independent* simulation-time system, the architecture will clearly be of little use in practice unless it can be easily implemented as a working software system. In this section, we use a prototype version of the ObjectView architecture to illustrate its feasibility and benefits in practice.

4.1 Overview of the prototype

In order to realize the ObjectView model, our prototype system makes use of the C++ programming language, UNIX domain *socketpair* interprocess communication, and an XWindow-based version of the international standard Graphics Kernel System (GKS). The prototype achieves complete independence and modularity of model components by mapping each component to a separate UNIX process. Layered on top of UNIX's *socketpair* IPC interface, a C++ stream-style *protocolManager* abstract base class provides the foundation for the message-passing protocol; descendant classes cater the *protocolManager* to the individual message-passing needs of each model component. A C++ *breakpointmanager* class handles the "administrative" instrumentation requirements of a program to be visualized; it defines a *monitor* method, which, when inserted before each line of executable code in a program, appropriately interfaces that program with the ObjectView architecture. An abstract base class called *visualObject* implements the basic instrumentation requirements of a visualizable object; by inheriting from the *visualObject* base class, programmers can identify and instrument program objects they wish to visualize using the natural object-oriented constructs of the C++ language, as shall be illustrated in Section 4.2. While, in our present implementation, the User Environment is implemented as a text-based, command line interface, our Viewserver uses the C interface to XGKS, a device-independent, high-level graphics language, to implement a graphical environment for displaying program objects. Finally, our present prototype does not implement the Mapper as a separate UNIX process; instead, mapping routines, which are written using the C interface to GKS, are included directly into the Viewserver source code.

A visualization session using our prototype proceeds as follows. Users launch the environment by running the User Environment executable code. From the User Environment, users may select an executable program to monitor; acting as the parent process, the User Environment then employs the UNIX *fork* and *execvp* commands to spawn subprocesses for both the program to be monitored (the "Instrumented Program" in Figures 1 and 2), and the Viewserver, which is then responsible for managing and displaying visualizable objects for that program. As long as the program to be monitored has been properly instrumented (as described below), it will be able to communicate with the other model components to facilitate simulation-time visualization within the ObjectView framework.

4.2 Using the ObjectView prototype to visualize an algorithm: An illustrative example

To make the prototype system described above more tangible, we present a simple example that illustrates the three step process one must follow to prepare an program for visualization within our prototype framework.

Step 1: *Using multiple inheritance to define a visualizable object.* Multiple inheritance, which allows programmers to combine the behavior of two or more C++ classes, provides a natural and elegant way to define a visualizable object within our prototype framework. The foundation of our multiple inheritance scheme is the *visualObject* abstract base class presented in Figure 3; by properly overriding that class, programmers can use multiple inheritance to interface any existing C++ class that they wish to visualize with our prototype framework.

Recall that our architecture identifies three specific instrumentation requirements for visualizable objects: (1) notifying the User Environment when a visualizable object enters scope; (2) notifying the User Environment when a visualizable object leaves scope; and (3) sending an execution-time representation of a visualizable object's current state to the appropriate Mapper in response to the *show* message. Because, according to the semantics of C++, *visualObject's* constructor and destructor will be invoked automatically when descendant instances enter and leave scope, our implementation transparently satisfies requirements (1) and (2) by placing the appropriate *inScope* and *outOfScope* message passing calls directly within *visualObject's* constructor and destructor. To satisfy requirement (3), we require programmers to override the pure virtual *show* method so that it interrogates the object's state, and writes a textual representation of that state to the Mapper. Figure 4 illustrates how one can create a visualizable stack class by using multiple inheritance to specialize the behavior of an existing stack class; as can be seen, the only real work in this step is to override *visualObject's* virtual *show* method.

