Educational Fusion: An Instructional, Web-based, Software Development Platform

by

Brandon W. Porter

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING
AND
MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 4, 1998

Author _____
Department of Electrical Engineering and Computer Science
February 4, 1998

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

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

Educational Fusion: An Instructional, Web-based, Software Development Platform

by

Brandon W. Porter

Submitted to the
Department of Electrical Engineering and Computer Science on

February 4, 1998

In partial fulfillment of the requirements for the degrees of
Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

Abstract

Educational Fusion (eFuse) is an instructional, collaborative, Web-based, platform for learning and teaching algorithmic concepts through implementation and verification. This platform is designed to be available to anyone with a Java-enabled Web browser, to encourage students and staff interaction and collaboration, to minimize the overhead required for a student to address the essence of the learning experience, and to allow students to implement, evaluate, and explore algorithmic concepts. Each algorithmic concept is represented as a module. Students implement modules in Java, while the system compares their implementation to a reference implementation. Modules can be interconnected in a dynamic data-flow architecture to create larger systems.

eFuse is best viewed from three perspectives: that of the students, the teaching staff, and the system developers. From the students' perspective, eFuse is a rich collaborative development environment that allows them to explore classic textbook algorithms and their interaction as components of a larger system. Students can work within the bounds of an algorithm, or they can extend existing modules within a dynamic data-flow system. The system allows visual comparisons of a student's implementation with an internal reference implementation, providing rapid feedback. From the teaching staff's perspective, eFuse is an integrated environment in which they can: select and organize dynamic course content that challenges students to solve problems, remotely and cooperatively assist students, manage student accounts, and efficiently evaluate student work From the system developers' perspective, eFuse is a client-server system which maintains user state, supports real-time communications, and deliberately abstracts the system complexity from the students and the teaching staff. At the same time, Educational Fusion is a run-time platform for creating dynamic data-flow applications from simple modules written in Java.

Thesis Supervisor: Seth Teller
Title: Assistant Professor of Computer Science and Engineering, MIT Electrical Engineering and Computer Science Department

Acknowledgements

First, I would like to gratefully acknowledge Professor Seth Teller's contributions to this work. His vision and guidance have inspired me to keep working at it, and pushed us to bring it to the students. His tremendous help in editing this document was invaluable. I also need to acknowledge Nate Boyd and Nick Tornow's contributions to this system. The foundation Nate developed for this system was terrific, and Nick's review of my plans for the data-flow architecture was extremely helpful.

MIT will always conjure fond memories of the special people I have met here. To my friends from MIT: thank you for keeping me sane during the long hours; for helping me fully exploring some of my strange ideas and letting me help explore yours. Virtual Toy Bricks, Distributed Java Virtual Machines, and numerous plans to conquer the Web are among my best memories at MIT.

Last, but certainly not least, I want to thank my family: my brother for reminding me that there are worthy intellectual aspirations outside the realm of science and engineering; and my parents for always being there and always encouraging me that MIT was all well worth it. It was.

Contents

Chapter 1

Breaking Barriers in Traditional Educational Settings

Traditionally, instructional software development environments incur an up-front time penalty before a student can begin solving the problem. These time penalties include getting the assignment loaded into the environment, and learning the sometimes arcane build, execute, debug cycle for the environment. Often these traditional environments require special computing resources that may be scarce and do not take advantage of the collaborative possibilities of a networked environment. Today, Web-based educational tools offer rich visualization environments, collaborative tools, and the opportunity for distributed learning, yet lack the functionality to meaningfully challenge students to solve complicated problems.

We set out to design a system that improves upon traditional software development environments while extending Web-based educational systems to support true problem solving. The Educational Fusion system for creative, collaborative pedagogy, or eFuse, is the result. eFuse combines a rich, extensible set of course material and the collaborative capabilities of a networked environment with a user-customizable, persistent, location-independent instructional algorithm development environment. This environment allows students to begin solving algorithmic problems quickly and easily build larger solutions based

on these algorithmic components while collaborating with classmates and interacting with teaching staff.

This chapter provides an overview of the system, describes our motivation, and lists the specific objectives we set out to achieve.

