Improved Algorithm Visualization and Witness Detection in Educational Fusion

by

Joshua E. Glazer

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2000

Author ....
Department of Electrical Engineering and Computer Science
August 28, 2000

Certified by ....
Seth Teller
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Improved Algorithm Visualization and Witness Detection in Educational Fusion

by

Joshua E. Glazer

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Educational Fusion is a collaborative, Java based, software learning environment tailored to teaching algorithms and running simulations and virtual labs over the web. Students are presented with a reference algorithm which they are supposed to implement, and then given a chance to explore, code and animate their own version of the algorithm.

This thesis details the development of a witness detection system and improvement of the existing algorithm visualization tools. The witness detection system allows students to test the correctness of their solutions by asking the system to search for small proofs of incorrectness in their algorithm's output. The new visualization system allows developers to create modules using animated data so that student and reference algorithms automatically animate, with little work from the developer and no work from the student. Finally, the new system also allows Fusion to support simulations of physical systems and create connections and interface to actual physical devices, helping students explore these phenomenon in a consistent, collaborative environment.

Thesis Supervisor: Seth Teller
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

I must begin by thanking Professor Seth Teller for all the help, speedy feedback and encouragement he has given me throughout this production. I am amazed that, considering the many extensive projects he manages, he always found time to meet with me and answer my e-mail almost immediately. I also thank the previous Fusion team for their solid foundation, especially Bhuvana Kulkarni who took time away from her own work to help me get started on mine. I'd also like to give a good luck wave to Wes Chao who will be taking over for me as FusionMeister in the year to come.

Then, since this document will be microfilmed and archived away on some dusty MIT shelf, probably becoming the longest surviving artifact I will ever generate (until I produce my next full length digital movie; watch for it in 2004) and since it will probably contain my last official chance at a "thank you" spiel (until I accept my academy award for the aforementioned production) I'd like to go ahead and thank all the people who have contributed significantly to my life up to this point.

Unfortunately, I have to keep this paper to only 108 pages, so I'll have to skip to the most important ones. Thanks mom and dad for blowing your whole retirement wad on my education and loving me and stuff. Thanks to my grandparents for loving my parents (and me) and bringing them up well enough to know that they should spend lots of time and money on me. And thanks to my sister for being born 9 years after I was, with the great result that we never get in silly sibling rivalry fights (and that I can beat you at anything except basketball and burping contests). Thanks to Professor Steve Copping, my Ju-Jitsu Sensei in 9th grade who taught me that pain is my teacher (and therefore helped me appreciate MIT!). Thanks to all my friends and family who've always been there for me (and also spent money on me) and last of all, thanks to Dianne, who not only lent me her name and her time to help me complete this huge project, but also her heart.

Happy Day!

Contents

List of Figures

Chapter 1

Plug And Play Education

Hurtling towards the year 3000, our society becomes more digital every day. As life revolves more around the "desktop" and the "dot com," education is just another field to jump on the bandwagon of computerization. Homework assignments are more complex and more PC dependent then ever before and the trend continues. The computer itself is now a meta-tool, as it teaches its users how to better use it. This thesis attempts to aid computers in this task by improving upon current education technology. It describes the implementation of a witness detection system which allows developers to easily build detectors that search for errors in student submissions. This thesis also describes a visualization system which allows students to explore computer algorithms, physically based simulations and virtual laboratories through animated representation of data, online collaboration, and a consistent and friendly user interface.

As useful and education enhancing a utensil as the computer is, the term "computer based learning environment" has long struck fear into the hearts of guinea pig high school and college students everywhere- or at least into the heart of this student. What computer lab veteran does not remember the horrors of being unable to do her homework because the brand new environment in which she is supposed to learn refuses to exist for more than three minutes without crashing? Or worse yet, what Brand X computer owner did not loathe doing all his homework on his hall mate's machine because his was not fast enough, new enough or running the correct platform to support some Learning Software of the Future?

The overhead associated with getting a computer learning environment up and running has traditionally been daunting and sometimes not even worth the gain offered by the environment. However, all is not lost in the age of digital education. There is a savior on the horizon which, with a little luck, will change the world of education forever and solve the problem of computer based learning. Educational Fusion is that savior.

1.1 Educational Fusion

Educational Fusion is a self-contained software learning environment which allows students in various physical environments to learn through collaboration, interaction and visualization. It requires a computer equipped with a Java-compliant web browser and a web connection. Although readily adaptable for any type of material, Fusion is currently tailored to teach computer algorithms such as those which might be found in a computer graphics course, or an introduction to algorithms course. In addition, Fusion is also a universal and efficient host environment for computerized simulations which may have previously been limited to a specific platform or a non-collaborative implementation [Tel+98].

The Fusion applet supports three distinct user classes, all of which have different functions, rights and experiences within the system. These classes are Student, TA and Professor/Developer and the next sections will describe the system as viewed from each of their perspectives.

1.1.1 Educational Fusion: Learning Tool

When running Fusion, the user is presented with a login screen on which she enters her username and password. She is then identified as a student or teacher. For the purpose of this section, we will assume she is a student and her name is Dianne.

Next, Dianne is presented with a default startup screen from which she can load her desired concept graph. A concept graph is a data flow diagram, showing an

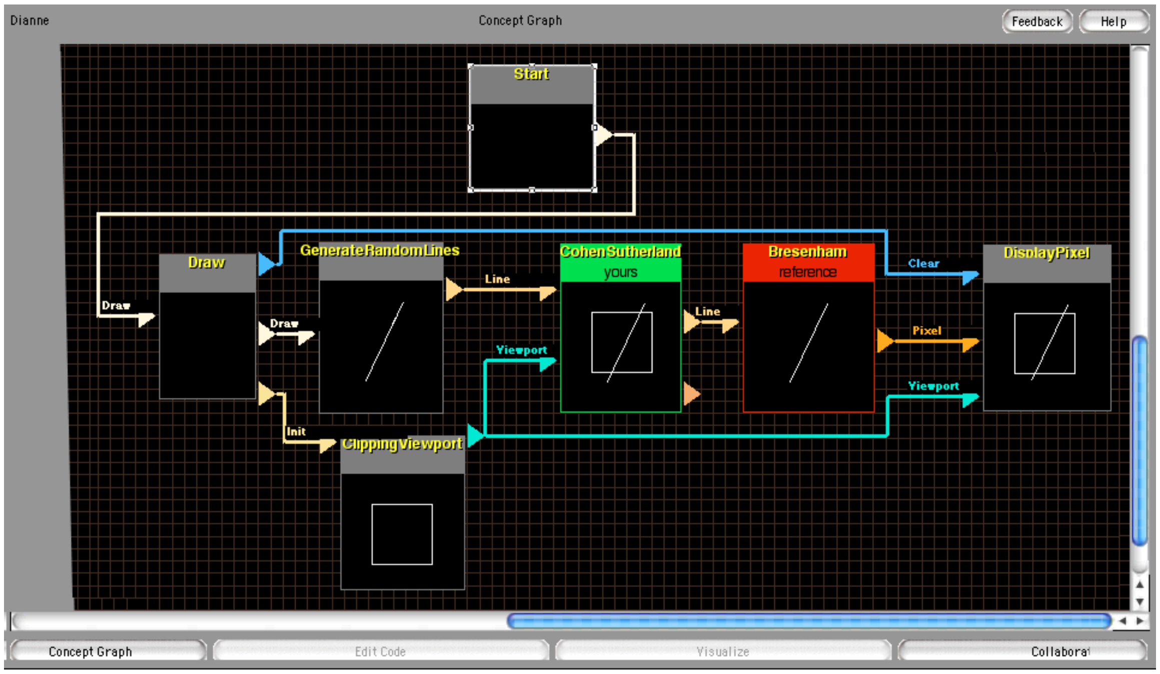


Figure 1-1: A sample concept graph

overview of how individual algorithms work together to complete a task. The concept graph for a line rendering procedure is shown in Figure 1-1. On the graph, each algorithm through which the data pass is represented by a *module*. A module contains all the code and information necessary to teach the algorithm to Dianne and looks like a green or red rectangle (as in Figure 1-1). Gray rectangles represent processes present in the data flow which are not designed to be taught to the students. When a module is red, it represents a reference version of the algorithm, included to show a student how the algorithm works when functioning. With a click, Dianne can switch a module to green, representing her current implementation of the algorithm.

Dianne can run the concept graph to watch the entire procedure in action, or she can click on one of the modules to attempt to implement the algorithm as required. After selecting an algorithm, she can edit the code of her implementation in the included text editor (Figure 1-2). This editor supports parenthesis balancing, revision control and a host of other features found in advanced development environments. When she is satisfied with her latest attempt, or just curious about the internal actions of the algorithm, she can switch to a special *visualization panel* to explore

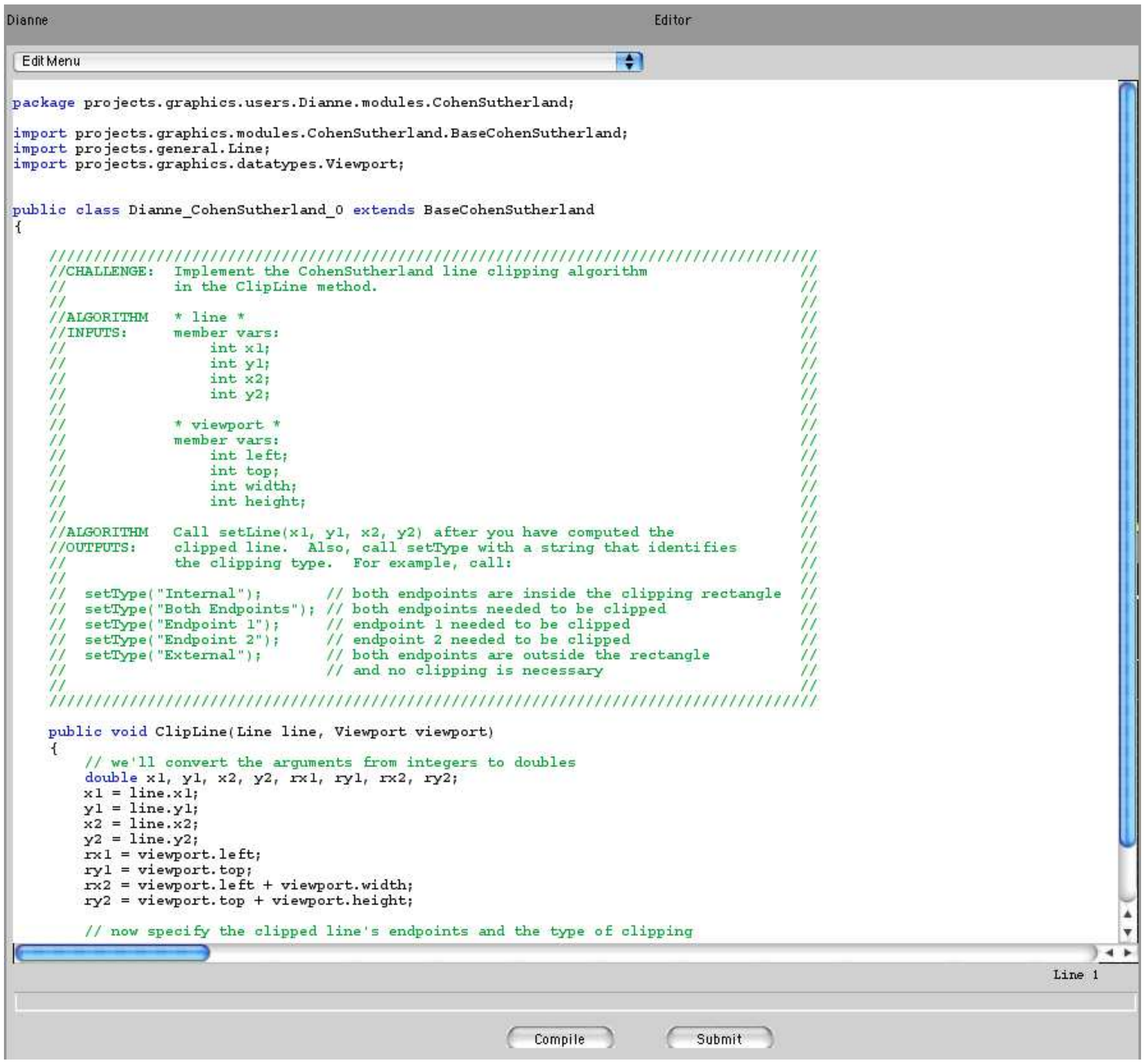


Figure 1-2: The fully functional text editor built into Educational Fusion

the algorithm further. The visualization panel (see Figure 1-3) is a dedicated area of the environment in which she can watch an animation of either the reference implementation or her own, in an attempt to better understand the workings of the algorithm. In addition, she can enable *witness detection*, asking the system to find and display *witnesses*- small, visual proofs that her implementation of the algorithm is incorrect and needs more work. This visualization panel is one of the focuses of this thesis and will be discussed in detail later.

At any point throughout her learning and programming, Dianne can switch to a chat mode (see Figure 1-4) and partake in various levels of collaboration. She can

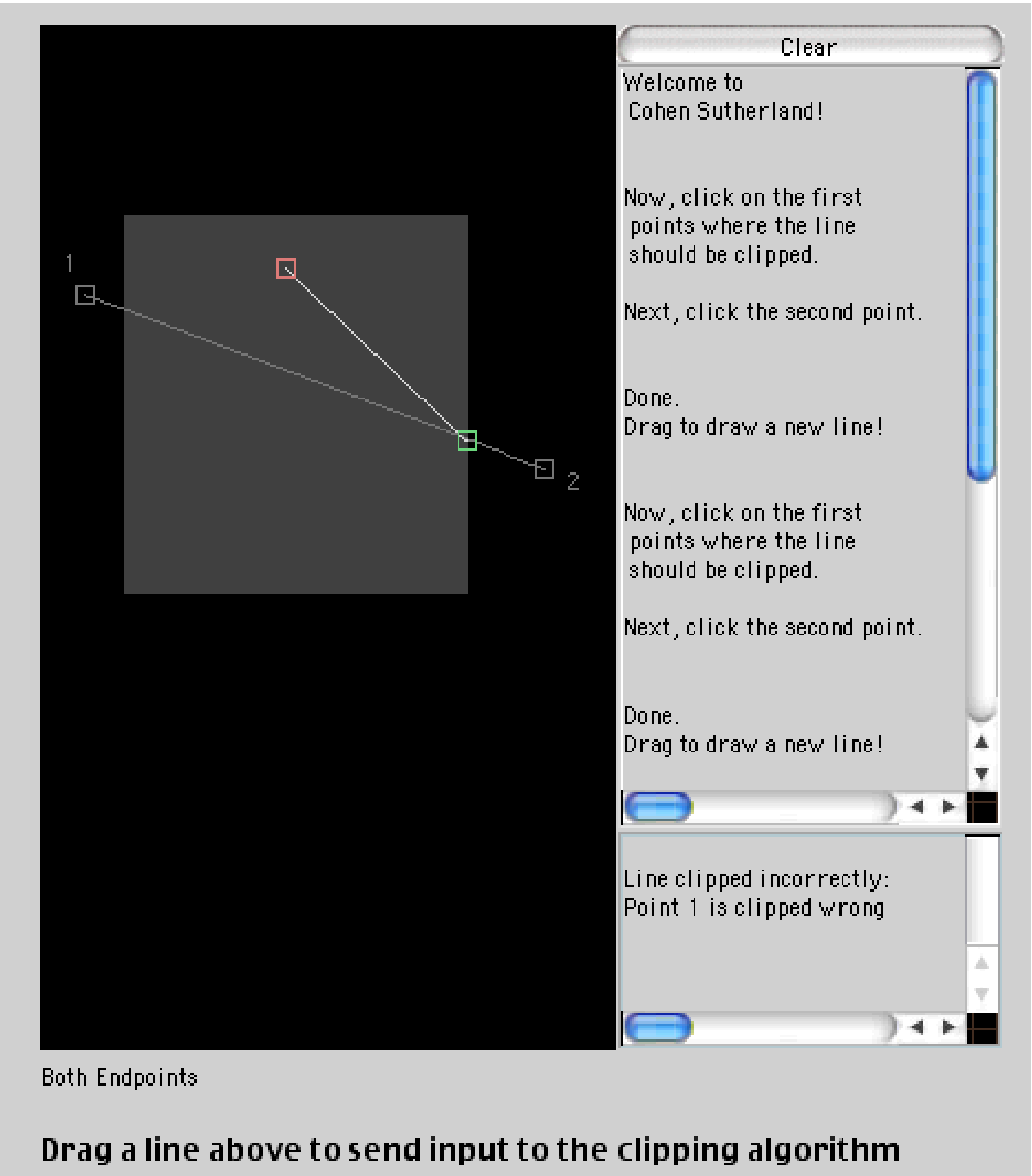


Figure 1-3: The visualization panel allows a student to explore the inner workings of an algorithm

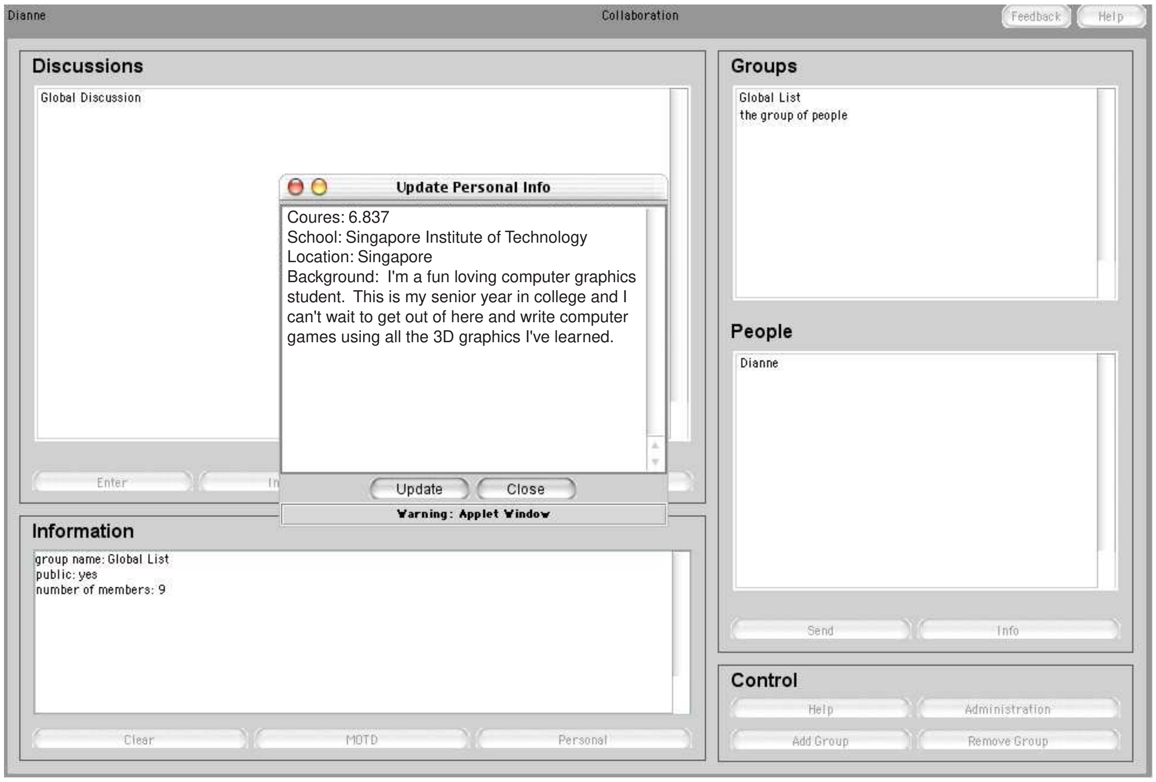


Figure 1-4: In chat mode, a student can collaborate with a variety of people

view a list of students working on her current problem and speak to one of them, ask for help from any of her friends in the class working on a different problem, or enter a help queue to receive help from a TA. Finally, when Dianne is satisfied with her implementation of an algorithm, she can turn it in to be graded by clicking the submit button on the side of her window.

1.1.2 Educational Fusion: Teaching Tool

The TA experience, although similar to the student's, is one of more power and omniscience. After loading a concept graph, the TA can see which modules all of his students are using. He can then click on a module and select a particular student's implementation to test it out. If something is not right, he can electronically look over the student's shoulder and examine the code the student is currently writing. If he wishes to comment, he can send the student a message pointing her in the right direction.

In addition, the TA can easily perform administrative tasks, such as checking the help queue for help requests, updating a message of the day which users see when logging on and grading work submitted by students.

1.1.3 Educational Fusion: Lesson Development Tool

For our purposes, a Fusion developer is anyone who wants to add a new module or set of modules to the Fusion environment. After logging in, the developer creates the desired new modules by selecting the create option from a popup menu. She then specifies the input and output types of the module. For instance, if creating a module to teach a sorting algorithm, she might create one input named `toSortVector` of type `java.util.Vector` and one output named `sortedVector` of the same type. Note, the names are completely arbitrary and can be anything she desires to give meaning to the data pathways. The data can also be of any type she chooses, as long as she specifies the proper package path, telling Fusion where to find info on the data type.