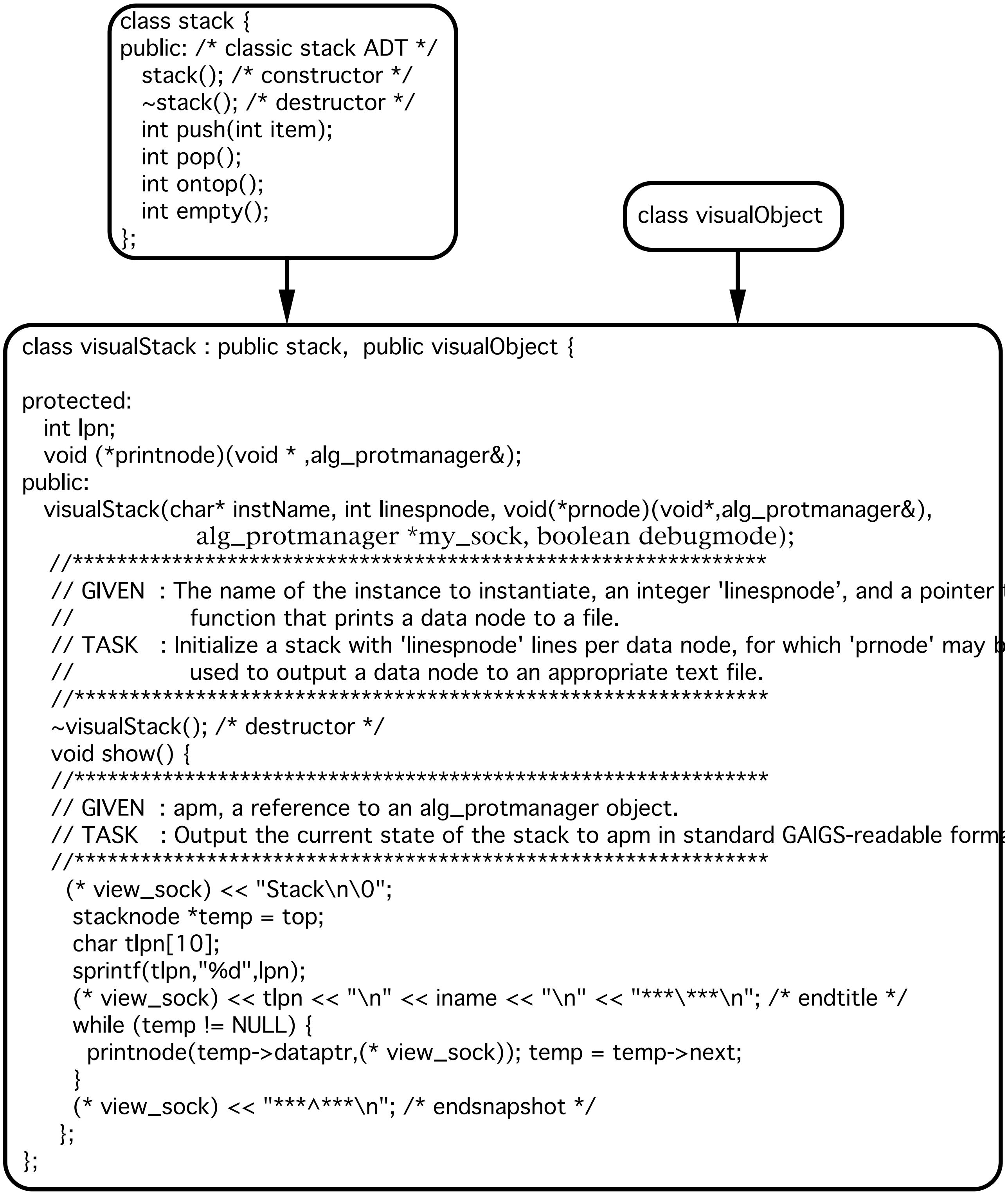
```
class visualObject {
  protected:
  char iname[64];                        /* The name of this instance. */
  char cname[64];                         /* The name of the class to which insta
   alg_protmanager *view_sock; /* Pointer to the UNIX pipe connection. */
  boolean dmode;                          /* Is the program running in debug mo

  public:
   /* Constructor--sends the User Environment an inScope message. */
   visualObject(char* instanceName, char *className,
                        alg_protmanager *my_sock, boolean debugmode);
   /* Destructor -- sends the User Environment an outOfScope message. */
   ~visualObject();
   /* This pure virtual method should use the insertion operator (<<)
        defined on the alg_protmanager class to write the textual snapshot
        over the view_sock that has been bound to this instance. */
  virtual void show() = 0;
};
```

Figure 3. C++ abstract base class visualObject, which programmers override to define a program object they wish to visualize, and to interface that object with the ObjectView framework.