1.1 Overview

eFuse is a Web-based environment for collaborative pedagogy that excels as an instrument for teaching algorithmic concepts. The teaching staff maps concepts from the course into algorithmic *modules*. Students are challenged to solve the implicit problem presented by the algorithm. For instance, one problem might be how to efficiently select the discrete pixels that comprise a line between two endpoints on a two-dimensional grid. Students or teaching staff members can connect algorithmic modules to create larger systems by linking modules in a data-flow architecture.

1.1.1 The Student Perspective

The eFuse system incorporates a number of concepts designed to help students learn the material with less overhead and more engaged learning. The course material is graphically represented by a collection of modules which can be interconnected or organized hierarchically in the overall *concept graph*, helping students to "see the big picture". Students "learn by doing", either by being asked to interactively solve the problem visually by hand, or by implementing the algorithm in code. Implementing algorithms involves traditional textbook rigor, but the ability to combine modules, extend them, and create new ones allows for creative expression. Collaboration is encouraged and facilitated. Each student and staff member using eFuse is represented with a graphical icon, or *avatar*, displayed in the concept graph space. Archived text-based instant messaging facilitates discussion and retrieval of discussion history.

1.1.2 The Teaching Staff Perspective

eFuse includes features useful to the teaching staff. These include simple Web-based student account creation capability, and semi-automated Web-based grading. Test modules can easily be developed which *link* to the student's module to validate the outputs. The system provides staff with the same messaging functionality that is available to students, but also allows the staff to open a stared view with a student. The teaching staff can open up any current or past instance of a student's work and view the content, allowing the staff to help a student understand the concepts in real-time. The teaching staff can also design input suites to automatically test a student's implementation on a full range of possible inputs.

1.1.3 The Developer Perspective

From a *system developer's* standpoint eFuse is technically sophisticated system that differs in many ways from traditional instructional software and development environments. First, eFuse encourages a template for instructional modules. This template includes four recommended modes for each module:

- *Manual mode* asks students to interactively implement a module.
- *Reference mode* provides a visualization of the algorithm, while hiding the implementation.
- *Implementation mode* allows a student to implement the algorithm and view its execution.
- *Difference mode* compares the trace of execution of the reference mode with that of the implementation mode and visually portrays any inconsistencies.

Second, eFuse automates many of the traditional processes in software development by abstracting the process of editing, compilation, revision control, and execution to a single action. In addition, eFuse maintains a complete state of the *user*'s session on the server, which can be retrieved by that student from any Java-enabled Web-browser.

Third, the system supports dynamically linking modules into a data-flow system. Each module is an individual *data-flow component* with defined inputs and outputs. Each module

inherently handles the multithreaded, asynchronous nature of the data-flow actions arriving from input ports and leaving from output ports.

1.1.4 Results

eFuse was first used in the Fall 1997 Introductory Computer Graphics course at MIT. Initial results were very encouraging. Administration was relatively simple, and students expressed a high rate of learning in relatively short time.

1.2 Motivation

This work was motivated by a number of observations about the methods of teaching today and the most effective means of learning. Professor Teller stated this clearly 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 amount of technical data and simulations come 'on-line,' and university courses begin to follow suit, we must fulfill the educators' roles of selecting, organizing, and presenting this material to the student. [Tel94]

In developing eFuse we noted the following specific changes in the instructional environment: more powerful computers and high-bandwidth networks are increasingly available to students often in their own rooms, remote education is increasing, and student/staff ratios are decreasing. We also noted the following limitations of existing systems: distributed content generally offers only limited passive interaction; current software learning environments are often highly customized to a particular environment, and difficult to translate; and current software development platforms require high initialization overhead for students.

An increase in the power of standard personal computers and the availability of high-bandwidth networking in the classroom, faculty offices, and student rooms reduces the need

for special-purpose hardware. For instance, MIT's introductory computer science class (6.001) has used HP 9000 terminals for assignments in Scheme. When originally deployed, these were the some of the fastest workstations available. Today many students have computers in their dorm rooms which are able to perform at the same level or better than these machines. Students and faculty also have access to high-bandwidth networks. These networks offer the opportunity for decentralized, location-independent access to courseware, course material, and user-specific data. At the same time, a single special-purpose machine can be used as a server for computationally expensive tasks such as compilation.