Fusion then creates some template files representing the new module. The developer must fill these in appropriately, and write the actual reference algorithm which the module is designed to teach. She then fills in another template with instructions, telling the student what to implement and how to send output data along the pathway. Finally, she fills in one more template for the witness detector, allowing the module to detect incorrectness in a student's implementation.

If desired, the developer can also create a visualization panel for the module, as mentioned above. Depending on the complexity of this panel, she needs to create Java subclasses of from one to three classes. These classes are used to create the visualization display, the user interface and the animated data type which shows the algorithm in action. These are explained in more detail below.

Finally, the developer can combine her modules on a concept graph and save it as a complete unit. It is then ready for students anywhere to load and implement and for TA's to check and grade.

1.2 Objectives and Motivation

The initial impetus for the Educational Fusion project came from Professor Seth Teller's vision of a new type of learning environment. As Brandon Porter, a fellow Fusion developer, points out in his thesis [Por98], Teller calls for an innovative, socially interactive experience in his NSF Career Development Plan:

> This proposal addresses the potential for collaborative interactive techniques to improve pedagogy at the undergraduate, graduate and professional levels, and performance evaluation at the undergraduate level. As increasing amounts of technical data and simulations come 'online' and university courses begin to follow suit, we must fulfill the educators' roles of selecting, organizing, and presenting this material to the student. [Tel94]

The time was ripe for a universal, multiuser framework to help teachers take advantage of the increase and availability of computing power- to help them move content and lesson plans out of the archaic realm of notebook and pencil and onto the vast network that already ran through educational institutions across the globe.

An important question remained, however, as to which features need be included in this new project and which should be left out in favor of a less complex or more platform independent implementation. Throughout the ongoing development, the objectives and goals of this endeavor have evolved as progress is made and we receive feedback from students who have used the system and professors interested in using the system. In addition, as the environment becomes more complete, we as developers can see farther into the future and envision innovative and exciting applications for Educational Fusion, causing us to revise our previous development plans. However, there have been a set of core objectives which remain central to Fusion's design and these are presented, along with their motivations, below.

1.2.1 Algorithm Context and Application

The student implementing an obscure algorithm should understand why the algorithm is important and how it affects the data flow of a more complex process. In traditional computer based assignments, a student may be asked to implement an obscure algorithm, such as balancing a red black tree, and then given a suite of tests with which to test his code. Although the student may drudgingly complete the assignment successively, he will not intrinsically be left with a sense of what he has accomplished. This is because the assignment lacks context.

Without context, it can be quite difficult to establish the importance of anything. Educational Fusion's concept graph helps show the relevance of an algorithm to its environment. By giving a graphical depiction of what input the algorithm requires, and what it must output, the student can see what significance the algorithm can have on a real world system. For instance, the lesson developer could design a graph of a simple tree based database, demonstrating the usefulness of the balancing algorithm. In addition, the developer could leave other key modules of the database open to student implementation, creating an entire lesson plan around the database model. By the time the student has completed the lesson, he will not only have learned the algorithms involved, but have gained an understanding of their real world applications, as well as a sense of accomplishment at completing a useful application. When a student can see and apply his work in a larger context, he is more likely to enjoy his work, and learn more thoroughly [Cho88].

1.2.2 Algorithm Visualization for Enhanced Learning

One of the more difficult tasks in coding up an advanced algorithm is understanding exactly how it works. Now matter how many detailed descriptions a students sits through or reads, she may still wind up at the dead end of incomprehension when she sits down to program her homework. We decided it would be ideal if the learning environment itself could somehow help the student understand the the specifics of the process which she is supposed to implement.

Our intuitions, as well as those of students and teachers alike, suggest that if a student could watch an algorithm work, then learning it would be all the easier. The process of algorithm animation inside Educational Fusion allows the student to see how the reference algorithm should manipulate data before the student attempts to implement the algorithm herself. This sort of animation raises learning efficiency especially well when presented in a homework style environment in which the student can reference the animation while solving problems [KST99]. Students also report that they feel like they have learned more after working with animated algorithms. These effects are particularly strong when the students are allowed to choose or specify the data sets on which the algorithm runs [LBS94], as is allowed in Educational Fusion [Tel+99].

To increase efficiency and reduce debugging time, we decided our environment should take an extra step in the direction of visualization and allow a student to examine *her own algorithm in action.* This way, when something is not working correctly, she can watch a visual representation of the entire process and gain insight on where her code may be going awry.

1.2.3 Witnesses Detection

Before submitting a homework assignment, most professors expect their students to check their work for errors. This is a reasonable and often beneficial expectation, as students often catch mistakes and learn from their correction. We decided Fusion should support this behavior and even point students toward it without the urging of a professor. For this purpose, we included the witness detection system.

When a student programs a simple algorithm, he can usually test its correctness by running it on some sample data sets and comparing the results to those expected. However, when the algorithm becomes somewhat complicated, such as ray tracing to a megapixel frame buffer or simulating the diffusion of Sodium ions through nerve cells, the output is often too massive to check carefully, causing the student to miss subtle errors, or possibly dissuading him from checking his work at all. If the student then turns in his work, the teacher cannot determine if the student would have been able to implement the algorithm correctly had he noticed the errors. The addition of witness detectors, procedures that check for small proofs of incorrectness, allows a student with a suitable test suite to make sure his algorithm is running as he and his professor expect. In the case where the student's implementation is incorrect, the witnesses allow the student to see how the implementation fails, aiding in the debugging process. For an example of witness detection in action, see Section 3.1, below.

1.2.4 Collaboration

Professors often want their students to collaborate on assignments. Ideally, the cooperating students will fill the gaps in each others' knowledge and understanding, reducing the amount of frustration associated with any given task. Students can also answer specific questions for each other in the absence of TAs, helping both the learning student and the teaching student to better comprehend the material.

We decided Educational Fusion should facilitate collaboration by establishing a virtual environment in which students can collaborate regardless of their physical location. Students should be able to view other students' working on similar problems and ask to work with them, allowing a new social framework to develop across the environment. In addition, students should be able to share their concept graphs, coding environments and visualization panels, allowing TAs and other students to look over their shoulders and help determine what is going wrong. In certain cases, a user should be able to highlight another user's code or twiddle virtual buttons on his interface, pointing out items of importance which may not have been noticed. Finally, continual conversation, first through typed text, and eventually spoken word and then video conferencing, should be supported, making collaboration virtually transparent over any distance.

1.2.5 Learn from home- in Somalia

As the Internet brings about a diaspora of knowledge throughout the ever shrinking world, there is an altruistic desire to bring the information of brick and mortar institutions into the living rooms of those not fortunate enough to be able to bring themselves to the information. Educational Fusion should aid immensely in this task, as its goal is to aid all teachers and students in their attempt to take advantage of the power of the digital age- not just the rich American ones.

Educational Fusion should remain transparent over distance, and entirely platform independent. This means that any module developed anywhere in the world should be usable anywhere else in the world with nothing but a Java enabled browser and a web connection. At the moment, any client can currently connect to Fusion from any platform, but by the end of the project, Fusion should also be servable and modifiable from any platform with no external software. At the moment, the server side is not completely self contained, but we are moving in this direction and should reach it soon.

By providing a framework around which professors can easily generate self checking, distance collaborative, animated problem sets, all that is necessary to teach those in even the hardest to reach places is an online text book and perhaps a streaming lecture. Once a professor has switched her traditional course to one with Fusion based problem sets, she can then share her knowledge and teaching ability with the entire world.

1.2.6 Comprehensiveness and Usability

Using Educational Fusion should not be a trial on patience, using the Internet or learning with on-line learning environments. Complexities of Educational Fusion should be hidden from the student, allowing him to focus on nothing but learning the lesson at hand. Also, getting the client environment up and running should be no harder than typing a URL into a browser. This way, the student will not become frustrated at the project before he has even begun work.

The server side of Fusion should also be easily installable and maintainable. At the moment, there is a short checklisted procedure (included in Appendix A) which a member of the teaching staff must follow to install Fusion. It is our goal to keep this procedure easy to understand and performable by a staff member of any subject who has typical computing experience.

Finally, the use of Educational Fusion should not require knowledge of an obscure and therefore useless programming language. Many times, software learning environments require that any lessons be designed to use the language in which the environment was written. Too often, this is a unconventional language like CLUE or CURL which the student is forced to learn and then unable to use outside her academic institution because of its limited appeal. We solved this problem for Educational Fusion by programming in Java. Although, presently, all modules must be programmed in Java, we have decided that Java is in wide enough use that it is reasonable to expect a computer science student to be familiar with it, or with a language very similar in structure. In addition, there is a concurrent project developing a SCHEME interpreter for Fusion to permit developers to create modules and lessons in that language as well [Bhu99].

1.2.7 Expandability towards nonalgorithmic oriented learning, including simulations

Within recent months, Fusion has outgrown its original limits. At conception, Fusion was designed to teach algorithms and only algorithms. This was a conscious decision based on the fact that Computer Science seemed particularly suitable to on-line learning, and there was no shortage of computer algorithms on which to test Fusion [Boy97]. So far, Fusion has been tested in real world situations as a teacher of algorithmic concepts and has performed promisingly [Por98].

This begs the question,"Can Educational Fusion do more?" It is our belief that it can. Related work in the field of on-line learning, addressed below, suggests that teachers are using on-line or computer based learning environments to support a myriad of subjects, from mathematics to history to biology. Although Fusion's visualization and concept graph based learning approach may not be suitable to more abstract topics such as history, we feel that Fusion's collaborative capabilities make it a worthy host for any subject's material that can easily be visualized.

In an attempt to encourage use of Educational Fusion outside the realm of Computer Science, we strive to facilitate expandability by new and unexpected modules. As a demonstration and test of this functionality, we have begun implementing Fusion versions of simulations and virtual labs from the disciplines of biology and electrical engineering. As Fusion development continues, we hope to maintain an adaptable infrastructure to encourage a vast array of interactive laboratory experiments such as these. This interactive lab concept is a topic of this thesis and will be discussed in detail below.

1.3 Background and Beyond

Thanks to the fabulous work of the previous Educational Fusion team, the foundations of the system were laid and tested long before this thesis was conceived. Collaboration, concept graphs, code submission and a variety of other features were already operational and have been used in real classes. Visualization panels were also operational, but clunky and more complicated than necessary. There was also an option to compare student implementations of algorithms to reference algorithms, hinting at the witness detection system to come.

This thesis focuses on three new aspects of the Fusion System: Witness Detection, Improved Algorithm Visualization, and Simulation and Interactive Lab implementation.

The next chapter gives some background on related work in the field of on-line education. This work played an important part in suggesting development paths for Fusion, particularly in the area of algorithm visualization.

Chapter three offers an overview of the witness detection system and explains its development.

Chapter four is the largest chapter of this thesis, describing the overhaul of the visualization panel and the concept of animated data structures. In addition, it gives several examples of suggested visualization panel layouts and explains the class hierarchy and module abstraction for panel development.

Next, Chapter five explains how this system is being used to implement interactive labs in biological and electrical engineering. Two sample labs are presented and their conversion to Fusion described.

Finally, Chapter six concludes with a review of the topics, some possible future work and advice to future Fusion developers. It also generalizes and extrapolates on the future of Fusion, as well as reminding us of the purpose of this educational endeavor.

Chapter 2

Related Work

Since the arrival of the Internet, it has been possible to widely distribute educational materials over long distances. E-mail, an active form of distribution, and the World Wide Web, a more passive form, have been used in classes at MIT for everything from homework distribution, to TA communication, to course preregistration. In recent years, Zephyr and AOL Instant Messenger have increased real-time connectivity,, allowing students to send messages to each other in real time, regardless of distance. This chapter gives a quick review of work in the field of computer aided education and compares and contrasts it to Educational Fusion.

2.1 Web based teaching

Most of the above technologies are noninteractive, and thus not very good platforms for complicated teaching environments. However, the web, with the addition of the Turing complete language, Java, has been the environment of choice for many Internet based teaching applications, including Educational Fusion. We will discuss some other applications here.

2.1.1 WebCT

The latest version of WebCT (2.1) may include everything a classroom could need, except a teacher [Gol+99]. The application, written in HTML, Javascript and some Java, fulfills the job of distributing all information related to a course. The student can pick which sections he wants on his home page and is automatically presented with them when logging in. The sections include syllabus, class notes, personal notes, grades, and a variety of other static pages authored by the course administrator.

In addition, the student can employ a suite of collaboration tools, such as chat, bulletin board and white board, which allow him to work with others. Finally, a student can log in to take an online quiz, consisting of multiple choice, matching and short answer questions which can be automatically graded by the system.

WebCT does a wonderful job of replicating a traditional class room on the Internet, allowing students to learn at a distance. It even supplies some amenities to the teacher, such as autograded quizzes and large databases of questions. However, it does not attempt to stretch the limits of the Internet by using networked computers to teach in some novel manner. For instance, a biomedical engineer would still require another piece of software to run a cell diffusion experiment. There seems no way to easily integrate third party code into the WebCT system.

Unfortunately, WebCT recently became a commercial product, no longer freely available to the impoverished professors and students of the world. However, according to WebCT statistics, "WebCT has more than 3.6 million student users in 97,000 courses at over 800 colleges and universities in more than 40 countries [Gol+99]," indicating that there is a definite niche for distance learning applications and suggests a bright future for Educational Fusion.

2.2 Visualization

Understanding through visualization has long been a concept which teachers have exploited to aid their instruction. Working through a math problem on the blackboard, performing a chemistry lab and watching a simulation of radioactive decay are all examples of learning through visualization. Several computer programs attempt to encapsulate visualization and make it easy for a professor to help her students see the inner workings of some subject. Most of these applications are limited in the scope of what they can help visualize; some of the more prominent ones are discussed below.

2.2.1 ZEUS

ZEUS is a system for algorithmic animation written in Modula-3 [Bro91]. It allows a professor to write an algorithm and include calls to procedures signifying interesting events. The programmer can then write modularized code defining a view which describes how the interesting events should be displayed and animated.

Zeus's greatest advantage over other algorithmic animations systems such as SAMBA [Sta96] is its layer of abstraction between algorithm and view. Once a view is defined to handle certain interesting events, the algorithm can be switched with another algorithm that uses the same events. For instance, a view that uses sticks of different lengths to show sorting techniques can support the visualization of a selection sort and a quick sort without rewriting view code. Likewise, the view could be switched to display balls of different radii instead of sticks and the algorithm code would not have to change.

ZEUS adds a layer of complexity, though, as it requires the algorithm coder to make explicit, often redundant calls to the interesting event procedures. That is, to swap two elements of an array, the programmer must call

$$\begin{aligned} temp &= a[i] \\ a[i] &= a[j] \\ a[j] &= temp \end{aligned}$$

and then call

$$EventSwap(a, i, j)$$

to tell the view to visualize the swap. While this may be just an annoyance to a professor writing code to be studied by students, it is a more significant problem if the students wish to see their own algorithms animated. The animation calls which they would have to insert might interfere with their understanding of the algorithm itself, causing the system to be more of a hindrance than an aid.

2.2.2 STPB and Weblab

Software for Teaching Physiology and Biophysics (STPB) and Weblab are two separate programs used to visualize experiments in MIT classes. STBP, written for Matlab, is used in the class 6.021, "Quantitative Physiology: Cells and Tissues" to simulate the diffusion of neurotransmitters and ions through cell membranes, to demonstrate the transmission of action potentials along nerve axons, and to elucidate the electrical properties of cells [Wei+92]. Students can pick a variety of numerical starting conditions and observe events by watching an animation of the diffusion or action potential and then interacting with a graph of the results. In the class, students are asked to predict trends in output based on input values and test their hypotheses with the software. According to the professor, the students gain a better understanding of the equations behind cell operations by watching them in action. Without a computer simulation, it would be too expensive and time consuming to perform this task as part of the class.

Weblab, used in the class 6.720J, "Integrated Microelectronic Devices" is similar to STPB. However, instead of being a simulation, it is a remote interface to a physical lab device [Ala99]. Whereas STPB is used because of the expense and impracticality involved in viewing cell diffusion, Virtual Lab is used because it's easier for fifty students to do their work without trekking into lab and crowding around one analyzer.

An interesting note about the two programs is that their front end interface is virtually identical. Students type in some numbers, slide some dials and get a graph back in response. However, the back ends are completely different. In STPB, a set of five differential equations are run to simulate diffusion. In Weblab, the student's input is fed into an actual piece of equipment, returning physical results which are then fed back over the Internet to the student. Recall that Educational Fusion has front and backs ends with tight abstraction barriers that can easily encapsulate both of these applications, making a consistent user experience for student electrical engineers and a consistent design experience for content developers.

2.2.3 CyberTutor

CyberTutor is another visualization tool at MIT, but includes a question and answer model for directing student study [Pri98]. Used in 8.01, "Physics I," CyberTutor allows a student to animate specific physical simulations by picking initial conditions and observing the results. For more control, a professor can write questions requiring a student to determine what initial conditions are needed to achieve a certain effect. These questions can be answered with numbers, analytic expressions or mouse drawn vectors and the student can immediately watch an animation displaying the effect of his answer. The system keeps track of correct answers and sends tallied results to the student and the professor when an exercise is complete. Its main advantage over the previous two systems is this cyberquiz ability.

2.3 Self Checking Applications

The witness detection system in Educational Fusion draws from the idea of self-checking applications - - programs that can check their own operation to test if they are generating the correct output. Educational Fusion does not test its own core system code, but it does use witness detection to check student implementation code, added into the system by the users.

Self-checking code is discussed by Manuel Blum and Sampath Kannan [Blu+95]. In their paper, they introduce the idea of a *program check*, an algorithm which checks the output of another algorithm. The program checker cannot verify the correctness of its code, but given a program and an input and output of that program, the checker determines whether the program ran correctly. Each class of program requires a specific program checker, tailored to test the algorithm realized by that program.

Blum and Kannan also discuss how program checkers can be used with carefully chosen inputs to establish a probability that a piece of code is working correctly. They give program checkers for sorting, matrix rank and greatest common denominator determination algorithms.

The witness detectors in Educational Fusion are very similar to these program checkers. However, they do not help produce sample inputs to give any probabilistic chance that an algorithm is correct. They do, however, allow feedback of any form, such as highlighting elements of output which helped the detector establish incorrectness. Blum and Kannan's checker returns only whether the program has performed correctly or not.

Chapter 3

Witness Detection

Witness detection is the ability of Educational Fusion to run an algorithm on a sample input and detect and display errors in the algorithm's output. It is not the ability to determine the correctness of a piece of code, which, incidentally, is a Turing incomplete problem. Therefore, it does not remove or diminish the function of a grader who must still check submitted code to make sure it is actually correct.

Although it is not immediately apparent how witness detection improves the life of the teacher, it is quite clear how it improves the life of the student. If a student consistently runs his algorithms on an ample test suite, he will never turn in incorrect code again. That is, if the student performs the proper array of glass box and black box testing on his algorithm, Fusion will help him pick out the errors in his output, allowing him to immediately hone in on the incorrect areas and correct them.

In this chapter, we discuss witness detection and give a short scenario of its use. We then give a history of witness detection in Educational Fusion and describe some advantages it now has over its earlier incarnations. Finally, we examine the technical details of a witness detector and explain how to implement one for a custom module.