Step 2: *Defining a mapping procedure.* The next step is to write a routine that can both interpret the textual format in which the *show* method defined in Figure 4 writes the stack's current state, and produce a corresponding graphical representation of the stack. While the precise details of such a routine are beyond the scope of this example, the routine involves just two steps: parsing the textual format into an internal data structure, and using that data structure to make the appropriate GKS graphical calls to draw the stack.

Step 3. *Instrumenting the program to be monitored.* Figure 5 summarizes the relatively simple "administrative" instrumentation required to interface a program with our prototype framework. Indeed, instrumenting a program amounts to nothing more than instantiating a *breakpointManager* class object, and inserting calls to the *breakpointManager*'s *monitor* method before each line of code. With this instrumentation in place, the program can be executed either as a subprocess of the User Environment, in which case all *visualObject* descendants can take visual form in the Viewserver, or as a standalone program, in which case it executes normally—just as it would without the instrumentation.



```
class stack {
public: /* classic stack ADT */
   stack(); /* constructor */
   ~stack(); /* destructor */
   int push(int item);
   int pop();
   int ontop();
   int empty();
};
```

```
class visualObject
```

```
class visualStack : public stack,  public visualObject {

protected:
   int lpn;
   void (*printnode)(void * ,alg_protmanager&);
public:
   visualStack(char* instName, int linespnode, void(*prnode)(void*,alg_protmanager&),
                     alg_protmanager *my_sock, boolean debugmode);
    //*************************************************************
    // GIVEN  : The name of the instance to instantiate, an integer 'linespnode', and a pointer t
    //            function that prints a data node to a file.
    // TASK   : Initialize a stack with 'linespnode' lines per data node, for which 'prnode' may b
    //            used to output a data node to an appropriate text file.
    //*************************************************************
    ~visualStack(); /* destructor */
    void show() {
    //*************************************************************
    // GIVEN  : apm, a reference to an alg_protmanager object.
    // TASK   : Output the current state of the stack to apm in standard GAIGS-readable form
    //*************************************************************
      (* view_sock) << "Stack\n\0";
       stacknode *temp = top;
       char tlpn[10];
       sprintf(tlpn,"%d",lpn);
       (* view_sock) << tlpn << "\n" << iname << "\n" << "***\***\n"; /* endtitle */
       while (temp != NULL) {
         printnode(temp->dataptr,(* view_sock)); temp = temp->next;
       }
       (* view_sock) << "***^***\n"; /* endsnapshot */
     };
};
```

Figure 4. Using mutiple inheritance to implement a *visualStack* class.

```
#include “alg_protmanager.h” /* Monitored programs include the alg_prot
#include "bpmanager.h"            /* header file, the bpmanager.h header file,
#include "visualStack.h"          /* header files of the the visualObject descer
. . .                                           that the program will use.*/
. . .
bpmanager *bpman;      /* An instance of the breakpointmanager class */
boolean debug_mod       /*  the global variable debug_mode tells the progra
. . .                              it is being monitored in ObjectView. */
. . .
void main() {
    /* Instantiate alg_protmanager object */
   alg_protmanager *alg_sock = new alg_protmanager();
  if (alg_sock->checkIfChild()) { /* See whether the program is being monitore
   debug_mode = true;                    /* If yes, set the debug_mode to true and
                                                the breakpointmanager object. */
     bpman = new bpmanager(alg_sock,10); /* The second parameter indicates
                                                         lines of source code in the
                                                      monitored */
 }
 else {
    debug_mode = false; /* debug_mode = false indicates normal execution outs
                                   ObjectView monitoring framework. */
   delete alg_sock;       /* We won't need it. */
   alg_sock = NULL;
 }
/* From here, each line of executable code should be proceeded by a call of t
  if (debug_mode == true) bpman->monitor(26);   /* monitor’s single paramet
                                                                the line number that
                                                           executed.  */
. . .
}
```

Figure 5. The instrumentation required in a program to be monitored within ObjectView