MIT has led the way in using this technology for location independent access to data with the development of Project Athena [Sal88] and the use of the Andrew File System (AFS) [Mor86] for the computers on campus. The Java runtime environment [Gos+95] allows anyone with a Web browser to access and run applications which are transparently distributed from a server. New revisions can be centrally posted to the server and are automatically received by the students the next time they request the application. At present, many students are not able to access this data effectively with personal computers which do not run the UNIX operating system. Java-enabled Web browsers are, however, available on almost all platforms.

Writing an interactive visualization which allows students to interact with a concept is difficult. Extending that visualization to allow greater interactivity, or even challenge the student to implement or extend the concept requires much more effort. Often a completely new visualization must be written for each concept. To the best of our knowledge, no standard template for creating Web-based visualizations existed before eFuse.

Finally, challenging a student to implement a concept often requires that the student learn the development environment. Determining how to compile, execute, and submit the result for grading often requires "debugging the environment". Students express a great deal of

frustration at this overhead, which usually relates little to the concept at hand. Thus assignments are generally large so that the percentage of "mindless grunge" is low compared to the percentage of "challenging work".

These observations motivated our design for eFuse.

1.3 Objectives

In designing eFuse we set out to create the ideal platform for collaborative interactive pedagogy. We were not explicitly motivated to design a system for distance learning, as many efforts recently have been. Distributed learning is only one of many features that we felt would improve the learning process. In this section, we will describe nine objectives that have guided the development of the system.

1.3.1 A Programming Laboratory with Intuitive Visualization of Course Material

Intuitive visualizations help students better understand material by providing a visual representation to complement text- or lecture-style descriptions. Creating a system that easily permits and encourages developers to create these visualizations for each module was a primary objective. The overall idea is that each concept is represented by a module which contains a visualization of that concept.

The environment should feel like a programming laboratory. All the tools to complete the assignment should be readily available and intuitive to the students. The components that the students are examining and implementing should be readily apparent. eFuse accomplishes this by creating a visual representation of the components the students are implementing and, in most cases, seamlessly providing all the tools necessary to build, execute, and evaluate that component.

1.3.2 Location-independent Access

Especially in today's residential educational communities, students often split their time between classes, research pursuits, and residential living. Increasingly, all of these environments are networked. Students and teaching staff should be able to access the resources of eFuse from anywhere. As distance learning becomes more common, location independent access to course material and assignments will permit greater flexibility. Students will be able to travel abroad for part of the semester and still keep up with assignments. Professors with busy travel schedules will still be able to help students and resolve issues in the class remotely. Guest experts from other educational institutions or industry will able to access the system and provide feedback.

1.3.3 Transparent Compilation, Execution, Visualization, and Validation

The objective of most courses is to teach the concepts of the course, not the arcana of a development environment. eFuse was designed with the explicit objective to make compilation, execution, visualization, and validation of a student's implementation simple. A single click of a button is all that is needed to compile, execute, and visually validate an algorithm implementation.

1.3.4 Effective Collaboration

Collaboration occurs throughout society. With increasing lecture sizes, but a continued emphasis on individual performance, developing the skills to learn effectively from others is difficult. Collaborating in a moderated forum helps students to resolve confusion while still learning the material. Fellow students have a unique perspective on the material at hand and are often able to quickly help peers resolve confusion. MIT's Zephyr system [Coh94] allows for collaborative messaging and is effectively used by students in a diversity of courses to help each other. To enable effective collaboration, eFuse provides a graphical representation

of all the students and staff currently using the system and allows text-based real-time communication along with a shared virtual white-board.

1.3.5 Easy Access to Student Work

Because development environments are relatively complicated to set up, quickly viewing, testing, and evaluating a student's implementation is generally difficult with current systems. The best solution is often for a member of the teaching staff to look over a student's shoulder, suggest possible changes and wait as the student implements and tests those changes. This is horribly inefficient, often leading to long queues of students requesting help. In a distributed context, this becomes even more difficult. Loading in the current instance of a student's implementation requires transferring a large number of files and configuring the environment to display those changes, presuming the teaching staff has the same resources that the student has. eFuse allows teaching staff to very easily load, view, change, debug, and comment on a student's implementation from any Java-enabled Web browser.