3.1 Witness Detection in Action

Suppose Dianne, from chapter one, is implementing a bubble sort. She writes the code for her sort using the Fusion editor, then compiles it. She then switches to the

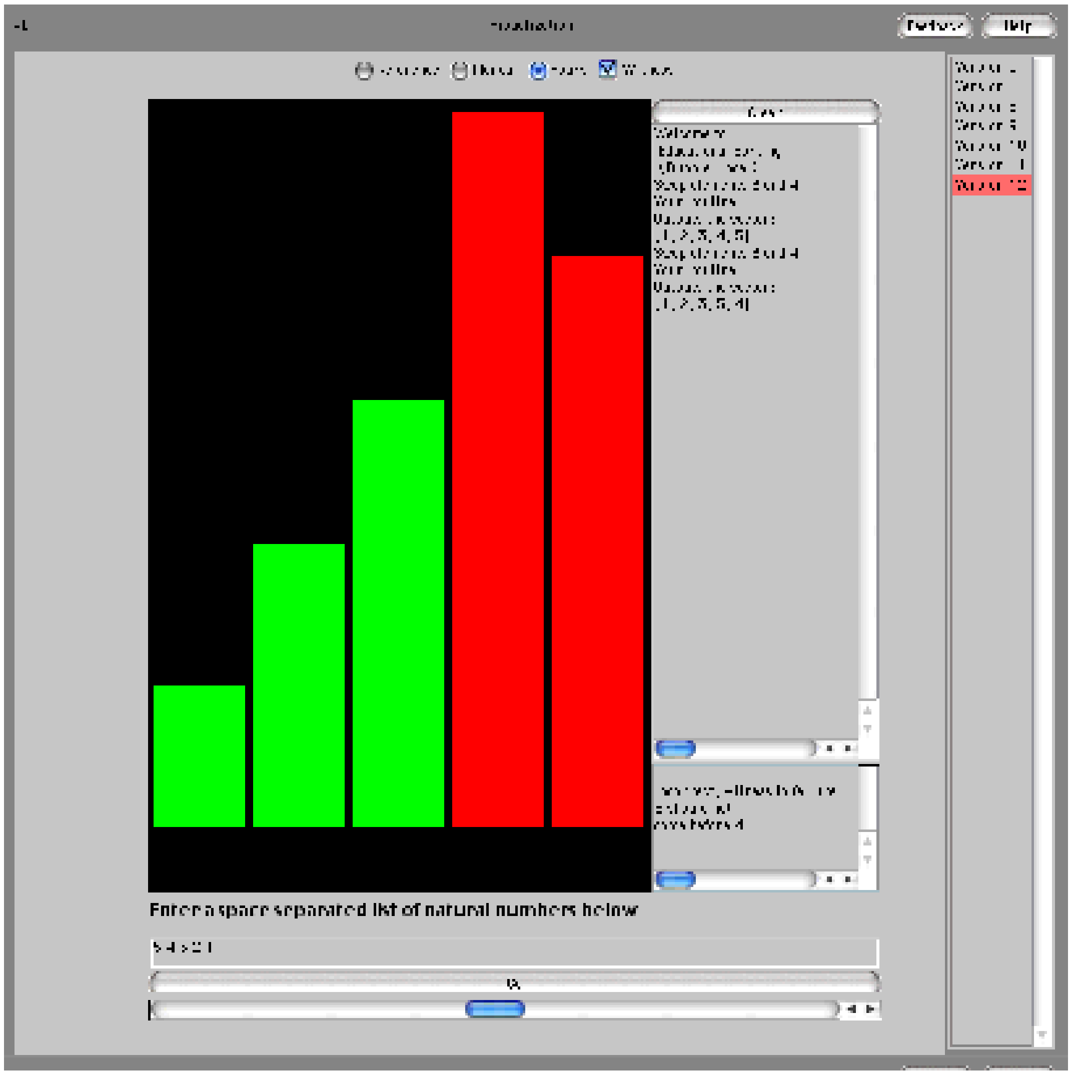


Figure 3-1: An incorrectly sorted vector

visualization panel to test her sort on the vector $\langle 5, 4, 3, 2, 1 \rangle$. She enters the data, turns on witness detection and clicks the go button. When the animation is complete, her screen looks like that of Figure 3-1.

This screen provides her with various kinds of feedback. The most noticeable is that the rightmost two bars are colored red instead of green like the other bars. This red warns her that there is something wrong with the output in this area. In her case, the last two elements of the list are out of order. For further feedback, if she does not immediately understand what is wrong, she can check the witness text box

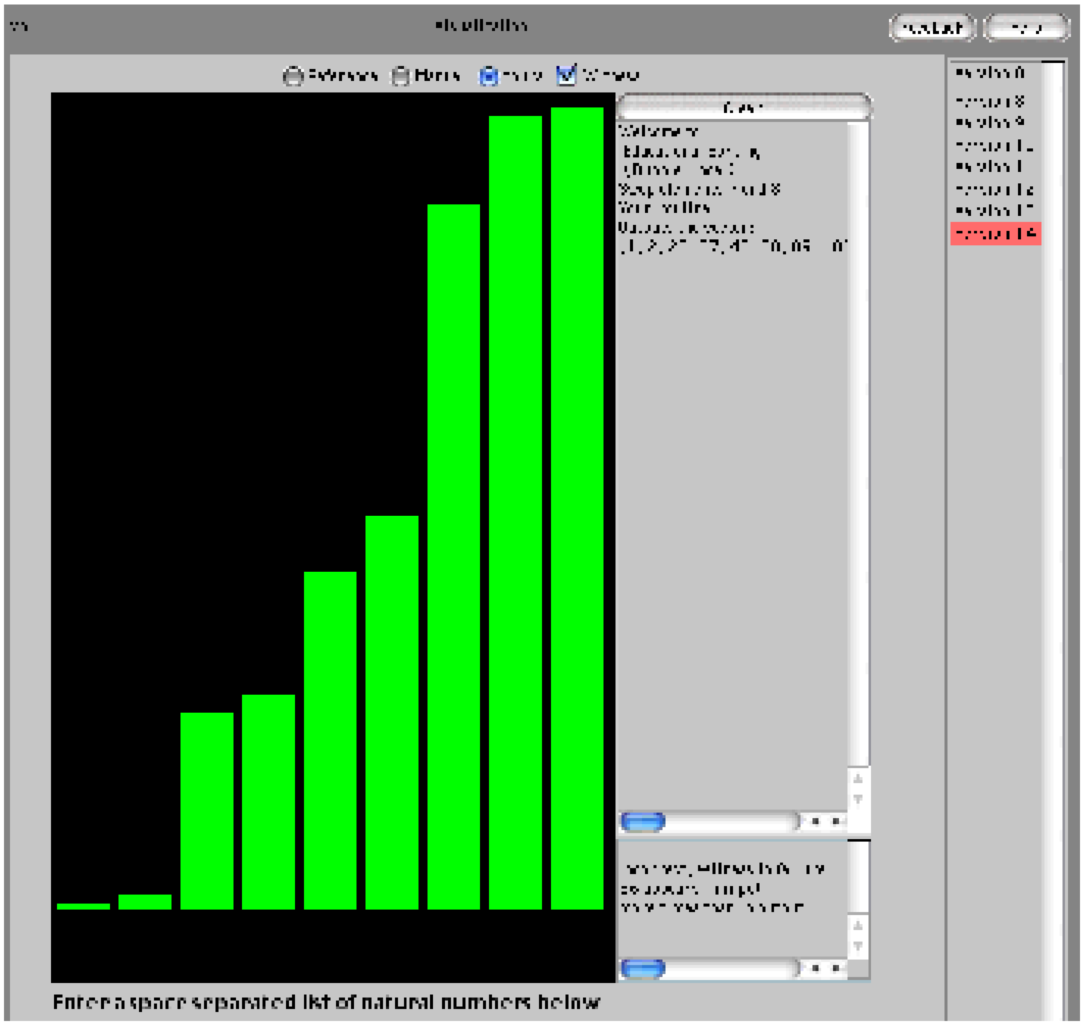


Figure 3-2: A correctly sorted vector with an element missing

in the lower right corner. This text box explains, “Incorrect. Witness to failure: 5 should not come before 4.”

Heeding the advice of the witness, Dianne returns to the editor to correct her code. She checks her loop invariant and finds she had an “off by one” error causing her to not check the last elements of the list. She then switches back to the visualization panel and tests her code on the vector $\langle 100, 101, 89, 43, 27, 25, 50, 2, 1, 56 \rangle$ only to find her implementation again incorrect, as seen in Figure 3-2.

After the animation completes, she breathes a sigh of relief because there are no red columns and no immediate visual clues to the incorrectness of her implementation.

She turns to the witness box to make sure the witness detection system agrees with her assessment of correctness but learns it does not. It gives her the message, "Incorrect. Witness to failure: 56 appears in input more times than in output." The witness detection system has alerted her that she is somehow losing the last element of the input vector. This is an even more useful function of the witness detector: In the first case, it was rather obvious to her that the sort was incorrect, just by the fact that the last two columns were out of order. In the second case, it was not obvious that there was an error in her code. In fact, had she been sorting a list of 1000 numbers, she would probably not have been able to notice the problem at all without witness detection.

Thanks to the detector, Dianne edits her code one last time, tries it out on some random data sets and receives the message: "Correct output for this input. Your code appears to be working correctly." This last message contains a subtle warning that her code is not necessarily correct just because it works on one, or even several, inputs. However, after a set of rigorous tests, it is good enough for Dianne and she submits her code to be graded by her TA.

3.2 Witness Detection History and Evolution

The witness detection system evolved from the "difference mode" option on the original visualization panel. In difference mode, a student could visually compare his implementation of an algorithm to the reference implementation by entering input and asking Fusion to run both implementations at once and return the results. Occasionally, a visualization panel would compare the two outputs, highlighting discrepancies for correction. For example, Figure 3-3 shows an instance of difference mode in the visualization panel of the Bresenham line drawing module. In the module, the student is asked to implement an algorithm which receives two endpoints as input and outputs all pixels which are covered by the line defined by the endpoints. The blue squares represent the reference implementation's output, the white square represents the correct pixel output by the student's implementation and the red squares

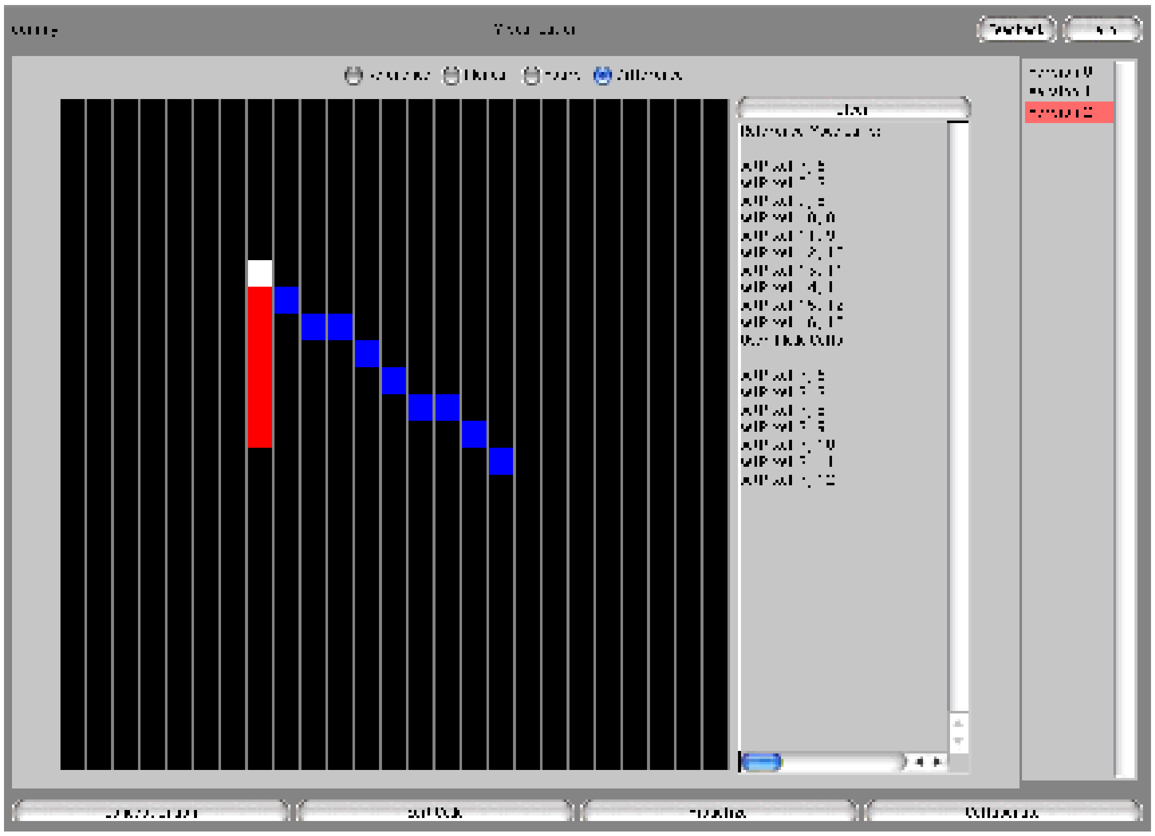


Figure 3-3: Bresenham Visualization in Difference Mode

represent the incorrect pixels.

Unfortunately, the method for supporting difference mode in a visualization panel was not well defined, and led to several different ways of displaying the results. This prevented any attempt at providing a uniform interface for Fusion, and made some the design and look of some panels a bit haphazard. When a panel did replicate other panels' implementations in an attempt to provide unity, much of the panel's code had to be copied as well, causing unsightly copies of identical procedures across multiple classes and modules.

3.3 Advantages to Witness Detection

The solution to the stylistic problems mentioned above was to abstract the "difference mode" concept out of the visualization and into its own subsection. The witness detector is now a distinct class which contains standard procedures to detect errors in an algorithm's output based on its input. Each module has its own witness detector, specialized to that module's algorithm. Any section of Fusion that uses witness detection has special hooks into the class which allow it to call the detector procedures. This witness detection evolution of the difference mode has several advantages over its predecessor which are described below.

3.3.1 Clarity and Elaboration of Results

Perhaps the greatest advantage of the new system is the standard ability to use any kind of data as a witness. Educational Fusion represents a witness as an object of type `Java.lang.Object`, so it can be any class possible in the Java language. This means witness detector results can be much more elaborate than the red dots drawn on the grid in Figure 3-3. Usually, the returned object is a string, explicitly stating whether the output is correct, and if not, exactly where the error lies. However, the object can be anything from a `java.awt.Point` describing the center of mass of a group of points to a heat map of a rocket engine after some algorithm is used to simulate fuel flow. Witness objects can even be based on time, allowing the witness detector to animate along with the algorithm on a visualization panel (see chapter 4).

In addition to the witness object returned, the new witness detector can also return a `java.util.Vector` containing a list of all invalid elements in the output. This is more useful than a single object if the developer wishes to somehow highlight aberrant objects for correction. With this function, it is easy to reproduce the behavior of the original difference mode of the Bresenham module discussed above. In this case, the error vector would contain pointers to all the pixels which are incorrect. The visualization panel could then interpret this data however desired, in this case, drawing the pixels in red. The single witness object, discussed in the previous paragraph, couls then be used to pass a string indicating whether the implementation has performed as expected.

3.3.2 Modularity and Abstraction

When converting the difference mode system to the witness detection system, there were three options as to where to put the witness detectors. They could be part of the visualization panel, part of the reference module or an entirely new class. The option of the visualization panel was the one currently employed by the difference mode and was unfavorable because it violated the purpose of the visualization panel, which was animating algorithms, not correcting code. It was an abstraction violation to have the visualization panel know how to check the correctness of an implementation, and thus not a proper decision.

Adding the witness detector to the reference module seemed a good choice because it did not violate any abstraction barrier. The reference module should know how to check for correctness, because it actually stores a correct version of the algorithm in the form of the reference implementation. After coding the reference implementation into a module, the developer would just add some procedures to handle witness detection, which could then be called by any subsystem that needed it.

The main problem with this alternative is that it does not allow for easy reuse of witness detectors. For instance, the witness detector used for the bubble sort described above could be used to detect witnesses for a variety of other sorts. Any algorithm which takes a list of elements as input and returns a sorted version as output could use the same "sorted list" detector to check for witnesses. If the witness detector were built into the reference module itself, the developer would have to copy and paste chunks of code for each module's detector. In addition, if he wants to modify the sorted list witness detector, he would have to go through each module and change the code everywhere it is used.

The solution here is to choose the final alternative for witness detector placement. Each witness detector should be in its own class, easily loadable by any module that needs its functions. This is the way witnesses are currently implemented in Educational Fusion, allowing for maximum modularity and code sharing.

3.3.3 Efficiency

The time required to perform an algorithm is not necessarily the same as the time required to check an algorithm's output for correctness. In many instances, the witness detection time can be much less than the time to actually run the algorithm. However, under difference mode, the only way to check an implementation's output for correctness was to run the reference implementation on the same input and compare, step by step. This was often a waste of time. For instance, in the bubble sort example, sorting requires $O(n^2)$ time, but having a witness detector check to make sure the list is sorted requires only $O(n)$ time. This may not seem significant in relation to sorting, but when the algorithm is performing a large ray trace or some other procedure for which n is on the magnitude of 10^6, this can amount to a large amount of time saved.

Note that it may occasionally take more time to check for witnesses than to generate a correct answer. This is most often the case when there is more than one correct answer allowed, as described in the next section.

3.3.4 Multiple Correct Results

The old, difference-mode style of feedback supported only algorithms with one correct output per input. This made no allowance for algorithms which mapped an input to more than one correct output and then returned one of the correct choices. For example, in the Bresenham line drawing algorithm, the implementation is supposed to output each pixel through which the input line passes. However, when the line passes directly between two pixels, it is acceptable to output either of the pixels, but not both. If the student and reference implementations pick different pixels, the difference mode will highlight the student's pixel in red, indicating an error where there is none and possibly driving the student into a rage of misunderstanding. The witness detector, however, can go through each pixel of the student's output, individually determining if that pixels is correct relative to the input and the other pixels in the output. In this way, any output can be tested based on a specific definition of correctness, rather than on similarity to the output of the reference algorithm.

3.3.5 Sanity Check

Since each witness detector is self contained, it does not rely on the reference algorithm to check for correctness. This means that it can even be applied directly to the reference algorithm. This is useful for the developer to test that she herself has programmed the reference implementation correctly. It is also useful for students who sometimes end up with buggy reference code: If they suspect the reference algorithm is not performing correctly, they can double check it by running a witness detector on its output. Of course, this is not foolproof because the detector itself may be buggy, but, in the case of a competent developer, the odds of both systems failing are rather low.

3.4 Witness Detector Implementation and Technical Details

To implement a witness detector, the developer creates a subclass of `edufuse.cg.BaseWitProof` and names it `Proof`*Modulename* where *modulename* is the name of his base module. The new class should reside in the same package as the module.

Next, he overrides the function `public Object BaseWitProof.FindWitness(Object input, Object output)`. This function receives the object which was input into the implementation, and the object which the implementation produced. It should test the output object for correctness and return some meaningful object indicating whether the output is correct and if not, what is wrong. Usually this output is a string, but is allowed to be any object for expandability. It is up to the function calling the witness detector to interpret this object.

Finally, the developer should override the function `public Vector BaseWitProof.`

`FindWitnessList(Object input, Object output)`. This function receives the same parameters as the previous function, but now outputs a `java.util.Vector` which represents a list of elements that are incorrect, or do not belong, in the output. This vector can contain pointers to the actual erroneous elements or, if the input itself is also a list as in the case of a sort algorithm, the vector might just contain integers representing the indices of incorrect elements. Note that for some algorithms, such as those that return only one number, the concept of a list of incorrect elements makes no sense and `FindWitnessList()` should just return an empty vector. Once again, it is the responsibility of the calling procedure to interpret the vector returned by this function.

The witness detector should also be used to hold any auxiliary functions which `FindWitness()` and `FindWitnessList()` invoke. The `BaseWitProof` contains some other functions which are useful for creating witnesses to simulations and are explained in the chapter on simulations, below. For reference, the complete code of the witness detector for bubble sort, and thus any sort, is listed in Appendix C.

3.5 Witnesses- Are We Making Life Too Easy?

When offering students the chance to automatically check their own work, there's always the chance that we might make their lives too easy. Is witness detection really such a boon to teaching, or will it just dull the abilities of students to notice their own mistakes? We believe that witness detection truly is beneficial and will not harm the learning process.

It is true that students need scrutinize their results less with witness detection. However, if the goal is to teach students to scrutinize results, then that should be a separate class, or a module of a class on the scientific method. Educational Fusion is designed for classes in which the purpose is to teach the material itself, not how to check yourself. In addition, when students apply their knowledge of these algorithms in the real world, there will be a host of debugging and verification tools available which they can use, or may even be forced to use by circumstances. Witness detection is just an application of verification technology at the academic level and students should be given an opportunity to adapt to it.

Without witness detection, there are some modules which would be very difficult to check by hand. For instance, when implementing a Phong shader, it is difficult to tell if the shader is operating correctly around the edges of the polygons it is shading. A student could stare at individual pixels for hours trying to determine if the shader oversteps the bounds by one pixel, or he could ask the witness detector and receive results immediately. This way, a student can remain focused on the task of implementing the shader itself and not become frustrated squinting at the output.

Finally, it is important to remember that the witness detector does not directly examine the code for correctness. It still relies on the student to run his algorithm on an adequate test suite, representing all possible classes of input. The witness detector is also limited by the fact that it cannot tell if an implementation is obtaining results in the correct manner. For instance, if a student is supposed to implement radix sort, but implements bubble sort instead, the witness detector will report no errors as long as the implementation outputs a sorted list[1]. Thus, a large part of the verification process still rests on the student and witness detection does not make his life too easy.

[1]This is true unless the witness detector is built to monitor execution time, binary operations, and such. Even then, it would still be rather difficult to discriminate between bubble sort and selection sort, or radix sort and counting sort.

Chapter 4

Algorithm Visualization

Algorithm visualization refers to the ability of Educational Fusion to display, through animations, the internal workings of an algorithm. Students access algorithm visualization by selecting a module and clicking the visualization panel button on the bottom of the screen.

In Educational Fusion, algorithm visualization is extraordinarily modular and parts are freely interchangeable. A developer can switch a module's visualization panel just by typing a new panel's name in the module configuration file. A panel can be linked to more than one module if the algorithms are of similar types. For instance, several sorting algorithms could all use the same visualization panel to display their animations. Or, if the representation of data must be switched, such as displaying a list of integers by tall bars of the appropriate length instead of balls in bins, it can be accomplished by changing just one or two functions. The system is designed to be flexible and expandable.