4.3 Practical benefits of the architecture

We believe that our prototype system demonstrates a good match between the implementation requirements of the architecture, and the implementation tools (e.g. UNIX, socketpair IPC, C++, and GKS) with which we realized the architecture; nonetheless, the high level of abstraction at which we specified the architecture in Section 3 afforded us much flexibility in choosing implementation tools. For example, we might have chosen to implement the message passing system using an interpreted language like Tcl (Ousterhout 1993) instead of our C++ stream interface to UNIX socketpair IPC. Similarly, users would not necessarily have to write mapping routines in a textual graphics language like the C interface to XGKS that we chose; instead, a graphical user interface, through which users could specify a mapping by direct manipulation, could be used to generate automatically the executable code required to perform a mapping. In short, because the ObjectView architecture addresses the requirements of simulation-time visualization using an abstract physical model whose components' interrelations are precisely identified, the architecture allows us to plug in an alternative implementation for any model component—User Environment, Instrumented Program, Viewserver, Mapper, or message passing system—without having to modify the other components in the model. Therefore, we believe that the architecture provides an ideal testbed on which to study the usefulness and benefits of simulation-time visualization in practice; we look forward to exploring the tradeoffs of alternative implementation strategies as our prototype system matures.

5 Conclusion and Future Work

In the preceding paragraphs, we have used four issues—portability, time of visualization, instrumentation, and program object-to-graphics mapping— central to AV system design as a framework for identifying the requirements of a *target-independent* simulation-time visualization system within the specific domain of breakpoint-based program visualization. The ObjectView architecture we have described not only meets those architectural requirements, but also lends itself to implemention using a variety of alternative implementation tools. Indeed, our current ObjectView prototype based successfully demonstrates the benefits of developing a program visualization system using a formal design framework.

In future activities, we intend to pursue three lines of research work. First, we want to conduct usability studies with the ObjectView prototype to evaluate user-level benefits of the design architecture. To do so will require improvements in the User Environment interface and the implementation of the Mapper as a separate component. Second, want to enhance the mapping procedures and graphics rendering capabilities of the ObjectView system by integrating other program execution analysis and view abstraction tools we have developed for parallel performance visualization. Finally, we want to use the ObjectView architecture and prototype as the basis for developing a breakpoint-based program visualization system for a parallel C++ language and execution environment.

References

Brown, M.H., & Sedgewick, R. (1984). A system for algorithm animation. *ACM Computer Graphics (Proc. SIGGRAPH '84) 18* (3), 177–186.

Brown, M.H. (1988). *Algorithm animation.* Cambridge, MA, MIT Press.

Hundhausen, C.D. (1993). Exploring the potential for Conversational Analysis in the evaluation of interactive algorithm visualization systems. Unpublished master's final project, Department of Computer and Information Science, University of Oregon.

Mikherjea, S., & Stasko, J.T. (1993). Applying algorithm animation techniques for program tracing, debugging, and understanding. *Proc. 15th IEEE Intl Conf. on Software Engineering* (Baltimore, MD).

Myers, B.A. (1983) Incense: a system for displaying data structures. *ACM Computer Graphics 17* (3), 115-125.

Naps, T.L. (1989). Design of a completely general algorithm visualization system. *Proc. 22nd Annual Small College Computing Symposium* (Eau Claire, WI), 233–241.

Naps, T.L, & Hundhausen, C.D. (1991). The evolution of an algorithm visualization system. *Proc. 24th Annual Small college Computing Symposium* (Morris, MN), 259-263.

Roschelle, J. (1990) Designing for conversations. Paper presented at the *AAAI Symposium on Knowledge-Based Environments for Learning and Teaching* (Stanford, CA).

Stasko, J.T. (1990). Tango: a framework and system for algorithm animation. *IEEE Computer* (September), 27-39.

Stasko, J.T., Badre, A., & Lewis, C. (1993). Do algorithm animations assist learning? An empirical study and analysis. In *Proc. INTERCHI '93* (Amsterdam, The Netherlands).

Ousterhout, J. K. (1993). *An Introduction to Tcl and T*k. Reading, MA, Addison-Wesley.

Tuchman, A., Jablonowski, D., & Cybenko, G. (1991). A system for remote data visualization. Center for Supercomputing Research and Development Report 1067, University of Illinois.