1.3.6 Students Both Work within the Bounds of the Course and Creatively Explore

Students learn best when they are actively engaged in the material. The advantage of asking students to implement algorithms is that there is a close and relatively fine-grain mapping between algorithms and concepts in a course. At the same time, if students are merely asked to implement what can be found directly in the book, the task will quickly become rote. eFuse attempts to balance the implementation of algorithms that helps students gain solid understanding of the material, with the opportunity for creative expression. Students can take their modules and link them into a larger system. Modules can be extended or added to the system easily and dynamically. This allows the course staff to challenge students who find the algorithm straight-forward to creatively extend the assignment.

1.3.7 Centralized Administration of Mutable Persistent System Resources

Student lists, assignment notes, and assignment implementations are examples of mutable system resources. Email and the Web have made it much easier for the teaching staff to centrally administer and update student lists and assignment notes. Assignment implementations are much more difficult to change. Often, due to unforeseen bugs in an assignment, the teaching staff must distribute new files to the students, who must incorporate those changed files into the system. Again, eFuse is designed to make it easy to change those elements of the assignment that the student need not modify, while combining these with simple administration of assignment notes.

1.3.8 Instructors Can Select the Appropriate Course Material

We noted in Section 1.2 that, especially as access to information becomes easier, the educators' role in selecting, organizing, and presenting material becomes even more critical. For instance, the instructor of an introductory computer graphics course may decide to emphasize techniques for geometry culling over how lighting and shading occurs. At the same time, the instructor would like to have a completely instantiated rendering pipeline so that students can see how all the components fit together. With eFuse, the instructor can use an entire pipeline, perhaps one already implemented by a previous instructor, and select those algorithms that students should implement as course assignments. The other algorithms could be left as exercises to the students, or could remain as purely functional reference components in the system.

1.3.9 Platform Independent on Both the Server- and Client-side

Students and teaching staff should not need to access special or particular computing resources to run either the client or the server system. While implementing eFuse, we chose technologies that were available on many platforms. On the client side, Java and HTML are

uniquely suited to this task. Any platform which has a Java-enabled Web browser can run the *eFuse client.*

Similarly on the server side, eFuse works with many Web servers and all added capabilities have been coded either in Perl or Java. This allows educators to run the system on whatever hardware is available. Data files are generally stored in ASCII format and thus will work regardless of machine byte ordering. For that data which is best stored in binary format, we created our own standardized byte serialization libraries[Boy97].

1.4 Outline

The remaining chapters of this thesis will first focus on related work in the field. Next, we will discuss the system from three perspectives. The first is that of the students, where we will give an example of how the system would be used in an introductory computer graphics course. The second perspective is that of the teaching staff, where we will discuss administration of the system. The third perspective is that of the system developers, where we will discuss the implementation details of the system components, paying particular attention to the architecture of the data-flow environment. We will then discuss the results from our first experience using the system in a larger course. Finally, we will discuss future work and our conclusions.

Many of the eFuse system functions are the designed to be the same for students, teaching staff, and developers. Throughout this document, we generically refer to anyone working in eFuse as a "user". For those cases where the system behaves in a manner specific to the type of user, we explicitly state whether the user is a student, teaching staff member, or system developer.

Chapter 2

Related Work

eFuse is a Web-based educational tool, an instructional software development environment, an algorithm visualization environment, and an architecture for authoring data-flow systems. Researchers have developed Web-based educational tools, such as WebCT, that are, at one level, related to eFuse [Gol+96]. Instructional software development environments are less common, though the Jscheme environment shares our vision by allowing students to implement assignments in a subset of the Scheme language from any Java-enabled Web browser [Hic97]. Algorithm visualization environments and toolkits, such as the Zeus system [Bro91] and Brown University's *Interactive Illustrations* [Dol97][Try97], have been an active area of research. Finally, data-flow systems such as SGI's ConMan [Hae88] and IBM's Data Explorer [IBM98], have been developed to simplify complex system design. eFuse is unique because it combines elements of all of these diverse systems into a single system for collaborative pedagogy.

2.1 Web-based Educational Tools

For an in-depth discussion of related tools, please refer to Chapter 2 of Nathan Boyd's thesis entitled *A Platform for Distributed Learning and Algorithmic Concepts* [Boy97]. We will discuss the important features of these systems, and their limitations.