This chapter describes, in detail, the visualization abilities of Fusion. It first presents a student's view of the system and then explains how visualization has evolved over Fusion's development. Next, it moves on to explain the framework with which Fusion supports the development of new panels. This section includes instruction on the `DefaultTeacher` and the `VizView` and finally provides an in-depth discussion on the concept of Fusion's animated data structures.

4.1 A Student's View of Visualization

Upon opening the visualization panel, Dianne can select from one of three modes: reference, manual and *yours*. In reference mode, she can enter sample input values and watch the reference implementation work. After observing how the algorithm is supposed to operate, she can switch to manual mode, in which she incrementally completes the steps of the algorithm to produce the correct output. For instance, if she is implementing Bresenham's line drawing algorithm, she would first input the endpoints of a line to draw and then click on the individual pixels which should be highlighted by the procedure. Finally, once she thoroughly understands the input-output mapping of the algorithm, she can write her own implementation and test it out in *yours* mode.

At anytime, Dianne can toggle the check box which controls witness detection. If she turns it on during reference mode, it performs the sanity check mentioned above. If she turns it on in manual mode, each action she takes is accompanied by an indication of its correctness. In the Bresenham module, her pixels immediately light up in blue if they are correct or red if they are incorrect. Finally, if she employs witness detection in *yours* mode, the visualization panel relates any witnesses of incorrectness that it can find in her implementation's output. For line drawing, it gives Dianne a message stating whether the correct pixels, and only the correct pixels, are highlighted by her algorithm. In all three cases, the system calls the same two procedures in the witness detector to produce the appropriate results.

4.2 History of Visualization in Educational Fusion

Algorithm animation and visualization has been an integral part of Educational Fusion since conception. The original team believed that animation allowed more thorough exploration of algorithms than a traditional setting and wanted to reach new levels of visualization with the project. The most important advancement over related work was the ability of students to visualize their own algorithms. A student could select an option on the visualization panel to test data on his own implementation and watch it work in real time. Outside of the visualization panel itself, a student could also run his implementation from the concept graph and watch as each module in the data flow displayed its output, allowing him to track down bugs by examining data at several checkpoints.

In the beginning, the team was eager to create a proof of concept prototype for the visualization panel and hacked together some effective code. Later, as each new visualization panel was created, the team tried to push the limits of the panel and improve the interface and structure of the code. Although this resulted in many working panels, the panels suffered from a lack of uniformity. In addition, large sections of code had to be rewritten and modified slightly each time a new panel was made because there was no framework around which development could be based.

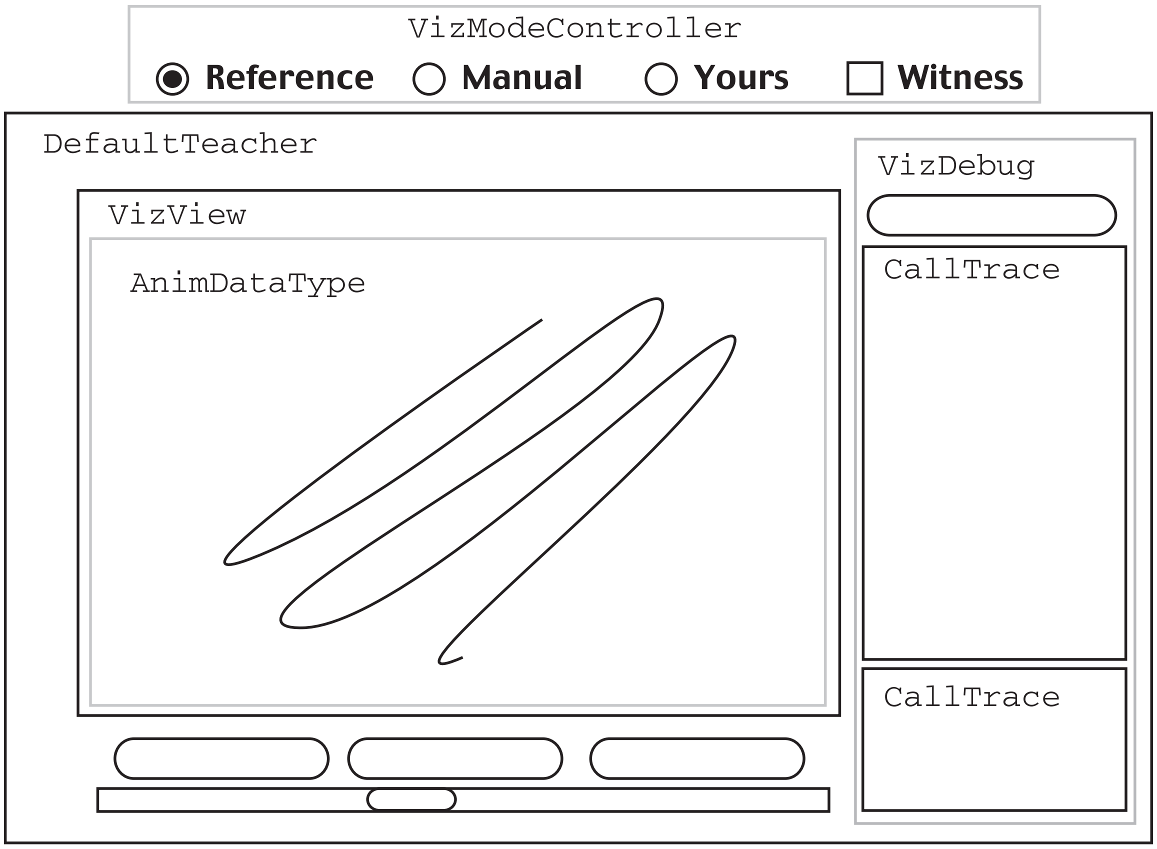
4.3 Visualization Framework

Now, however, Educational Fusion hosts a rich framework, capable of supporting any visualization panel desired. The large chest of tools makes it easy for a developer to build a panel in just a matter of hours, and the standardization provided make it easy for a user to understand the panel in a matter of minutes.

Each section of the visualization panel is supported by a specific hierarchy of classes, as shown in Figure 4-1. The dependencies between these classes is illustrated in Figure 4-2.

4.4 DefaultTeacher

The class `edufuse.cg.DefaultTeacher` is the main control board for events and animations on the visualization panel. As the middle link between the user interface and the back end of fusion, it serves a dual purpose. First, it is responsible for linking to external code, such as the student and reference modules and the witness detection system, ensuring that data can be passed back and forth between the user,



Light gray boxes represent functional groupings.

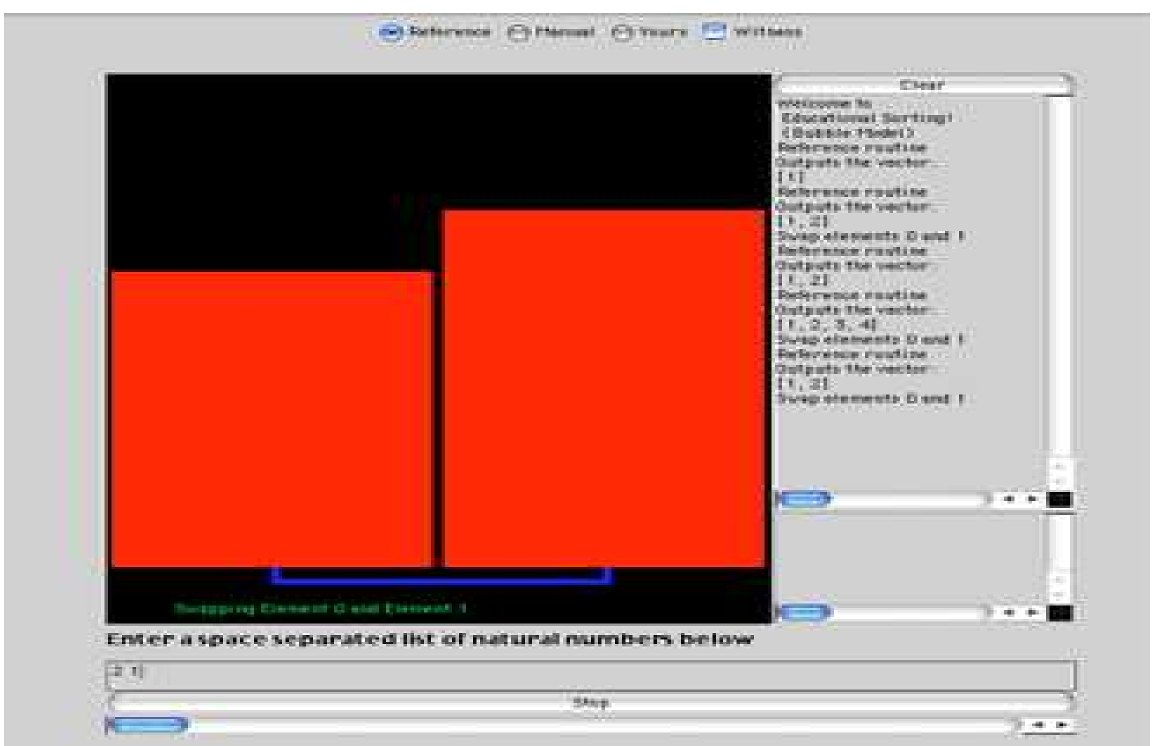


Figure 4-1: Visualization Panel Layout

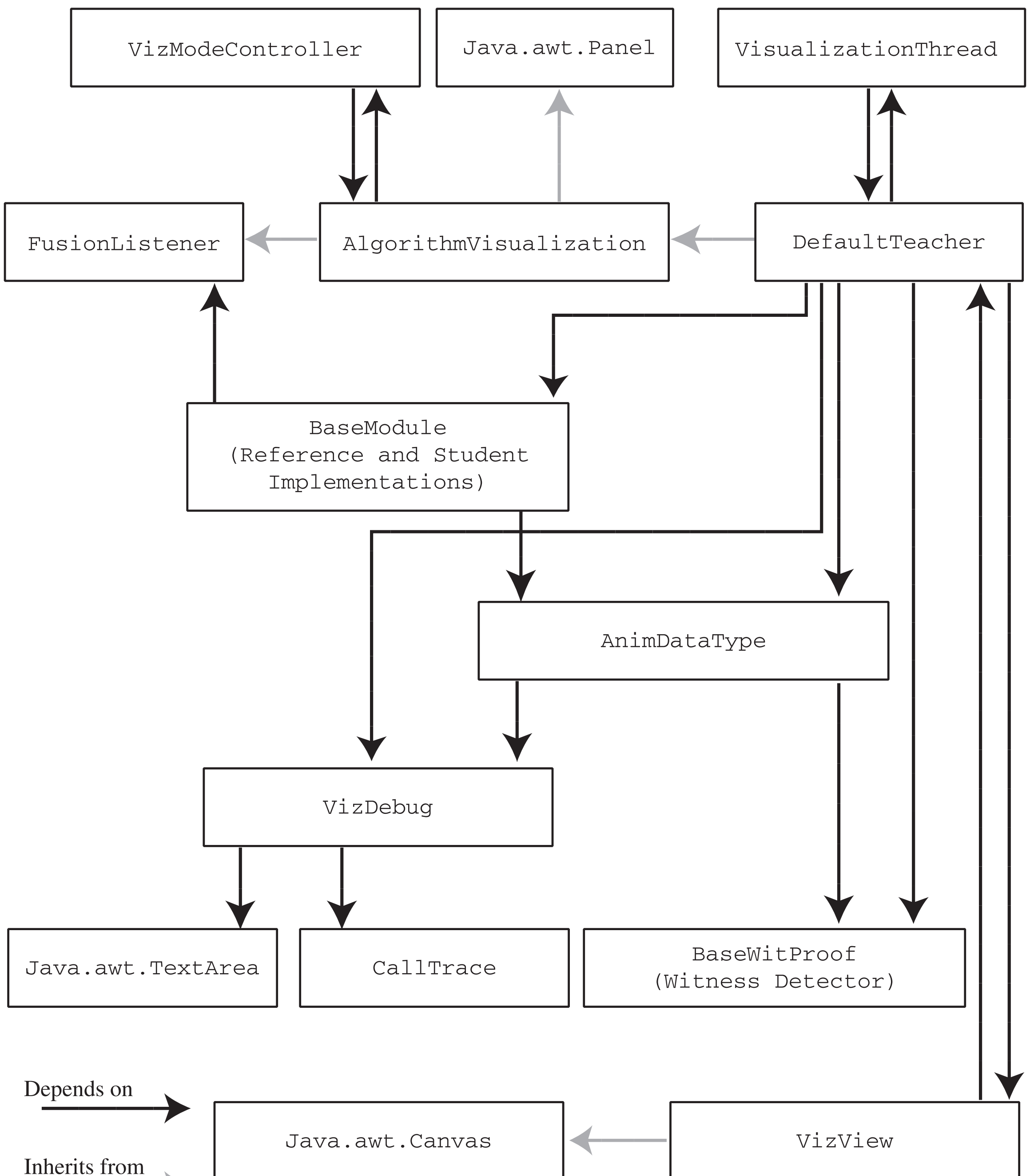


Figure 4-2: Module Dependency Diagram for Visualization Panel.

the visualization, the implementations to be tested and the panel's witness detector. Next, `DefaultTeacher` fills the secondary role of holding and organizing all of the visualization's user interface components.

To create a custom panel for a new module, the developer must create a subclass of `DefaultTeacher` named *mod*`Panel` where *mod* is the name of the new module. Then, the developer overrides the appropriate functions as described in the next several sections.

4.4.1 Interaction With External Code

`DefaultTeacher` contains a large number of methods and fields to aid in managing external code. The steps used to connect and send messages to external modules are displayed in Figure 4-3 and explained below.

Loading the Modules

Before `DefaultTeacher` can interact with external modules, it must somehow connect to the appropriate classes. It does this by creating and storing instances of the desired classes with the `notifyName()` procedure. `Public void notifyName(String name)` takes the full path name of the class which contains the user's implementation of the algorithm to be visualized. This name reflects the version of the implementation the user has selected, so that the appropriate revision can be loaded. For example, `notifyName()` might receive the string:

projects.graphics.users.vbunny.modules.VectorSort.vbunny_VectorSort_14

This string, which is just a standard full pathname in Java, tells the function to load the 14th version of `vbunny_VectorSort` inside the package containing modules for the user vbunny. Having established this information, `notifyName()` passes the string to `setUserModuleBean()` to actually instantiate the class and save it in a member field.

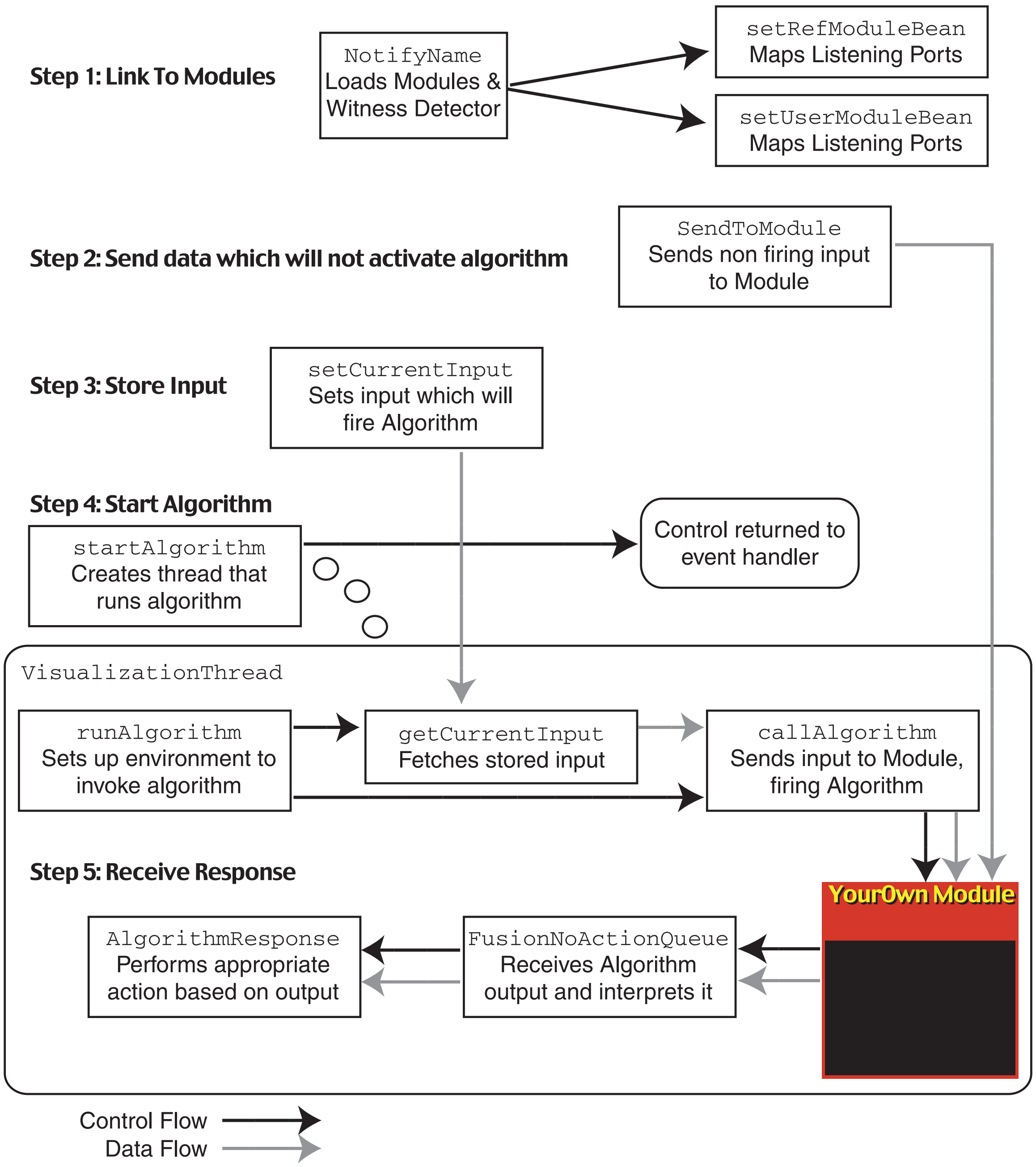


Figure 4-3: Control and Data flow to run an algorithm within a visualization panel.

Next, `notifyName()` strips away user-related data in the input and uses this information to form the path names of the appropriate reference class and witness detector. In this case, these would be `projects.graphics.modules.VectorSort.ref_VectorSort` and `projects.graphics.modules.VectorSort.proofVectorSort`. It then instantiates these classes and stores them by calling `setRefModuleBean()` and `setProofModuleBean()`, respectively.

Linking to the Modules

Once the modules are loaded, Fusion must build data pathways from the visualization panel to the student and reference modules. These pathways are identical to those used in the concept graph, as explained in Nathan Boyd's thesis on Educational Fusion [Boy97]. In short, the class `edufuse.cg.BaseModule`, from which the student and reference modules are derived, and the `DefaultTeacher` class both implement the interface `edufuse.cg.FusionListener`. This interface allows modules and panels to exchange data by connecting an input port of the visualization panel to the output port of a module. For our purpose, this is accomplished through the use of the function, `public boolean BaseModule.mapListener(FusionListener listener, int localIndex, int remoteIndex)`. Calling this function tells the module that the listener would like to receive notice on its input port numbered `remoteIndex` whenever the module sends data from its output port numbered `localIndex`. The function returns a boolean indicating whether the mapping completed successfully or not.

Conventionally, even-valued input ports represent data received from the reference implementation and odd-valued input ports represent data received from the student implementation. Also by convention, the 0th input port receives the main output of the reference implementation and the 1st input port receives the main output of the student implementation. Any auxiliary data should be sent on other ports, preserving the even/odd convention. The *main* output of an algorithm is the output which an observer would most expect the algorithm to output. For example, a reference implementation of a line clipping algorithm would send the clipped line down its 0th

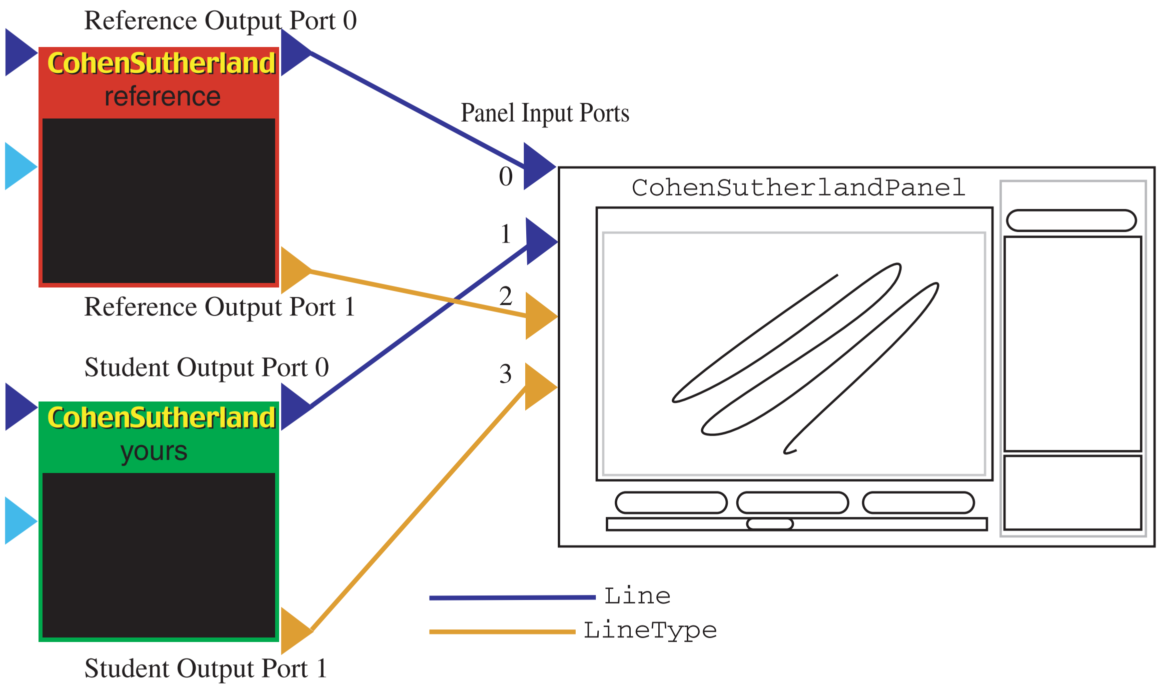


Figure 4-4: A pseudo-concept graph, displaying links from modules to visualization panels

output port and the type of the clipped line (external, internal, etc.) down its 1st output port. It would then map from output port 0 to input port 0 and from output port 1 to input port 2. Similarly, the student implementation should map from output port 0 to input port 1 and from output port 1 to input port 3 (Figure 4-4).

Usually, the `setXXXModuleBean` functions handle all necessary listener mapping. `Public void DefaultTeacher.setRefModuleBean(String ClassName)` first instantiates the reference module and saves it in the member field, `mRefModuleBean`. It then maps the 0th output port of the module to the 0th input port of the panel and the 1st output port of the module to the 2nd input port of the panel, as described above.

`Public void DefaultTeacher.setUserModuleBean(String ClassName)` acts as `setRefModuleBean()` except that it stores the user module in `mUserModuleBean` and maps from ports 0 and 1 to ports 1 and 3, respectively.

If more than two pathways are needed per module, your `modPanel` should override the `setXXXModuleBean` functions and call `mapListener()` appropriately. Note that it may also be necessary to modify `DefaultTeacher.FusionNoActionQueue()` if you wish to name these pathways, as described below.

Sending Data to a Module

It is interesting to note in Figure 4-4 that there are no data pathways drawn from the visualization panel to the modules. This is because data is passed by calling module functions directly rather than having the module listen to the visualization ports.

By having modules listen to the visualization panel through the listener interface, we made it very easy to change the entire visualization panel framework without affecting or having to adapt the module framework. However, we do not expect to modify the module framework and thus are not very concerned about what affect this might have on the visualization panel framework. Thus, we allow the visualization panel to depend on a static module design and to call functions directly. The benefit of this allowance is that the system remains less complex and visualization panels are easier to implement and maintain.

To send data from a visualization panel to a module, use the function `protected void DefaultTeacher.SendToModule(Object data, int inputNum)`. This sends data to input port number `inputNum` of the module. Note that there is no need to specify the module to which to send the data. Instead, `SendToModule()` checks the mode of the visualization panel (i.e. reference or yours) and sends the data to the appropriate module.

`SendToModule()` should be used to send only auxiliary data to the modules. It should not be used to send the main input parameter which causes the algorithm to execute and return data. For instance, for a line clipping module, use `SendToModule()` to send the boundaries of the rectangle to which the line should be clipped. To send the actual line, and therefore trigger the clipping mechanism and receive a response, use `startAlgorithm()`, described below.

Storing the Input

Before running an implementation, your visualization panel must store the input which will fire the algorithm by calling `public void DefaultTeacher.setCurrentInput(Object in)`. If the input is going to be changed by the algorithm and you want the data in the variable `in` to remain unchanged, you should clone it when passing it as a parameter.

The input is stored in this manner for two reasons. First, it is useful for the visualization panel to always be able to access a copy of the input which fired the algorithm. That way, if the panel checks directly for witnesses, it can access the input by calling `public Object DefaultTeacher.getCurrentInput()` and feeding it into the witness detector. Second, it is necessary to store the input so that the algorithm itself can access it. To facilitate multi-tasking and customizable user interfaces, the algorithm is run in a manner that does not allow passing in a direct parameter. Thus, the input must be stored beforehand in a place accessible by other visualization panel methods.

Running the Algorithm

Once the triggering input has been stored, the algorithm can be running by calling `public void DefaultTeacher.startAlgorithm(void)`. At first, this may seem a curious way to run an algorithm, as it takes no parameters and returns no response. However, the input was previously stored, as mentioned above, and the responses come through listening input ports of the panel, as described in section 4.4.1. `startAlgorithm()` is designed this way so that it can be called by a user interface element, such as a button or menu. That way, a developer can build a custom interface for the visualization panel and have it trigger the algorithm in any manner desired.

Once called, `startAlgorithm` creates a new thread to run the algorithm. Through this, the user retains control of Educational Fusion even while the algorithm is running. A user can stop a long algorithm in the middle, or possibly even change the speed of the animation, all while the algorithm is executing. Also, incorrect student implementations which execute infinite loops can be stopped by clicking a stop button, rather than necessitating a restart of the Fusion environment.

The thread created by `startAlgorithm()` is an `edufuse.modbase.VisualizationThread` which receives a pointer to its parent `DefaultTeacher` during construction. It then executes its `DefaultTeacher`'s `runAlgorithm()` method. This method usually just invokes the algorithm by calling `public void DefaultTeacher.callAlgorithm(Object input)` on the currently stored input. However, you should override this function if you wish your algorithm to be called in a specific way. For example, if your algorithm creates a permutation of its input, like an algorithm that plays Conway's Game of Life [Ber+82], you may want to call the algorithm repeatedly, each time providing the previous output. To do this, put the call to `callAlgorithm(getCurrentInput())` within an infinite `while` loop. Then, make sure to set a new current input each time the visualization panel receives a response from the algorithm. This process would only end when the user clicks the interface's stop button, killing the thread[1].

You should also use `runAlgorithm()` as an opportunity to obtain any input from the user interface which has not yet been sent to the module. For instance, in a visualization panel for a sorting algorithm, you might have a text box for entry of the list to sort. You would not want to poll the content of this box until the user is actually ready to run the algorithm, so you should use `runAlgorithm()` to check the box and call `setCurrentInput()` with its contents before calling `callAlgorithm()`.

The method `callAlgorithm()` is actually just a wrapper for the `SendToModule()` method. It exists to make it clear that `runAlgorithm()` is actually triggering the algorithm by sending the critical piece of data. `callAlgorithm()` usually sends its input parameter straight to the appropriate module by calling `SendToModule(input, 0)`. However, although there is a convention suggesting how a module should use its

[1]There are more elegant ways to have the algorithm terminate. For instance, the visualization panel could just monitor a boolean flag which the stop button trips when the algorithm should be stopped. However, this method is not totally impervious to bad student code and the only way to definitely prevent runaway student algorithms is to kill the thread completely.

output ports, there is no standard requiring input port 0 of a module to be the port which causes the algorithm to fire. In fact, a module can use any port it wants as the firing port [Boy99]. If your module uses a port other than 0, you must override `callAlgorithm()` to send the data down the appropriate pathway.

Note that there are many functions mentioned above which result in the eventual triggering of the algorithm. However, to preserve abstraction and ensure smooth running of visualization panels, `startAlgorithm()` is the only one that should ever be called externally. To help prevent misuse, it is the only public function while the rest are private. Although there are ways to work around this barrier, they should be scrupulously avoided.

Receiving the Algorithm's Output

When a visualization panel receives output from a module, it immediately calls the method `public void DefaultTeacher.fusionActionNoQueue(Object object, int index, boolean local)`, where `object` is the data received, `index` is the port on which it arrived and `local` is whether the data should be passed on to another module. For visualization purposes, `local` is always false.

`fusionActionNoQueue()` then examines the port to determine the source and name of the output it received. If the port is even, it assumes the data arrived from a reference implementation and if the port is odd, it assumes a student implementation. This is in accordance with the numbering convention mentioned above. To determine the name of the received object, `fusionActionNoQueue()` has a small, built in lookup table based on port number. If the port number is zero or one, it names the data "main." If the port number is two or three, it names the data "param," because the auxiliary port is usually used to return the parameters a module generated to create its output (see Chapter 5 on Simulations, below). Any other response is simply named after the port on which it arrived. That is, data arriving on port six would be named "6." If you wish to use different names for the data, you should override this function and edit the if-then clauses which pick the name. For instance, in the line clipping example given above, you could easily set the name of the data to "Line Type" if it came in on port two or three [2].

After establishing the relevant information, `fusionActionNoQueue()` calls `protected void DefaultTeacher.AlgorithmResponse(String name, Object object, int mode)`. In this call, `name` is the name of the data, `object` is the piece of data itself and `mode` is either `VizModeController.REF_MODE` or `VizModeController.USER_MODE`, to indicate reference mode or student mode, respectively. `AlgorithmResponse` is a higher level version of `fusionActionNoQueue()`: It receives its data in a more manageable form and it has a more informative name. Therefore, this is the function that should handle a response from the algorithm. Your panel must override this method if you want it to do anything meaningful upon receiving a response from an algorithm. If witness detection is on, this method should fetch the input which triggered the algorithm, send it to the witness detector along with the output and display the results in the witness area, as described below. See line 78 of `VecSortPanel.java` for an in depth example [Gla00].

Looking for Witnesses

If the user has activated witness detection, he will probably expect some kind of feedback from the witness detector whenever an algorithm is run. To obtain this feedback, `DefaultTeacher` has a host of witness utilities. The first of these is `public boolean AlgorithmVisualization.getIsWitnessing()`. This allows `AlgorithmResponse()` to determine if witness detection is on or not. If witness detection is not on, then nothing need be sent to the witness detector and no feedback should be shown to the user. Conversely, `DefaultTeacher` also provides `public void AlgorithmVisualization.setIsWitnessing(boolean isW)` to force witness detection on if necessary.

Next, `DefaultTeacher` provides `public Object DefaultTeacher.FindWitnes-(Object o)` and `public Object DefaultTeacher.FindWitness(Object i,`

[2]It would be interesting if the concept graph architecture were modified to include a typing mechanism for data passed between modules. Every time a module produced output, it would label the output with its name. That way, `fusionActionNoQueue()` would not be responsible for naming the data, tightening the abstraction barrier even more. Unfortunately, that is beyond the scope of this thesis, but a suggested implementation is given in chapter 6.

`Object o)` to obtain witness objects from the detector. The parameter `o` should be the output returned from a module and the parameter `i` should be the input which caused that output. The two functions are almost the same except that `FindWitness(Object o)` assumes the triggering input is the one obtained by calling `getCurrentInput()`[3].

Once an object is returned by a witness detector, it is up to the visualization panel to display it. This is usually accomplished by printing the string in a small text box designated for witnesses. This is explained below in Section 4.4.2.

Finally, `DefaultTeacher` also provides routines to access the witness list, as described in chapter 3. To do this, use the function `public Vector DefaultTeacher. FindWitnessList(Object o)` or `public Vector DefaultTeacher. FindWitness-List(Object i, Object o)`. As for the methods mentioned earlier, the former function is implicitly passed the input with `setCurrentInput()` and the latter function is explicitly passed the input in the parameter `i`.

4.4.2 User Interface Framework

The algorithm activation framework just described would be rather useless if it did not allow a user to activate it with desired inputs. Therefore, the `DefaultTeacher` class also contains a toolkit for establishing and maintaining a standard visualization user interface. With it, a developer can create a panel to control the operation of the algorithm and its animation, attach an input bank to allow the user to finely tune the input to the algorithm, and send feedback to the user through a variety of specialized channels.

Initializing the User Interface

To build its user interface, `DefaultTeacher`'s constructor calls the function `public void DefaultTeacher.standardSetup()`. `standardSetup()` is in charge of chang-

[3]Currently, this assumption should always be true, but the earlier method is available in case some future, unanticipated visualization panel needs to call the witness detector with output generated outside the algorithm and input not stored as the current input.

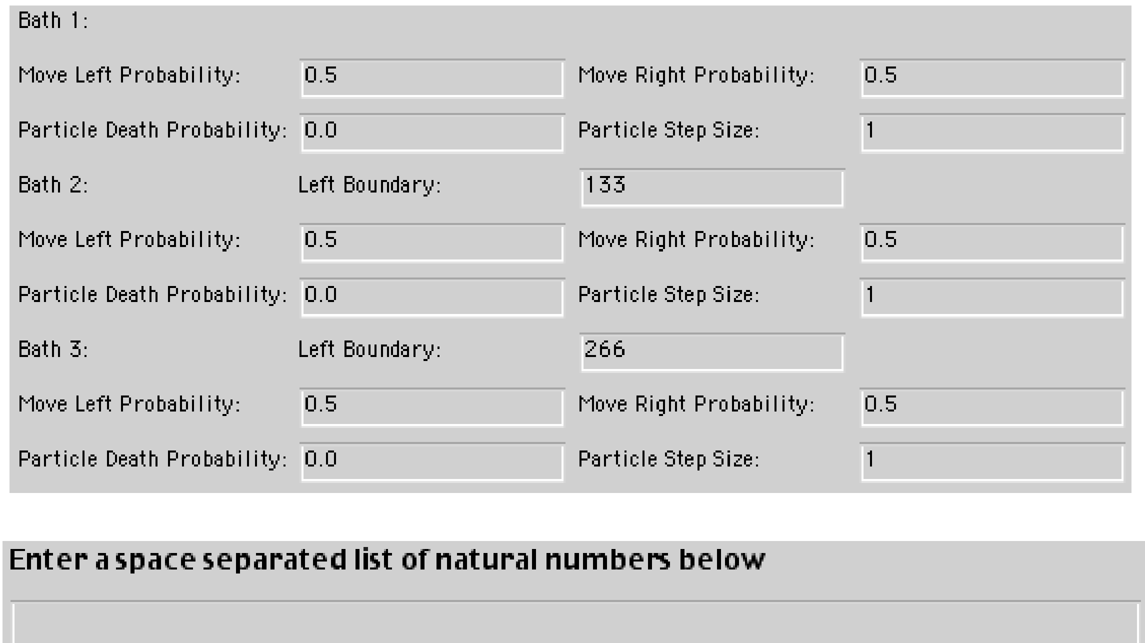


Figure 4-5: Sample layouts of input banks, simple and complex

ing the panel's layout to `java.awt.BorderLayout` and installing the `VizView` and the `VizDebug`, two interface elements which will be explained later. `standardSetup()` lays the groundwork which makes a visualization panel look like a visualization panel, supporting a standard interface which makes visualizations easy to use from one panel to the next.

The constructor then calls `public void DefaultTeacher.setup()`. Currently, this sets up a user interface with no controls, and therefore, you should override it in your own visualization panel. Your `setup()` should create a panel to be used as an input bank and attach it to the south of the main panel. This input bank should contain any interface components necessary to collect appropriate input for the algorithm, as well as concise labels explaining their purposes. The components should be laid out neatly, possibly using a layout type of `GridLayout` or `GridBagLayout` (Figure 4-5). The input bank should also contain, at its bottom, a panel created by calling `setupControlPanel()`, which is described in the next section.

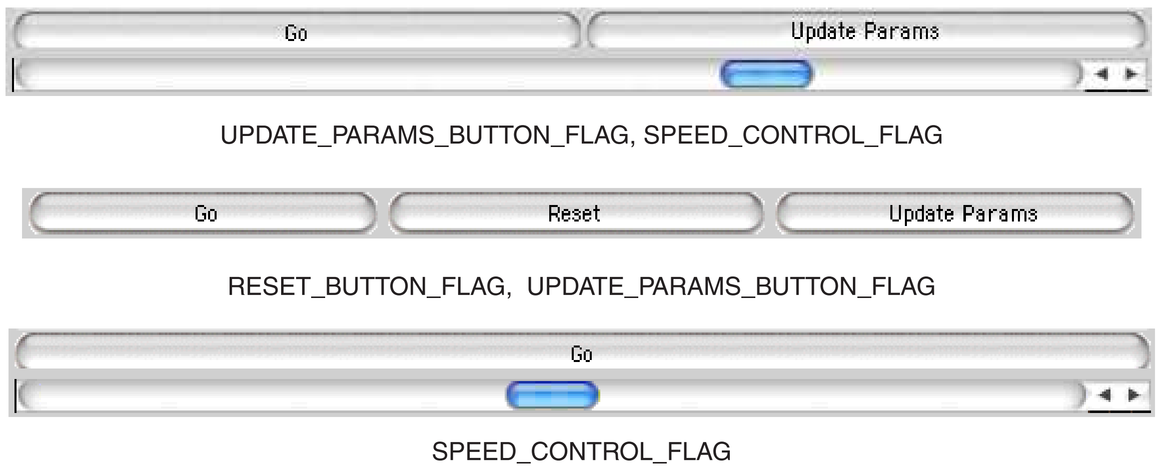


Figure 4-6: Sample control panel configurations

The Control Panel

The control panel is responsible for starting, stopping, resetting, updating and setting the speed of the algorithm. It is generated by calling the function `protected Panel DefaultTeacher.setupControlPanel(int flags, String problemNames[])`. The value `problemNames` represents a list of different versions of the module and is only relevant when dealing with simulations (See Chapter 5). For a non-simulation visualization panel, it should be `null`. The value `flags` indicates the desired features of the control panel and should be a bitwise or of the following constants, defined in `DefaultTeacher`: RESET_BUTTON_FLAG, UPDATE_PARAMS_BUTTON_FLAG and SPEED_CONTROL_FLAG. Examples of various control panel configurations are given in Figure 4-6.

The only button present in every control panel is the *Go* button. Pushing the *Go* button results in a call of `public void DefaultTeacher.GoButtonActionPerformed(java.awt.event.ActionEvent e)` where `e` is the event during which the button was pushed. This function checks if the algorithm is running, and if not, calls `startAlgorithm()` to get it going. If a control panel has been setup, `startAlgorithm()` automatically changes the label on the *Go* button to read, “Stop”. Thus, if the button is clicked and `GoButtonActionPerformed()` finds that the algorithm is already running, it halts the action by calling `stopAlgorithm()`, which changes the button label back to "Go". Note that to ensure the accurateness of the *Go* button label, the `VisualizationThread` calls `stopAlgorithm()` as soon as it returns from calling `runAlgorithm()`.

The reset button is only included in the control panel if RESET_BUTTON_FLAG is passed in as a parameter. When clicked, the reset button calls `public void DefaultTeacher.ResetButtonActionPerformed(java.awt.event.ActionEvent e)` where `e` is the event during which the button was pushed. By default, this function calls `DefaultTeacher.resetAlgorithm()`, which is currently the same function as `stopAlgorithm()`. You should override `ResetButtonActionPerformed()` and add any commands necessary to reset the state of your visualization panel. This is most useful in simulations (See Chapter 5).

The update button is also activated by passing in the appropriate flag: UPDATE_PARAMS_BUTTON_FLAG. When clicked, this button calls `public void DefaultTeacher.UpdateParamButtonActionPerformed(java.awt.event.ActionEvent e)`, which then calls `protected void DefaultTeacher.updateParams()`. `updateParams()` should poll the panel's user interface elements and send the contents to the appropriate module by means of `SendToModule()`. For example, in the line clipping algorithm, the user might type in the bounds of the clipping rectangle and then send them to the module with the update parameter button. This is most useful for updating the parameters of an algorithm while it is running.

The speed control is created by passing SPEED_CONTROL_FLAG to the `setupControlPanel()` function. This creates a scroll bar at the very bottom of the screen which can be used by the student to set how fast the animation should run. The scroll bar ranges from a minimum value of zero to a maximum of one hundred. Your visualization can query the desired speed with the routine `public int DefaultTeacher.getSpeed()` or force a certain speed with `public void DefaultTeacher.setSpeed(int speed)`. It should be noted that the speed slider has no direct control over the speed of your animation. Therefore, it is up to your code to make sure the selected speed is respected, by calling `getSpeed()` and delaying each frame of animation an appropriate amount. Zero is the slowest speed and one hundred is the fastest, which should result in no delay at all.

VizDebug

The control panel allows a user to make requests of the visualization, but it is through the `VizDebug` that the `DefaultTeacher` communicates with the user. The `VizDebug` controls both the call trace area and the witness area. The call trace area contains a welcome message, explaining the purpose of the visualization panel and describing the student's goal, which is usually to gain understanding of the algorithm. When an algorithm is running, the call trace area should display important transitions as they happen. Usually, call trace messages will describe what is happening in an animation. For instance, in a bubble sort algorithm, the call trace would report each time two elements are switched. When the call trace area fills up, a user can clear it by clicking the "clear" button at its top, deleting all information except the welcome message.