2.1.1 WebCT

WebCT is a system that allows teaching staff to easily design and administer course content on the Web [Gol+96]. WebCT provides extensive tools for both the teaching staff and students. These tools include chat spaces, course content that is easy to create and modify, and databases of multiple-choice questions for students which can be automatically evaluated. WebCT also has the capability to track students through the system. WebCT has gone further toward simplifying administration of course material than eFuse; however, the system does not include the ability for students to perform true problem-solving. The multiple-choice question format for student evaluation has limitations.

2.1.2 Hamlet

The Hamlet project seems to provide tutorial-style courseware for introductory computer science courses while researching new ways to deliver content [Zac+97]. This work is similar to eFuse in its focus on engineering and computer science topics. The Hamlet project, however, relies greatly on the external Hamlet viewer. For some content, the Hamlet client must be installed on the local computer so that tutorials can be installed and maintained on the student's local computing platform. One important advantage of this is that the Hamlet system can easily link to commercially available software packages such as Maple or Matlab for instructional work. eFuse takes a different approach by maintaining the current student's state information on the server. eFuse also does not require any special client software beyond a Java-enabled Web browser. eFuse, however, does not have the ability to easily interact with existing commercial software on the client-side due to the limitations of the Java security model, which does not allow Java applets to access to client-side resources such as files and custom applications. Files must be read from the Web server that served the applet.. Custom applications, such as compilers, must reside on the server and are executed by network calls from the client.

2.2 Instructional Software Development Environments

The area of instructional and collaborative software development environments is largely an open field. Software development environments used for instruction are often standard software development environments occasionally modified to support added features. We are not aware of software development environments which display information about which assignments students are solving in real time. No software development environment of which we are aware allows teaching staff to easily and remotely load, view, execute, debug, and comment on students' code. This is a major limitation which makes it especially difficult to help students understand implementation intricacies.

The Jscheme project [Hic97] has created a simple interface for executing programs written in a subset of the Scheme programming language. Jscheme code can be executed by typing code in a Java applet and pressing "Evaluate". This simple environment has the advantage that students can work on Scheme code from any location with a Java-enabled Web browser. Much of the Scheme functionality has been incorporated, and additional functionality for creating windows and drawing in them is available to students. The Jscheme applet is especially applicable to teaching introductory computer science courses in which the Scheme language is commonly used. However, Jscheme does little to create a collaborative, persistent environment, nor does it appear to include the administrative capabilities and system-building opportunities included in eFuse.

2.3 Algorithm Visualization

Algorithm visualization has been and continues to be an active area of research. Some of the earliest and most influential work originated with Marc Brown's work on the Brown ALgorithmic Simulator and Animator (BALSA) system[Bro+84] at Brown University and later the Zeus system at Digital's Systems Research Center [Bro91].

2.3.1 Brown Algorithmic Simulator and Animator

BALSA was developed in 1984 to animate algorithms interactively on special-purpose, networked, graphical workstations. In order to animate an algorithm, the algorithm designer augments an implementation of an algorithm with event signals which provide information to the animation routine, causing a display update. This work was especially compelling because students could step through an algorithm and watch the animation change as each line of an algorithm was executed, instead of merely playing back an pre-recorded animation. This work also illustrated that these animations were successful in helping students understand the material.

2.3.2 Zeus

Zeus continued the ideas set forth in the BALSA project, but extended them to take advantage of strong-typing and parallelism. Again, algorithms were augmented with event signals. Libraries were developed to allow for more advanced animations including color, sound, and later 3D graphics.

2.3.3 Interactive Illustrations in the Exploratory Project

Recently, Brown University has been working on the Exploratory project [BGG98]. One component of this project is to develop a toolkit for creating *Interactive Illustrations* [Try97] in Java which might, for example, augment the text and illustrations in an electronic book. None of these systems, however, attempts to automatically animate an algorithm created by a student; instead student are allowed only to view the results of a "reference" algorithm implemented and augmented with event signals by an expert.

2.3.4 Student-Generated Animations with SAMBA

At the University of Georgia, students have animated their own algorithms by adding simple annotations to their own algorithms and then piping them through the SAMBA [Sta96]

front-end to the POLKA [Sta92] algorithm animation system. Students expressed a high degree of learning and enjoyment in being able to animate their algorithm at work.

2.3.5 eFuse Improvements

eFuse incorporates two new ideas to the topic of algorithmic animation and visualization for instruction.