The `DefaultTeacher` provides access to the `VizDebug` through several functions. The first of these is `public void DefaultTeacher.setCallInitial(String callString)`, which sets the welcome message displayed in the call trace box to the string `callString`. Calls can be added to the call trace box with `public void De-fault-Tea-cher.addCall(String callString)`, where `callString` represents the call to add.

The witness box operates in a similar manner. If a witness detector returns a string as a witness object, your panel should call `public void DefaultTeacher.-setWitnessText(String witnessString)` with `witnessString` as the witness object. This will set the text in the gray area below the call trace to the witness string. If you wish to append something onto the text in the witness box, instead of replacing it by using the above function, call `public void DefaultTeacher.addWitnessText(String witnessString)` where `witnessString` is the string to append.

4.4.3 Auxiliary Support for Manual Mode

The `DefaultTeacher`'s last function, though not a primary one, is to assist with bookkeeping in manual mode. Often, it is necessary to separate the actions of manual mode into distinct steps, or groups of steps. For instance, in the manual mode of a visualization for Bresenham's line drawing algorithm, the visualization panel would first need to receive the endpoints of the line to be drawn. Then, it could begin accepting user input regarding which pixels should be highlighted. However, it is convenient to use the same interface to indicate points in both cases, so the visualization panel must keep track of which information it is gathering. Step one is to gather the input to the algorithm and step two is to collect pixels which the user thinks should be highlighted.

To support this, `DefaultTeacher` provides functions to keep track of the step the panel is running. The method `public void AlgorithmVisualization.setManualStep(int step)` sets the current step to `step` and the method `public int AlgorithmVisualization.getManualStep()` fetches it. For convention, the step which garners input to the algorithm should be numbered zero, and all further steps should be successive positive integers.

4.5 The VizView

The `VizView` is the center component of the visualization panel. Through it, the panel sends graphical information to the user. When necessary, the `VizView` may also act as a large user interface element, relaying clicks and other events to the visualization panel. Often, a `VizView` is not responsible for drawing itself, but is just a conduit for the `AnimDataType` to use, explained below. However, it does contain a set of painting and buffering tools to be used when it does need to draw.

4.5.1 Painting a VizView

By default, all painting to a `VizView` is double buffered. The class maintains a `java.awt.Image` in `VizView.mBuffer`, constantly adjusted to match its dimensions. When the Java toolkit calls the `VizView.Paint(Graphics g)` method, it responds by copying the image to the screen. Thus, all drawing which your `VizView` does should be done directly to the `mBuffer`.

The `mBuffer` is maintained at the correct size by calling `public void VizView.-ValidateBuffer()`. `ValidateBuffer()` checks to make sure the buffer exists, and if not, creates it. It then ensures it matches the size of the `VizView` and if not, adjusts it. Finally, it prepares the background by calling `DrawBackground()`. Since `ValidateBuffer()` is responsible for making sure the `mBuffer` exists and is the correct size, your drawing routine must call it directly before drawing anything. Failure to do so may result in a seemingly random `NullPointerException` being thrown up the chain of command.

`Public void VizView.DrawBackground()` is responsible for preparing the buffer to be displayed, and is called every time the buffer is erased, or before an empty buffer is copied to screen. By default, it does nothing. However, if you require the center panel of your visualization to display a certain graphic, even when no algorithm is running, then you should override this method. For instance, in the Bresenham line drawing algorithm, it is useful to display a grid on the screen, so the user knows which pixel he is clicking when he selects endpoints. This grid is drawn by `BresenhamVizView.DrawBackground`.

It is interesting to note that `DrawBackground()` draws to `mBuffer` and therefore calls `ValidateBuffer()` first, as mandated above. This seems like endless recursion, since `ValidateBuffer` calls `DrawBackground()`. However, `ValidateBuffer()` only calls `DrawBackground()` if it find that it must adjust the `mBuffer` in some way and risk erasing the background image. Thus, a call to `ValidateBuffer()` would first fix the `mBuffer`, then call `DrawBackground()`, which would call `ValidateBuffer()` to check the `mBuffer` again. This time, though, the `mBuffer` has already been adjusted, so `ValidateBuffer()` returns immediately and drawing continues successfully.

4.5.2 Obtaining User Input

To collect input, the `VizView` relies on the somewhat primitive `public boolean handleEvent(Event e)`. This is a standard method of gathering mouse clicks and drags, and the `e` received is a `java.awt.Event` and interpretable in the typical method. Unfortunately, `handleEvent()` has been declared obsolete from the official Java language, and may be removed entirely soon. Therefore, if you create a new `VizView` for your visualization, it is recommended that you use individual event handlers, as are standard in JDK 1.2.

To send information back to the visualization panel, `VizView` stores a pointer to its `DefaultTeacher` in `VizView.mTeacher`. That way, any input received from the user can be sent through the teacher, to the module. This is why the two way dependency is needed between `DefaultTeacher` and `VizView`in the module dependency diagram in 4-2.

Sometimes, the `VizView` may even substitute for the *Go* button in the control panel. For example, in the line clipping algorithm, the algorithm should run automatically, as soon as the user has drawn a line to be clipped. To accomplish this, have the handler for mouse up events call `mTeacher.setCurrentInput()` on the user's line and then `mTeacher.startAlgorithm()` to set the algorithm in motion. If a "Go" button existed, the user may even be impressed as he watches the label change from "Go" to "Stop" as the process begins.

4.5.3 Often No Need to Inherit

Since the brunt of the algorithm animation is done by the `AnimDataType` and most user interface is handled by the `DefaultTeacher`, it is usually unnecessary to subclass the `VizView`. This seems slightly unintuitive, since the `VizView` is the big black square in the center of the visualization that should display completely different information for each module. However, the `VizView` really only needs to be subclassed and overridden if you want a constant background that is not black [4], or if you want to provide an intricate mouse interface, not supportable by text boxes and buttons. Following these guidelines will ensure simplicity and minimal code size.

4.6 Animated Data

As mentioned above, the primary shortcoming of several existing algorithm animation systems is the lack of transparency to the algorithm coder. That is, a programmer trying to swap two variables would have to swap them in her data, and then call a function to trigger a swap animation. This seemed sufficient for a developer creating an algorithm with the purpose of teaching it through animation. However, for Educational Fusion, we felt this was an unacceptable burden to place on a student trying to learn an algorithm and animate it at the same time.

Through Educational Fusion's history, we have considered three solutions to this problem. The first, and most difficult solution to implement, is to build some kind of state monitoring procedure. This procedure would monitor all variables marked as animated and draw their state, in a suitable form, on a `VizView`. Thus, an algorithm's progress could be monitored by graphically observing the state of all its variables at all times. Even better, the omniscient procedure could examine changes in state and determine what actions caused the state change. It could then create animations to reflect those actions and display those along with the state.

The above solution is the utopia of algorithm animation and, quite clearly, rather difficult. Although not necessarily impossible, its creation would be a task too daunting for an Educational Fusion developer and perhaps might be the result of a dedicated professors life's work. However, simpler solutions exist, and Educational Fusion employs them.

[4] If you want to avoid the ugly black box as soon as your visualization panel appears, try creating your animated data type in the constructor of your panel and drawing it. This will copy it to the `VizView`'s offscreen buffer which will be copied to the screen as soon as the panel is displayed. This is what allows the Random Walk Simulation (explained in Chapter 5) to give an initial display of the particles without needing to override its `VizView`.

An alternative solution is to have modules output interesting events, similar to the way ZEUS animates [Bro91]. For instance, instead of having to swap data and then tell the animation system to display the swap, a sort module might simply output "Swap elements 3 and 4". Then, a visualization panel would receive the series of swaps and perform the operations as well as animate them. The result would be a sorted list in the visualization panel's memory, and an animation of the sorting process on the screen.

This is exactly how Educational Fusion worked in the past. It was an effective solution, and allowed for easy development, but it had important disadvantages. First, a module required a separate output port for each interesting event it could send. Thus, an involved algorithm with an intricate animation would require many output ports and an excessive amount of complexity. Second, a module could not accurately reflect the data flow of its algorithm, since it was not outputting actual data. On the concept graph, a sort module could not be accurately represented as taking an unsorted vector as input and sending a sorted vector as output. Instead, a vector generator would pass the sorter an input vector and the sorter would output a series of swaps necessary to sort the vector. Then, an output "assembler" module would need to receive both the series of swaps and the original input vector to assemble the mishmash into a sorted vector which could then be passed to another module.

A third solution, and the one currently used by Educational Fusion, is to infuse the data structure with animation capabilities. Using Java's superior permissioning abilities, a developer can restrict access to data as tightly as desired. Once all data access methods are controlled, each one can be enhanced with hooks into animation routines which display the effect of the accessor. For example, for a simple sorting algorithm, a vector can be created which allows no access to its data except through `getElement(int i)`, which returns its `i`th element, and `swap(int i, int j)` which swaps elements `i` and `j`. Then, these functions can be overridden to perform not only their functions on the data, but to animate the examination or exchange of the data's elements.

4.6.1 Creating the Animated Data Structure

When creating the animated data structure, it was necessary to decide which type it should subclass. An animated structure could inherit from either the class which it most closely resembled or a special `AnimDataType` which contained functions useful for animation. Unfortunately, due to Java's lack of support for multiple inheritance, it could not inherit from both. One immediate alternative was to make `AnimDataType` an `interface` and have an animated data structure `extend` a similar structure and `implement` the `AnimDataType`. However, this would mean `AnimDataType` could define no member fields, nor could it contain the bodies of its methods, making excessive code duplication necessary.

Eventually, we settled on making each animated structure a direct subclass of the `AnimDataType` class. This has two advantages over alternatives. First, it allows the code reuse mentioned above. Second, it makes it much easier to restrict permissions to unwanted accessor functions. For instance, if an animated vector called `AnimatedVector` were derived from `Java.util.Vector`, it would be difficult to limit access to the public method `java.Util.Vector.setElementAt()`. By deriving `AnimatedVector` from `AnimDataType`, there is by default no access to any data. In fact, to complete the `AnimatedVector`, it is necessary to add an extra field to represent the data itself. By convention, each animated data type should add a private field named `mSelf` which is used to store the main data. Thus, `AnimatedVector` has a field named `mSelf` of type `java.util.Vector`.

To provide access to any methods of the `mSelf` class, simply create a wrapper in the animated structure. That is, `AnimatedVector` has a method `public Object AnimatedVector.getElement(int i)` which just returns `mSelf.getElement(i)`. This way, access is provided to data through only the channels which the developer intends; nothing is left unanimated by accident.

4.6.2 Initializing

To initialize itself, an `AnimDataType` needs to know where it should display its output. This is done by calling the method `public void setViews(VizView vv,VizDebug vd)` for which `vv` is the `VizView` on which the structure should draw and `vd` is the `VizDebug` to which the structure should record its actions. Usually, these are just the `mVizView` and `mVizDebug` of the `DefaultTeacher` which created the structure.

A structure also needs to know which witness detector it should use, so it can look for witnesses while animating. Set the witness detector by calling `public void setProof(BaseWitProof proof)` with the desired witness detector. Usually, this is `mProofModuleBean` of the owning `DefaultTeacher`. Note, you should only set the witness detector if witness detection is turned on. If not, the `AnimDataType`'s witness detector will remain `null` and it will not look for witnesses.

After receiving a witness detector, the structure also needs to save a copy of its original input, so the witness detector can use it to judge the output. This is accomplished with public void AnimDataType.saveOriginalInput(Object in). If the algorithm modifies the input to produce the output, you should save a clone of the input, so the witness detector can have an unmodified version.

Finally, if the visualization has a speed control, the `AnimDataType` requires its value. This is set with `public void AnimDataType.setSpeed(int speed)` and obtained inside the animation routines with `public int AnimDataType.getSpeed()`.

4.6.3 Drawing

An animated data structure should be able to draw itself at any time, accurately reflecting its state. The call to `setViews()` takes care of validating the `VizView`'s frame buffer, and saves a copy in `AnimDataType.mImage`, so all drawing can safely be done there. For reference, `public int AnimDataType.getWidth()` and `public int AnimDataType.getHeight()` return the pixel width and height of the image, respectively.

Before drawing, the `AnimDataType` should always check `public void AnimData-`

Type.getIsAnimating() to make sure it is supposed to draw. If getIsAnimating() returns false, then the AnimDataType is just being used to pass data in a concept graph and is not associated with a VizView or visualization panel. In this case, mImage is null and any drawing will result in an error.

The job of drawing an AnimDataType is handle by the method public void draw(). You should override this method in your custom type and draw the contents of the data structure in some meaningful manner. For an example, see line 247 in Appendix D.

Drawing the data structure provides an excellent opportunity to display errors or incorrectnesses within it. As for the vector sort visualization described in chapter 3, out of order elements might be drawn in red while correctly placed elements are drawn in green. To accomplish this, the animated data structure allows access to the witness detector through the function public Vector AnimDataType FindWitnessList(Object output). This method uses the input stored with saveOriginalInput() and the output object to obtain a list of incorrect elements in the output. The draw() method can then search this list and visually display the errors to the user.

4.6.4 Animating

To animate the data structure, all that remains is to script short segments of action for each overridden accessor function. For instance, a swap in a sort could be animated by having the two elements circle around each other and then land in the appropriate places. All drawing should be to the AnimDataType.mImage to prevent flicker. However, when a frame is complete, you should copy it directly to the screen. The initialization function, setViews stores a pointer to the screen's graphics object in AnimDataType.mGraphics so you can copy the frame by calling mGraphics.drawImage(mImage,0,0,null). After each frame is copied, you should wait an appropriate length of time, based on the speed obtained with getSpeed(). To assist this process, use the method public void AnimDataType.delay(long time) which delays roughly time milliseconds. When the speed control is centered, each animated function should take about one second to display.

To help increase the student's understanding of the animation, the data structure should report each data access into the call trace of the `VizDebug`. To do this, use `public void AnimDataType.addCall(String s)`. If a call to an accessor is later retracted, perhaps by a user in manual mode, use the method `public void AnimDataType.removeCall(String s)` to remove just one call in the middle of the call trace. For instance, if a student is clicking to highlight pixels in manual mode of the Bresenham algorithm module, he can click an incorrect pixel to unhighlight it. The panel then uses `removeCall()` to remove the call which highlighted the pixel from the call trace area.

4.6.5 Copying

Occasionally, it may be necessary to copy an animated data structure, possibly when saving it as original input. If the copy is to draw itself, it must somehow get access to all the drawing information and buffers of the original. To do this, call `public void cloneAnimStuff(AnimDataType other)`. This will copy all relevant information, including whether to animate or not, from the `other` structure to the current one.

4.6.6 Linking Your Modules with Animated Data, and What to Do When You Cannot

For the Educational Fusion Visualization system to be complete, modules must be designed to use animated data structures. This means that when a new module is defined as described in other documents [Boy97], it must take animated data as input and output. For instance, a vector sort module would simply take in and put out an `edufuse.modbase.AnimatedVector` and not a `java.util.Vector`. Since `AnimatedVector` is not derived from `Vector`, the two are not interchangeable. To run the sort, a visualization panel would create an appropriate `AnimatedVector`, store it as input with `setCurrentInput()` then run the algorithm with `startAlgorithm()`. Animation is automatic each time one of the `AnimatedVector`'s animated methods is called.

Finally, it is sometimes not necessary or even possible to use animated data structures as links to modules. For instance, if an algorithm creates an output from scratch, rather than modifying its input, it is not clear how to link the modules with animated data. The new structure which the module creates cannot be linked to the `VizView` without violating many abstraction barriers. In this case, the new structure should be created by the visualization panel and passed into the module as input, already containing the `VizView` information, or the animated data type might reside solely in the visualization panel and receive pieces of data individually from the module. That is, the visualization panel should create an animated data type and link it to its `VizView`. However, the module should operate in the old Educational Fusion manner, just spitting out important commands, such as "swap items 3 and 4" or "color pixel at [5,6] red". The visualization panel should then apply these to the animated data structure and animation and data mutation would again occur as one.

Chapter 5

Simulations and Virtual Labs

Simulation and virtual lab hosting is Educational Fusion's newest feature. With this addition, we hope the usefulness of Educational Fusion will extend past the borders of computer science into the realms of chemistry, biology, electrical engineering, physic and many other sciences.

This chapter starts out by explaining what it means to host a simulation in Educational Fusion. It then gives a short example of simulation use and goes on to explain the changes in Fusion's framework which allow for easy hosting. Finally, it discusses the importance and ability of Fusion to host a virtual lab- a module which looks like a simulation, but is in fact just a front end for a network connection to a real machine performing experiments on real components and returning actual results.

5.1 What Does it Mean to Solve a Simulation

As stated above, Educational Fusion is focused on presenting a problem, teaching the concept of the problem, and receiving a solution to be graded by a TA. To adapt Fusion to host simulations, it had to be decided what was meant by "presenting the problem" or "receiving a solution" to a simulation.

The first possibility revolved around the fact that a simulation is just a complex algorithm. It accepts input and exports output the same way an algorithm would, while it imitates the system which it simulates. So, the reference algorithm presented to the student could be a working version of the simulation, which the student is asked to implement in his own code. The visualization panel would provide an interface to the simulation and the witness detector would let the student know if his simulation performs properly.

This seemed an excellent way to teach the student about the simulation and the actual system involved. He would need to learn the internal equations which represented the system in order to implement them in his code. He would need to use the simulation extensively to understand its properties and test his own implementation. Unfortunately, this approach is too centered around programming and computer science. Even if the professor first provided a student with the necessary equations and formulas, it would be a tough programming assignment to ask her to implement a full simulation of a real system. This is acceptable for a computer science student, but it was our goal to make Educational Fusion simulations usable by students of all disciplines. It is unreasonable to ask a chemistry student to program a mass spectroscopy simulator just to show she understands the concept.

The second possibility, and the one currently in use, is to ask students to use their simulations to produce specific results. That is, instead of using their implementation to code an entire simulation, the students must write small snippets of code which set proper inputs to the simulation, designed to produce requested outputs. For this option, we assume that students of all sciences will have some rudimentary Java knowledge. Even if this is not so, modules can be designed which accept their input in manual mode, requiring no student coding at all. After the input is delivered, responses from the witness detectors indicate to the students when they are right and have successfully "solved" the simulation.

Finally, a third possibility is to ignore Fusion's problem/solution model entirely. This would amount to a homework assignments involving the simulation being run as they were before the simulation was ported to Fusion. Results would be submitted by hand, instead of electronically with the "submit" button. Although this does not take advantage of some of Fusion's qualities, it still gives the simulation the benefit of being platform independent and running in a strongly collaborative environment.

For instance, students can still use the chat room to see which of their friends are working on the same simulation and then ask them questions.

5.2 A Sample Simulation in Educational Fusion

Educational Fusion currently includes a random walk diffusion model, often used in MIT's biophysics courses [Wei+92]. In the simulation, there are three adjacent baths which contain a large number of particles. For each bath, the students can set the size, life expectancy of a particle and the probabilities that a particle will move left or right. They can then set the simulation in motion, using standard Fusion controls, and attempt to establish requested particle distributions by manipulating the parameters, either through their code or through typing numbers into text boxes in manual mode (Figure 5-1).

Upon starting up the visualization, the student (perhaps Dianne again) is greeted with a message explaining that she should create a particle distribution in which there are no particles in the center bath. She thinks this could easily be accomplished by giving all baths a strong probability to move their particles to the right edge of the screen. Thus, all particles would end up in the far right bath and the center would remain empty. She switches to manual mode at the top of the screen, types in the appropriate numbers and clicks "Go." To be sure she is correct, she also enables witness detection.