First, eFuse includes a *manual mode* in which students attempt to manually solve the implicit problem that the algorithm attempts to solve – for instance, how to sort objects from smallest to largest. This manual interaction helps students understand the problem that the algorithm solves through a quick hands-on, highly interactive experience.

Second, none of these systems considers the case of automatically animating the *difference* between a correct and incorrect implementation of the same algorithm. We believe the benefits of animating an algorithm have been proven. Allowing students to be able to easily visualize their own implementation is especially important, but should not require students to learn an annotation language. Fundamentally, algorithms reduce to primitive actions (i.e. "SetPixel" in Bresenham's algorithm, and "Swap" in a sort routine). In eFuse, the *module designer* abstracts those base calls into methods which the student must call. Those methods perform the algorithm operation, while also signaling the animation and storing a trace of the algorithm for difference comparisons.

The manual mode and difference mode are particularly unique to the eFuse system. Developing effective manual interactions with the algorithm, and visual comparison's between correct and incorrect implementations is difficult. The effort on the part of the module designer is justified because the difference comparison can greatly aid a students in understanding tricky aspects of an algorithm, such as boundary cases, while helping verify the results of the student's implementation.

2.4 Data-flow Systems

ConMan was one of the first systems to support the creation of dynamic data-flow systems graphically [Hae88]. Components could be dynamically interconnected by graphically linking the output of one component with the input of another component. This system allowed users to develop complex graphics applications easily from interactive components, but the paper describing the system does not describe how difficult it is to build these components. Data flow components have the natural advantage of being highly modular. The limitation is that global state information does not exist. Consequently, any information needed by more that one component must be either passed along the entire chain of components or fed into each and every module that requires it.

IBM's commercial Data Explorer also implements a visual data-flow system [IBM98]. Modules have been developed to make complex data analysis easier by graphically linking elements in a data flow. Modules are developed by creating a module description file which describes the inputs and outputs of the module. The user then creates an associated C function which accepts a pointer to an array of inputs and a pointer to an array of outputs. Data Explorer inputs and outputs are not statically typed. Instead, the modules must cast all inputs to an internally meaningful type. Modules are not allowed to maintain state information, as results are cached. Modules also can not return results as they are generated [IBM97]. Instead, there is a one-to-one mapping between a set of inputs and a set of outputs.

eFuse is specifically designed to make module creation extremely simple, while also providing the intuitive interface that a *data-flow application* builder allows. Modules are created by simply specifying the input types and output types in a dialog box. Explicit use of *design patterns* allows automatic code generation of eFuse modules. Module inputs and outputs are typed, allowing run-time type checking before a link is instantiated without relying on "meta-data".

An eFuse module can send output to the next module throughout the course of its execution. For instance, the Bresenham line-draw algorithm can send the first pixel object as an output while the next is being calculated. This allows the system to support different levels of data granularity without *blocking*. eFuse relies on the Java threading model to keep modules running in parallel. Thus, if the Java threading implementation supports multiple processors, then modules can automatically take advantage of them.

2.5 Summary

All of these systems share properties with eFuse. In certain aspects, these systems offer more than eFuse, and in other ways eFuse is an improvement. The fundamental difference, however, is that eFuse combines the best of each of these. eFuse is a Web-based educational tool, an instructional software development platform, an algorithm visualization workspace, and a data-flow system combined to create a complete platform for collaborative pedagogy.

Chapter 3

The Student Perspective

This section introduces the system by describing how students might use eFuse in a Computer Graphics course. At the same time, it will introduce the reader to the components of eFuse, and illustrate how each maps into our overall vision. This section will largely avoid implementation details.

A screen-captured image of the student workbench is shown in Figure 1. The large open grid is the "concept graph" (A). This area contains the visual representation of the modules available in a system and the interconnections that exist between these modules (B). The concept graph also displays which aspects of the system individual students are currently exploring with small graphical avatars (C). The toolbar provides quick access to common functions such as the ability to load a concept graph, open a module, or create a new module (D). The status bar at the top provides information about the current state of the system (E). Below all of this is the communications center in which students can type messages or draw their ideas for others to see (F). The editor for individual modules appears as a floating window on top of the workbench (G).