Dianne watches in dismay as her particles turn red, indicating they are incorrectly placed. In addition, the message in the witness area tells her her distribution is incorrect. She examines the instructions again and finds they state that the center of mass of the particles should remain in the center bath, at least 50 units away from all other particles. In addition, the particles should reach the desired distribution from any starting layout. After thinking a while, Dianne comes up with a solution and types in new parameters. Without stopping the simulation, she clicks "update parameters" and the particles swiftly change their behavior. She watches a little blue square, drawn by the witness detector to indicate the center of mass, drift over

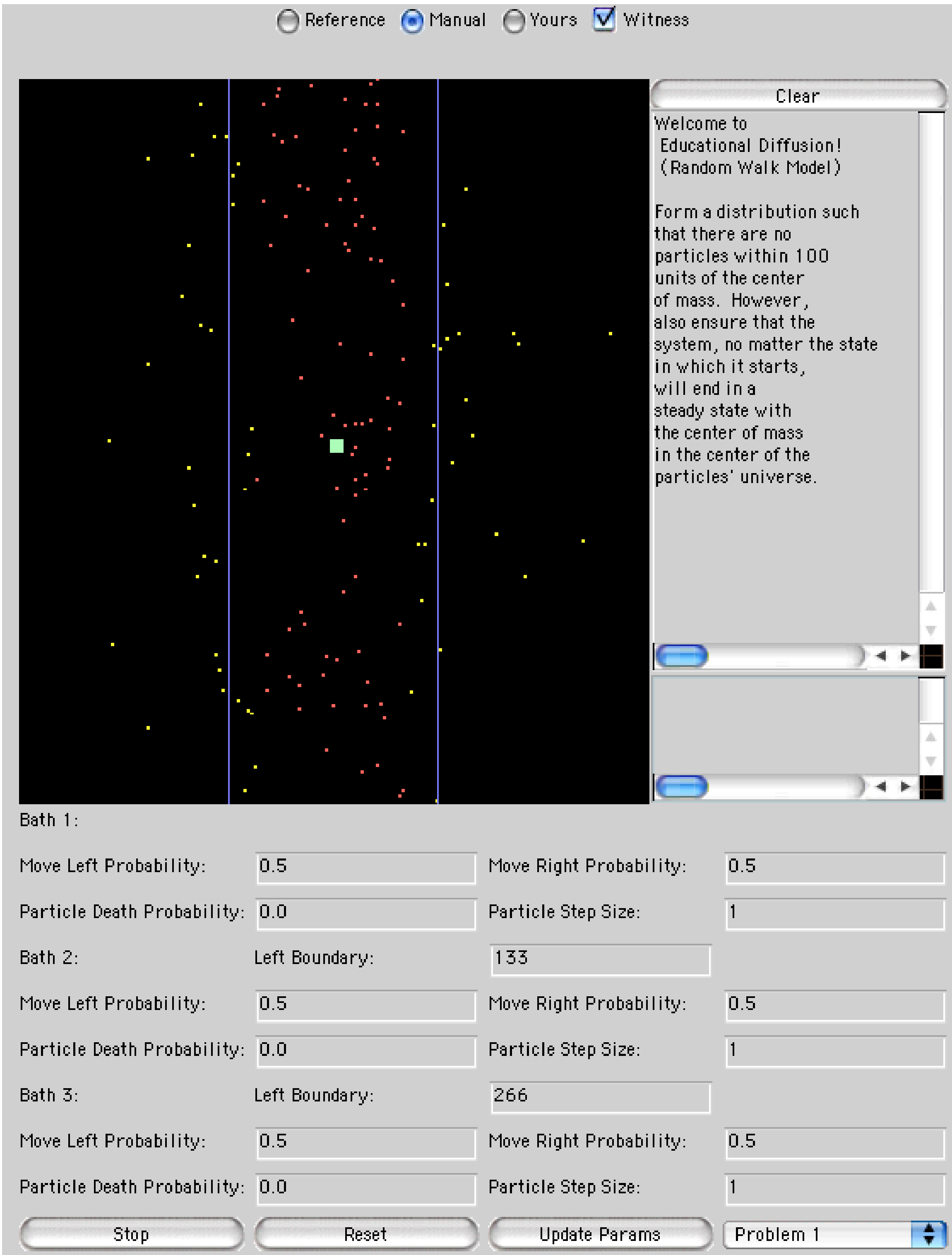


Figure 5-1: An Educational Fusion simulation of random walk diffusion

to where it belongs. Slowly, all the particles turn yellow, indicating they are far enough away from the center of mass, and the witness area congratulates Dianne for completing the first part of the simulation. Having solved the problem, she writes the parameters from the panel into a short procedure which she submits as her answer for problem one.

Next, Dianne selects problem two from the bottom of her screen. This time, she is asked to set the particles in the center bath oscillating, as if someone were shaking the bath. She realizes that this will require modifying the underlying simulation, as she cannot adjust the parameters quickly or accurately enough by hand to create the oscillating phenomenon. She writes a quick routine, based on a sine wave, which oscillates the parameters of the center bath. She switches to "yours" mode in the visualization and observes her routine generating input. She watches in delight as the text boxes which she used in manual mode now update constantly to reflect the parameters which her procedure is generating. The witness detector turns her particles yellow, letting her know they are oscillating correctly[1]. She is then asked to determine the connection between the period of oscillation and percentage of particles in the center bath. She can submit this response as a comment with her snippet of code.

5.3 Modifications to Witness Detectors

Because a simulation is so similar to an algorithm, witnesses remain somewhat unchanged. However, a simulation can have a broader variety of problems based around it than a typical algorithm. For the random walk module, the student can be asked to produced a variety of different distributions. Without adapting the witness detection system to simulations, a whole new module with a new witness detector would be needed for each requested distribution. Therefore, we introduced the idea of differ-

[1]Here is an interesting example of the power of witness detectors. Instead of examining the actual distribution to test for oscillation, which would be rather difficult, the detector examines the input parameters. If they match the parameters which it expects, it decides the distribution must be correct.

ent "problems" into a single module. Once a detector knows which "problem" the student is attempting to solve, it can look for witnesses appropriately.

To set the problem number or name, use `public void BaseWitProof.setProblemNumber(int n)` or `public void BaseWitProof.setProblemName(String s)` respectively. To query the number or name from within a witness detector, use `public int BaseWitProof.getProblemNumber()` or `public String BaseWitProof.getProblemName()` respectively. Usually, you should not need to set the problem number or name, as it is taken care of automatically by the visualization panel.

5.4 Modifications to Visualization Panel Framework

Luckily, the visualization panel was already well adapted to collecting input and displaying simulation results. However, some modifications are needed to deal with the addition of multiple problems per module. For this purpose, the method, `public Panel DefaultTeacher.setupControlPanel(int flags, String problemNames[])` accepts an array of the names of all problems supported by this simulation. It then adds a popup menu to the control panel to allow a user to select a desired problem (Figure 5-1). When a student selects an option from the menu, the new problem name and number are sent to the appropriate module and witness detector.

To help the student know what kind of distribution he should create, the framework also provides `public void DefaultTeacher.setInstructions(String name, int number)`. When a problem is selected, this function is called with the problem name in `name` and the problem number in `number`. You should override this function to use `setCallInitial()` to change the welcome message of the call trace to instructions explaining the point of the problem.

Both `setInstructions()` and `setProblemNumber()` are called by the method `public void DefaultTeacher.problemItemPerformed( java.awt.event.ItemEvent e)`. If you wish anything else to happen when a student selects a problem, you should override this function. You can access the selected problem number by calling `DefaultTeacher.mProblemChoice.getSelectedIndex()` or the problem name by calling `DefaultTeacher.mProblemChoice.getSelectedItem()`.

One interesting change should be noted here regarding the way in which a visualization of a simulation is expected to perform. For a typical algorithm, the visualization panel is used to send input to the reference implementation, student implementation or student manual mode, which then performs the algorithm. For a simulation, it is the reference, student, or manual mode which provide the input to the simulation algorithm. Therefore, in reference or "yours" mode, a student would input very little from the visualization panel. He would just click "Go" and the simulation would query the appropriate implementation for input and then the run the simulation on that input. However, in manual mode, a full panel should be set up, as in Figure 5-1, allowing the student to create any input, just as the reference or student implementation might, and then send it to the algorithm by pressing the "update parameters" button.

A student may become confused if he is running a simulation, types some new parameters into the manual box, presses "update parameters" and sees no change. This is probably because he is running in reference or "yours" mode and the visualization panel is ignoring the manual input. To avoid this situation, the developer of the panel may want to automatically switch to manual mode when the "update parameters" button is pressed. This is done by calling `public void AlgorithmVisualization.switchToManualMode()` within `DefaultTeacher.updateParams()`.

5.5 Modifications to BaseModule

When adding simulation abilities, the changes to the module system itself were the most significant. A module now needs two custom procedures, one to set the parameters according to the problem number and the other to run the actual simulation. Because simulations are slightly more complicated than traditional algorithms, Educational Fusion suggests a few conventions for simulation modules. Each simulation should have one input, consisting of parameters suggested by the visualization panel. In addition, it should have two outputs, one for the new set of parameters created by the parameter generation function, and the other for any output from the simulation.

Upon receiving a block of parameters on its input channel, a simulation module should first check if it is being run in manual mode. It does this by calling `public boolean BaseModule.isManualMode()`. If the simulation is in manual mode, it means the block of parameters received were input directly by the user and should be observed and sent straight to the simulation function. If the simulation is not in manual mode, then the parameter generation function should be called, to modify the parameters based on the code written by the user. Next, the new parameters should be passed to the simulation function, and sent back to the visualization panel through the parameter output. Upon receiving the parameters, the visualization panel should exhibit them, so the user can make sure the code is working correctly. Finally, the simulation results should be sent to the visualization panel for display.

5.6 Virtual lab and Parameter Passing

As far as students can tell, a virtual lab is just an elaborate simulation, possibly with more dials. As far as the developer is concerned, a virtual lab is easier to implement than a simulation, because there are no complex equations or hidden system models which need be programmed into the module.

As a demonstration of its virtual lab hosting ability, Educational Fusion includes weblab, an electrical engineering lab based on the work of MIT Professor Jesus del Alamo [Ala99]. The lab allows the exploration of voltage and current responses in a variety of transistors connected to an hp4155 signal analyzer. Because weblab is so complex, it was not easily adaptable to Fusion's problem/solution model, and was best left as an exploratory laboratory, without the benefit of Fusion problems and witnesses.

However, weblab was easily ported into Fusion using the new simulation conventions and framework. It is broken up into two sections, with the interface built into the visualization panel and the connection to the weblab server supported by the hp4155 module. Because of this separation, weblab has taken on a collaborative nature which would have been difficult without the Educational Framework. Fusion's networking architecture allows for the sharing of information and interconnection of modules between users [Tor98]. This solves one of the most significant problems which Professor del Alamo was having with weblab. Because the weblab interface is so complex, students would often become confused and require technical support from TAs before they could continue with their labs. Often, support through e-mail was not enough, because a TA could not see a student's screen to easily examine every parameter a he was using. Sometimes, a student would accidentally type something into an input box without even noticing and this would cause his tests to perform incorrectly. Since he was unaware of typing the error causing input, there was no way he could relate it to a TA.

With Fusion, a TA can now directly receive the block of parameters which a student is sending to the hp4155 module [2]. This means a TA can see the same panel a student sees, and diagnose exactly what is wrong. This will ease the burden of both TAs and students in the future, demonstrating the true potential of long distance collaboration under Educational Fusion.

[2] This information can actually be sent a variety of places beside a TA. For instance, a thread could be added to the hp4155 module which logs tests and caches the results, allowing the module to quickly return results for popular test without querying the analyzer.

Chapter 6

Conclusion

So far, we have given an overview of Fusion's layout and discussed previous work related to on-line learning, visualization and self-checking code. We have also discussed witness detection, algorithm visualization and simulation hosting in greater depth. This chapter concludes the thesis by examining recent work and checking how useful it really is. It then suggests possibilities for future work and finishes by summing up the goals of Fusion.

6.1 Does all this stuff work?

To determine the usefulness and usability of the new standard visualization environment, and obtain suggestions for future development, we conducted a usability test on the vector sort module. The experiment was not sufficient to prove anything, because it consisted of one subject, but it provided an opportunity to study the visualization panel in use by a real student. The subject, whose name was actually Dianne, was aware of the syntax of Java, but had not programmed in seven years. She also claimed to have never programmed a sort algorithm before, nor had she heard the term bubble sort. After receiving a brief explanation of the three modes of the visualization panel, she set out to implement a bubble sort.

At first, she watched the reference mode in action, several times, inputting different lists every time. After roughly 30 minutes, she believed that she had figured out how the sort worked. However, she suggested that it would have been much easier to learn the sort if, in addition to allowing her to choose her own input, the panel recommended input which would best display the workings of the sort.

Next, she switched to the edit mode and coded a solution. She switched back to visualization mode to test her answer and found that her answer was incorrect. She did not need witness mode to realize this, but turned it on for future reference. Even though her solution was incorrect, the algorithm she coded still animated, and she was happy to see that her custom solution would animate even when not correct. She then attempted to debug her code by watching the animation, but found that the animation was going too fast for her to follow, even at the slowest speed setting. She suggested that this be fixed by either lowering the slowest speed, adding a pause or single step button to the control panel, or printing out the state of the vector after each swap operation, so she could scroll back in the call trace area and follow the operations when the code had completed.

Finally, Dianne managed to fix her algorithm to sort the list in descending order. However, the witness detector told her she was wrong, because it was built to check for ascending order. By oversight, the direction in which the list would be sorted was not stated, and has now been added as a comment in the user template.

Dianne mentioned that she had a little trouble using the animated data structure because all the necessary methods were not described in the comment in the user template. The `swap` and `elementAt()` methods were clearly documented, but the developer had neglected to mention the `size()` method which returned the length of the vector. This confused Dianne, who, as a student, put her faith in the developer and just assume that she didn't need to know the length of the vector to sort it, and sought some other solution. Unfortunately, there was no other solution and she lost a great deal of time because the animated data structure was incompletely documented. This should serve as a warning to the Fusion module developer to always document the user template as completely as possible, and to test the modules before delivering them to trusting students.

6.2 Suggested Future Work

Although this is the end of this thesis, it is surely not the end of Educational Fusion development. For Fusion to become a world wide educational aid, it needs some improvements and touch ups to help make it more friendly and more powerful to the non-computer-oriented world. Some of these improvements are suggested below.

6.2.1 Data Naming System

At the moment, modules and visualization panels identify data strictly by examining the port on which the data arrived. This make for messy lookup tables and meaningless numbers in a situation where higher level abstraction should be used. Ideally, when data arrives, it should have a tag attached, identifying it by name, so `fusionActionNoQueue()` procedure can be skipped and the data can be sent directly to `AlgorithmResponse()` which could act based on the name of the received object.

To do this, there must be a field in each piece of passed data which identifies its name. All the types of passed data could be subclass of one `PassedData` type which includes this field. Unfortunately, if we changed the architecture now to accommodate this system, none of the existing modules would work, because they currently pass a variety of data types, none of which are derived from `PassedData` and none of which contain a `name` field. We could go through thousands of lines of code, updating every data type, but there is an easier way to slowly phase in data naming.

First, there is no need for a new `PassedData` class. Since most future modules will be built to pass data derived from the `AnimDataType`, we could build the data naming architecture directly into the `AnimDataType` class and insist that all future modules pass nothing but animated data. Then, when receiving data, a visualization panel or module would check if the data is derived from `AnimDataType` before calling `fusionActionNoQueue()`. If not, it would make the call and the response would proceed as usual. However, if the data is derived from `AnimDataType`, the module would call `AlgorithmResponse()` directly, passing in the name which arrived as part of the data.

Eventually, when all modules pass only animated data, the `fusionActionNoQueue()` method could be deprecated and the naming system will be automatic, raising abstraction barriers even higher and making the system even less complex for future developers.

6.2.2 Module Design Wizard

The process of creating a new module and visualization panel, although made easier by the redesign of the visualization framework, is still a complex and involved one. Currently, is is infeasible to think of creating a new visualization panel and module in under four hours. While this is a reasonable amount of time for a teacher to develop a lesson plan, it could be much reduced by exploiting the full power of the online learning environment.

There is a simple checklist in Appendix B which enumerates the steps necessary to create a new module and visualization panel for Educational Fusion. Most of these steps are simple and well defined, many of them involving finding words and replacing them with the name and location of the desired module. These steps could all be automated by a module creation wizard, activated by the `Create Module` command, given from the concept graph. Currently, this commands creates the appropriate files needed for the module, but editing them and creating the visualization panels is left up to the user.

In the future, the wizard should automatically complete all of the simple checklist steps, such as filling in the name of the module in the correct places in Educational Fusion files. Then, the wizard should ask relevant questions about the type of module to be created and help setup the visualization panel and input / output port mappings by generating some of the code in the appropriate files. The wizard could even assist in creating a new animated data type, if necessary. It would do this by asking which data type the new one should implement and then asking which accessor methods the data type should make public. Finally, for the steps which must be done by hand, the wizard would act as a guide, indicating which step must be accomplished next and checking the user's input to ensure validity upon completion.

As a terrific side effect of the wizard, module development would become platform independent and possible from within the Java enabled browser. Since the wizard would be part of Educational Fusion, it would have the ability to activate the Java compiler, already used to compile student code. For editing, the wizard could bring up the student editor, but filled with module or visualization panel code instead of student code. This way, many of the working parts of Educational Fusion could be reused, making wizard implementation a simple task, and module development easier and more self-contained than ever.

6.2.3 Demo Input System

Expecting students to learn an algorithm solely by watching it animate is a bit unrealistic. That is why Educational Fusion will not put teachers out of work: They are still necessary to explain basic concepts to the student and assist them through difficult processes. However, if Fusion becomes a world wide accessible product, or even if it is just used by very busy professors, teachers may sometimes not be as available as possible to fully explain concepts necessary to implement the algorithm. In these cases, visualization is supposed to fill in the gaps.

Unfortunately, visualization is not always useful unless students understand the algorithm enough to know what inputs they should use to watch the most useful animations. Algorithms may have special case operations which students would never guess if they do not input the correct special cases. For this reason, Fusion should have an easy way to include sample inputs with each visualization panel, allowing a student to watch preselected animations in action, as well as input their own.

To support this, the `setupControlPanel()` should take another list, indicating whether the visualization panel has included demo inputs and if so, what their names are. Then, a separate procedure, coded by the developer, would be called each time an item is selected from the demo input list. This procedure would be responsible for checking which item was selected and then filling in the user interface elements with the parameters appropriate for the selected demo input.

Once the creation wizard, mentioned above, is completed, entering demo inputs could be the last step of the wizard's creation checklist. It would take the developer to a preview version of the visualization panel, have her enter her demo inputs directly into the user interface and then click a button to take a snapshot of the user interface as a demo input. Then it would generate the code to recreate the snapshot, ask the developer for the name of the snapshot, and add it to the demo input menu.

6.2.4 Editor Enhancement

If Educational Fusion is to be the student development environment of the new millennium, the code editor needs to be more like the current editors available to professional programmers. Although it supports auto-tabbing, text-coloring and parenthesis balancing, these features work in a way unique to Educational Fusion. Their behaviors should be adjusted and made to work more like a student who has used other editors would expect. In addition, the editor needs new features, like a class browser, popup menus when methods or objects are clicked, and a debugger. This would then make programming in Educational Fusion no less enjoyable or easy than programming in any environment, making it the ideal choice for student programmers around the globe.

6.3 Goals Revisited

While working on this project, I gained much respect for and insight into the world of software learning environments. Although this was a happy side-effect of my work, it was not its main purpose, a purpose which should be kept in mind by all future developers. Fusion's purpose is to serve the community by providing a universal, platform independent, reliable, easy to use, real world savvy learning environment that can be used anywhere, by anyone with a Java-enabled web browser.

One aspect of this goal worth noting a second time is the platform independence of Educational Fusion. This independence, which I consider one of its most important objectives, is also one of the greatest restrictions of the project. Many commercial institutions are eager to see Educational Fusion succeed, as it will hopefully increase the general efficiency of computer education throughout the world. However, they show their eagerness by thrusting platform dependent technology upon us, attempting to better Fusion by limiting it to only a percentage of the population, instead of allowing it universality as originally intended. Unfortunately, by retaining its virtue of ubiquitousness, it is missing out on technologies which could significantly improve user experience. However, the cost of these technologies is too great and we can only hope that the companies which offer them will develop platform independent versions, expanding the usefulness of their products rather than limiting the usefulness of ours, and bettering the future of education in the process. With the support of the commercial and academic societies behind it, as well as the development skills of the excellent team members who have worked on the project before me, Fusion cannot help but become a universal learning tool of the new millennium.

Appendix A

Server Installation Checklist

This is an updated version of the checklist, originally written by Aaron Boyd [Boy99]

To install the system:

1. Copy the base source tree, currrently located in the directory Fusion on the computer Fusion.lcs.mit.edu, into a directory on your server, called Fusion.

2. Create a mapping on the web server from http: //*hostname*/edufuse/ to Fusion\Documents on your server, where *hostname* is the name of your computer.

3. Create a mapping from http://*hostname*/*myclient* to Fusion\Root, where *myclient* is an arbitrary name for your fusion client.

4. Set the default page to view in /edufuse to be System.html using your web server administration tools.

5. Change the codebase in the applet tag in System.html so the line reads:
   ⟨`applet codebase=‘‘`*myclient*`’’ code=‘‘edufuse.cg.EFApplet’’ width=100% height=100%` ⟩

6. Change the RegistryPort parameter in System.html so the line reads:
   ⟨`PARAM NAME=‘‘RegistryPort’’ VALUE=‘‘`*regnum*`’’` ⟩
   where *regnum* is a five digit number you will use to identify the port on which to connect to the registry server.

7. Change the line in Fusion\Efregsvr.bat to read[1]:

   `c:\winnt\jview.exe /cp:p c:\Fusion\Root edufuse.streams.RegistryServer`

   regnum `c:\Fusion\Root`

8. Change the line in Fusion\EventServer.bat to read:

   c:\winnt\jview.exe /cp:p c:\Fusion\Root edufuse.streams.EventServer *evtnum*

   where *evtnum* is a five digit number you will use to identify the port used by the event server.

9. Change the paths and numbers in Fusion\Servlets\EF.config to be correct. Specifically, make sure the following lines are present and updated:

   `EFRoot=c:\Fusion\`;

   `{EFHome}=http://`*hostname*`/Fusion/`;

   `{RegistryName}=`*hostname*;

   `{RegistryPort}=`*regnum*;

   `{ServerName}=`*hostname*;

   `{ServerPort}=`*regnum+1*;

   `{EventPort}=`*evtnum*;

   Where *evtnum* and *regnum* are the same numbers as in steps 6 and 8.

10. Install the jdks (and the jsdk if before jdk1.2).

11. Make sure jvc.exe exists in the Fusion directory. If not, copy it from your jdks installation.

12. Run the two batch files, Efregsvr.bat and EventServer.bat.

[1]If you did not install fusion on your c drive, then substitute the letter of your drive in every path in which the letter c appears.

Appendix B

Module Creation Checklist

This appendix gives a list of steps necessary to create a new module and accompanying visualization panel. For further explanation of the module steps not explained in this thesis, see Aaron Boyd's document on module creation [Boy99].

1. Create the reference, base and student modules.

   (a) Start by logging into Educational Fusion and viewing the concept graph panel. Create a new module by right clicking on the concept graph and selecting "create module."

   (b) Enter the types of inputs and outputs and the name of the module in the new dialog and click "create."

   (c) Exit Fusion and go to the new directory, `Fusion\Root\projects\graphics\mod-ules\`*yourModuleName* where *yourModuleName* is the name of your new module.

   (d) Change the third line in the file *yourModuleName*_info.txt to read:
   `INSPECTION_PANEL=projects.graphics.panels.`*yourPanelPackage*`.`*yourModuleName*`Panel`;
   where *yourPanelPackage* is the name you will use for your package and *yourModuleName* is the same as above. See line 4 of `ref_VectorSort_info.txt` for an example [Gla00].

(e) Duplicate the file `ref_`*yourModuleName*`.java` and name the new copy `Base`*yourModuleName*`.java`. Change the class name inside to be `Base`*yourModuleName*. See `BaseVectorSort.java` for an example of a completed base module [Gla00].

(f) Set up the input and output ports in the constructor of `Base`*yourModuleName*, as shown on line 13 of `BaseVectorSort.java` [Gla00].

(g) Handle the receiving of input by overriding `input()`, as shown on line 63 of `BaseVectorSort.java` [Gla00].

(h) Handle mouse events and drawing, by filling in `handleEvent() and paint()`, as shown on lines 47 and 30 of `BaseVectorSort.java` [Gla00].

(i) Create a function for the reference algorithm called `do`*myAlgorithm(stuff)* where *myAlgorithm* is the name of the algorithm the module teaches, and *stuff* is any parameters the algorithm needs to take. Declare this method abstract and leave the body empty. (See line 61 of `BaseVectorSort.java` [Gla00].)

(j) Open the file `ref_`*yourModuleName*`.java` and make `ref_`*yourModuleName* extend `Base`*yourModuleName* instead of `BaseModule`.

(k) Delete all methods and functions in `ref_`*yourModuleName*`.java`.

(l) Override the function `do`*myAlgorithm(stuff)* which you created in `Base`*yourModuleName* and actually implement the algorithm. See `ref_VectorSort.java` for an example reference implementation [Gla00].

(m) Open the file `user_`*yourModuleName*`.tmpl` and make `user_`*yourModuleName* a subclass of `Base`*yourModuleName*. Make sure to import the package which contains `Base`*yourModuleName*.

(n) Again, override the function `do`*myAlgorithm(stuff)* which you created in `Base`*yourModuleName*. Implement as much of the algorithm as you want students to see. Give ample comments explaining the animated data structure in use and how to complete the rest of the algorithm. See `user_VectorSort.tmpl` for a sample user template [Gla00].

2. Create the witness detector.

   (a) Create a file in the same directory called `Proof`*yourModuleName*`.java`. Inside it, implement a class called `Proof`*yourModuleName* which is a subclass of `BaseWitProof`. Make sure to import the package `edufuse.cg`.

   (b) Implement `FindWitness()` and `FindWitnessList()` as explained in Chapter 3. See lines 15 and 73 in `ProofVectorSort.java` or in Appendix C for sample methods [Gla00].

3. Create the visualization panel.

   (a) Create a subclass of `DefaultTeacher`, put it in the package `projects.-graphics.panels.`*yourPanelPackage* and name it *yourModuleName*`Panel` where *yourPanelPackage* and *yourModuleName* are the same as above.

   (b) Override `setup()` to setup the user interface, as specified in Chapter 4 and demonstrated on line 21 of `VecSortPanel.java` [Gla00].

   (c) Override `AlgorithmReponse()` to call your module, as specified in Chapter 4. Remap the ports in `FusionActionNoQueue()` and `setUpXXXModule-Bean()` if necessary. For an example, see line 73 of `VecSortPanel.java` [Gla00].

   (d) Override `runAlgorithm()` to call your module, as specified in Chapter 4 and demonstrated on line 104 of `VecSortPanel.java` [Gla00]..

   (e) Subclass `AnimDataType` or `VizView`, if necessary, as specified in Chapter 4. Include the new classes in the same package as *yourModuleName*`Panel`.

   (f) Override any other methods as necessary, as specified in Chapter 4.

Appendix C

Sample Witness Detector

```
package projects.graphics.modules.VectorSort;

import edufuse.cg.*;
import edufuse.modbase.*;
import java.util.*;
import java.awt.*;
import java.awt.image.*;

public class ProofVectorSort extends BaseWitProof
{
    public ProofVectorSort () {
      //System.out.println("Made a ProofVectorSort");
    };

      //This routine checks for a witness of incorrectness in
      //a sorted list. To do this, it first makes sure the
      //list has all elements in order, and then checks
      //that elements appear in the sorted list if and only
      //if they were present in the list to be sorted.
      //input should be the list which was to be sorted
```

```
    //and output should hold the list supposedly sorted by
    //the algorithm. They should both be AnimatedVectors.
  public Object FindWitness(Object input,Object output) {
    AnimatedVector vo=(AnimatedVector)output;
    Enumeration e;
    Integer i=new Integer(0),j;
    //first, test to see they're in order

    if(vo==null)
       //Oops, the output was null
      return new String(``Incorrect: Returns Null Vector!'');
    else {
      e=vo.elements();
      if(e.hasMoreElements())
        i=(Integer)e.nextElement();
      while(e.hasMoreElements()) {
        j=(Integer)e.nextElement();
        //Are these elements out of order?
        if(i.intValue()>j.intValue())
          return new String(``Incorrect, witness'' +
            `` to failure:\n'' +i.toString() +
            `` should not\ncome before ''
            +j.toString()+``.\n'');
        else
          i=j;
      }

    //now, test to make sure right elements are present
    AnimatedVector vi=(AnimatedVector)input;
    AnimatedVector tvi=(AnimatedVector)vi.clone();
```

```
AnimatedVector tvo=(AnimatedVector)vo.clone();

if(vi==null)
    return new String("Error!  Null Input\nsent to proof.");

int index;
e=vo.elements();
//Loop through the elements of the output...
while(e.hasMoreElements()) {
    i=(Integer)e.nextElement();
    //Was this element in the input?
    index=tvi.indexOf(i);
    if(index==-1)
        //Nope.
        return new String(``Incorrect, Witness to '' +
            ``failure:\n'' + i.toString()+
            `` appears in output\nmore times than in input.'');
    else
     //Remove it from input, so we don't allow multiple
     //copies in the output just because it appears
     //in the input once
        tvi._removeElementAt(index);
}

//Now, we makes sure all the input is present in
//the output, using a similar procedure to above.
//First, loop through elements of the input
e=vi.elements();
while(e.hasMoreElements()) {
    //Check if the element is in the output...
```

```
            i=(Integer)e.nextElement();
            index=tvo.indexOf(i);
            if(index==-1)
                //Not present.
                return new String("Incorrect, Witness to " + "
                    failure:"n" + i.toString()+
                    " appears in input\nmore times than in output.");
            else
                //Yes, present, so remove from output,
                //For same reason as above.
                tvo._removeElementAt(index);
        }

        return new String("Correct output\nfor this input!");

    }

}


//return a vector (sorted in ascending oreder)
//with ids of each incorrect object
//in the output
    //i should hold the list that was input into the sort
    //and o should hold the list returned.
public Vector FindWitnessList(Object i, Object o) {
    Vector  badVec=new Vector();
    AnimatedVector vo=(AnimatedVector)o;
    int k;
```

```
        //first, test to see they're in order

        if(vo!=null) {
            for(k=1;k<vo.size();k++) {
                if(((Integer)vo.elementAt(k-1)).intValue()
                   >((Integer)vo.elementAt(k)).intValue()) {
                    //vo[k-1] is greater than vo[k],
                    //let's say they're both incorrect?
                    badVec.addElement(new Integer(k-1));
                    badVec.addElement(new Integer(k));
                }
            }
        }
        //for now, let's ignore missing
        //or extra stuff in the output
        return badVec;
    }

}
```

Appendix D

Sample Animated Data Type

```
package edufuse.modbase;

import edufuse.cg.*;
import java.util.*;
import java.awt.*;

/**************************************************
Sample Animated Data Type for Vector of integers
for Educational Fusion

Copyright, Joshua Glazer, May 2000
MIT LCS
**************************************************/


public class AnimatedVector extends AnimDatatype
{

    Vector mSelf;

```

```
Color       kStripeColor=java.awt.Color.green;
Color       kBadStripeColor=java.awt.Color.red;

long        FRAME_DELAY = 100;
int         BAR_SPACING=3;
int         GAP_BOTTOM=50;
int         GAP_TOP=10;
int         SWAP_STEPS = 10;


//These are the constructors...the init the imagebuffer and         //the graphics to which the data draws itself.
//You should almost always use the bottom one.
public AnimatedVector() {
    mSelf=new Vector();
}

public AnimatedVector(Vector someVector) {
    mSelf=someVector;
            mImage=null;
            mGraphics=null;
}

//This takes the VizView and VizDebug from the DefaultTeacher.
public AnimatedVector(Vector someVector, VizView vv, VizDebug vd) {
    mSelf=someVector;
    setViews(vv,vd);
}

/**************************************************
Routine involving the Vector nature of the Structure,
```

```
altered to support auto-animation / visualization
*****************************************************/

  //Swap elements i and j of the vector and display
  //the animation, if appropriate
  public void swap(int i, int j) {
    Object temp;
    Graphics g;

    //first, if there is a debug area, record the swap
    if(mVizDebug!=null) {
        mVizDebug.addCall("Swap elements "
        + String.valueOf(i) + " and "
        + String.valueOf(j));
    }

    //animate if possible
    if(mIsAnimating && mGraphics!=null && mImage!=null) {
      draw();
      g=mImage.getGraphics();

      g.setColor(kStripeColor);
      g.drawString("Swapping Element "
        + String.valueOf(i) + " and Element "
        + String.valueOf(j),40,mHeight-10);

      //This routine animates the actual swap.
      drawSwap(i,j);
      //Now clear the offscreen buffer for the next
      //operation
```

```
            g.setColor(java.awt.Color.black);
            g.fillRect(0,mHeight-GAP_BOTTOM,mWidth,GAP_BOTTOM);
            delay(FRAME_DELAY);
        }


        //now swap
        temp=elementAt(i);
        mSelf.setElementAt(elementAt(j),i);
        mSelf.setElementAt(temp,j);

        draw();
    }

    public int  getMaxElement() {
        int maxElement=1;
        int test;
        Integer i;
        Enumeration e;

        //finds the element of greatest magnitude which is at least 1

        e=mSelf.elements();
        while(e.hasMoreElements()) {
            i=(Integer)e.nextElement();
            test=i.intValue();
            if(test<0)
                test*=-1;
            if(test>maxElement)
                maxElement=test;
```

```
    }

    return maxElement;
}

//these are the standard vector commands we
//want usable, and how to animate them
public Object elementAt(int i) {
    return mSelf.elementAt(i);
}

public int size() {
    return mSelf.size();
}

public Enumeration elements() {
    return mSelf.elements();
}

public int indexOf(Object o) {
    return mSelf.indexOf(o);
}

public void _removeElementAt(int i) {
    mSelf.removeElementAt(i);
}

public String toString() {
    return mSelf.toString();
}
```

```
    public Object clone() {
        AnimatedVector temp=new AnimatedVector((Vector)mSelf.clone());
        temp.cloneAnimStuff(this);
        return temp;
    }

/****************************************************
Routine involving the Animated nature of the Structure
****************************************************/

    public void setSpeed(int speed) {
        System.out.println("Speed is "+String.valueOf(speed));
        FRAME_DELAY = 2000 - 20*speed;
    }

    public void drawSwap(int swap1, int swap2) {
        if(mImage==null || size()==0) return;

        int barWidth=mWidth/size();
        int x1,x2,bottom,w,h1,h2,h,hdiff,stepnum;
        float   heightRatio;
        Graphics    g;
        Enumeration e;
        Integer i;
        float   f,f2;
        Vector  badVec;
        Color   c1,c2;

        g=mImage.getGraphics();
```

```
f=mHeight−GAP_BOTTOM−GAP_TOP;
f2=getMaxElement();
heightRatio=f/f2;

if( mProof != null)
    badVec=mProof.FindWitnessList(mOriginalInput,this);
else
    badVec=new Vector();

//System.out.println("Proof= "+mProof);
//System.out.println("badVec= "+badVec.toString());

if(badVec.indexOf(new Integer(swap1))==−1)
    c1=kStripeColor;
else
    c1=kBadStripeColor;
if(badVec.indexOf(new Integer(swap2))==−1)
    c2=kStripeColor;
else
    c2=kBadStripeColor;

//System.out.println("heightRatio is "+String.valueOf(heightRatio));
//System.out.println("barWidth is "+String.valueOf(barWidth));
//now, get the x positions and heights of the two to swap
x1=barWidth*swap1+BAR_SPACING;
i=(Integer)elementAt(swap1);
f=(float)i.floatValue()*heightRatio;
h1=(int)f;
```

```
x2=barWidth*swap2+BAR_SPACING;
i=(Integer)elementAt(swap2);
f=(float)i.floatValue()*heightRatio;
h2=(int)f;

w=barWidth-BAR_SPACING*2;
bottom=mHeight-GAP_BOTTOM;


hdiff=h2-h1;


//draw a little connector showing the swap is happening...

g.setColor(java.awt.Color.blue);
g.fillRect(x1+w/2,bottom,5,10);
g.fillRect(x2+w/2,bottom,5,10);
if(x1<x2)
    g.fillRect(x1+w/2,bottom+10,x2-x1+5,5);
else
    g.fillRect(x2+w/2,bottom+10,x1-x2+5,5);
//swap the bars, slowly!
//it should take SWAP_STEPS steps to swap them
for(stepnum=1;stepnum<=SWAP_STEPS;stepnum++) {
    //erase the old blocks
    g.setColor(java.awt.Color.black);
    g.fillRect(x1,0,w,bottom);
    g.fillRect(x2,0,w,bottom);

//Draw the new ones
```

```
            g.setColor(c1);
            h=h1+(stepnum*hdiff)/SWAP_STEPS;
            g.fillRect(x1,bottom-h,w,h);

            g.setColor(c2);
            h=h2-(stepnum*hdiff)/SWAP_STEPS;
            g.fillRect(x2,bottom-h,w,h);

        //Copy directly to the screen
            mGraphics.drawImage(mImage,0,0,null);
            delay(FRAME_DELAY/SWAP_STEPS/3);
        }
    }

    //This routine draws the structure in its animatable form
    public void draw() {
        //for the stripe vector, we will draw a
        //strip of varying length.
        //the max element of the vector will have
        //the full height of the screen, etc.

        if(mImage==null || size()==0) return;

        int barWidth=mWidth/size();
        int x,bottom,w,h;
        float   heightRatio;
        Graphics    g;
        Integer i;
        float   f,f2;
        Vector  badVec;
```

```
int     k;

g=mImage.getGraphics();

//erase the old
g.setColor(java.awt.Color.black);
g.fillRect(0, 0, mWidth, mHeight);


f=mHeight-GAP_BOTTOM-GAP_TOP;
f2=getMaxElement();
heightRatio=f/f2;

// System.out.println("heightRatio is "+String.valueOf(heightRatio));

x=0;
w=barWidth-BAR_SPACING*2;
bottom=mHeight-GAP_BOTTOM;

//last thing, get the vector of incorrect elements
if( mProof != null)
    badVec=mProof.FindWitnessList(mOriginalInput,this);
else
    badVec=new Vector();

//loop through and draw bars!

for(k=0;k<size();k++) {
    i=(Integer)elementAt(k);
    f=(float)i.floatValue()*heightRatio;
```

```
            h=(int)f;

            //if the current index is in the vector of bad
            //then we draw the bar in the bad stripe color
            if ( badVec.indexOf(new Integer(k))==-1 )
                g.setColor(kStripeColor);
            else
                g.setColor(kBadStripeColor);

            g.fillRect(x+BAR_SPACING,bottom-h,w,h);
            x+=barWidth;
        }
            mGraphics.drawImage(mImage,0,0,null);
    }

}
```

Bibliography

[Ala99] Alamo, Jesus del. "Weblab." *http://www-mtl.mit.edu/~alamo/weblab/index.html*. Massachusetts Institute of Technology, 1999.

[Ber+82] Berlekamp, et al. "Wining Ways, Volume II." Academic Press,1982.

[Blu+95] Blum, Manuel and Kannan, Sampath. "Designing Programs that Check Their Work." JACM V.42 No. 1 pp. 269-291, Jan.1995.

[Boy97] Boyd, Nathan D. T. "A Platform for Distributed Learning and Teaching of Algorithmic Concepts." MIT Thesis. 1997.

[Boy99] Boyd, Aaron "Educational Fusion: A Distributed Visual Environment for Teaching Algorithms." MIT Thesis. 1999.

[Bhu99] Kulkarni, Bhuvana. "Educational Fusion." *http://edufuse.lcs.mit.edu*. Massachusetts Institute Of Technology, 1999.

[Bro91] Brown, Marc. "Zeus: a system for algorithm animation and multi-view editing" In IEEE workshop on Visual Languages, page 4-9, Oct 1991. Also appeared as SRC research report 75. *http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-075.html.*

[Coh94] Abbe Cohen. "Inessential Zephyr." *http://www.mit.edu:8001/afs/sipb/project/doc/izephyr/html/izephyr.html*. The Student Information Processing Board, 1994.

[Cho88] Chorover, SL. "Paradigms lost and regained: epistemological and methodological issues in the study of human systems." New Directions for Research in Creative and Innovative Management. Cambridge, MA: Ballinger, 201-245. 1988

[Gla00] Glazer, Joshua. "Sample Module Code" *http://fusion.lcs.mit.edu/papers/code/*. MIT Department of Electrical Engineering and Computer Science, 2000.

[Gol+99] Goldberg, Murray, et al. "Web Course Tools." *http://about.webct.com*. WebCT, 1999.

[LBS94] Lawrence, Andrea W., Albert N. Badne and John T. Stasko. "Empirically Evaluating the Use of Animations to Teach Algorithms." ftp://ftp.cc.gatech.edu/pub/gvu/tech-reports/94-07.ps.Z Technical Report GIT-GVU-94-07. Georgia Institute of Technology College of Computer Science. 1994.

[KST99] Kehoe, Colleen, John Stasko and Ashley Taylor. "Rethinking the Evaluation of Algorithm Animations as Learning Aids: An Observational Study." ftp://ftp.cc.gatech.edu/pub/gvu/tech-reports/1999/99-10.ps.Z Technical Report GIT-GVU-99-10. Georgia Institute of Technology College of Computer Science. 1999.

[Por98] Porter, Brandon W. "Educational Fusion: An Instructional, Web-Based Software Development Platform." MIT Thesis. 1998.

[Pri98] Pritchard, David E. "CyberTutor." *http://cybertutor.mit.edu*. Massachusetts Institute of Technology, 1999.

[Sta96] Stasko, John T. "Using Student-Built Algorithm Animations as Learning Aids." ftp://ftp.cc.gatech.edu/pub/gvu/tech-reports/96-19.ps.Z Graphics, Visualization and Usability Center, Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-GVU-96-19, August 1996

[Tel94] Teller, Seth. "NSF Career Development Plan." *http://graphics.lcs.mit.edu/~seth/proposals/cdp.ps*. MIT Department of Electrical Engineering and Computer Science, 1994.

[Tel+98] Seth Teller, Nathan Boyd, Brandon Porter, and Nick Tornow: Distributed Development and Teaching of Algorithmic Concepts, in Proc Siggraph '98 (Educational Track), July 1998, pp. 94-101, also available at *http://edufuse.lcs.mit.edu/fusion/papers/siggraph98/siggraph98.html*.

[Tel+99] Teller, Seth, et al. "Computer Science and Engineering Education: Infrastructure for Collective Exploratory Teaching and Learning" MIT Department of Electrical Engineering and Computer Science, 1999.

[Tor98] Tornow, Nicholas J. "A distributed Environment for Developing, Teaching and Learning Algorithmic Concepts." MIT Thesis. 1998.

[Wei+92] Weiss, Thomas F,. et al. "Software for Teaching Physiology and Biophysics." Journal of Science Education and Technology, Vol. 1, No. 4. Plenum Publishing Company, 259-274. 1992.