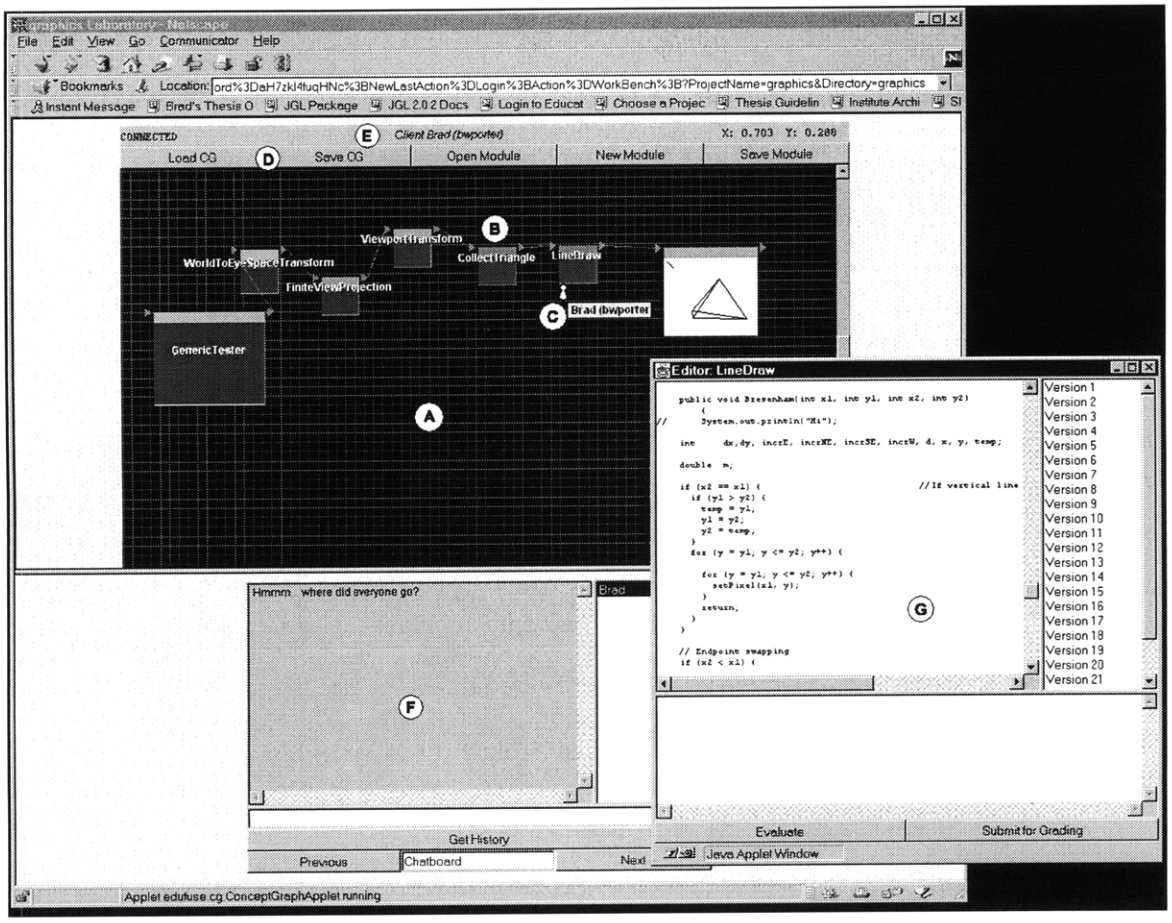


Figure 1: eFuse Student Workbench

First, we describe a canonical example, the classic polygon rasterization pipeline. We will use this example to show how students can get a sense for the "big picture" in a course. We will illustrate how the system encourages students to learn by doing, and how eFuse incorporates textbook rigor while allowing creative expression. Finally we will describe how students collaborate with fellow students, and seek guidance from teaching staff.

3.1 Computer Graphics

In order to best illustrate the eFuse system from the student perspective, we will use an introductory computer graphics course as our primary example, focusing the concepts of

three-dimensional graphics. Computer graphics is a field that is particularly rich in concepts that can be presented algorithmically and visually. However, we note that eFuse can be used as a vehicle for courses in algorithms, introductory programming, and other courses in which concepts can be expressed and understood algorithmically.

Fundamentally, computer graphics is concerned with efficiently translating a representation of a graphical structure into a corresponding visual representation of that structure suitable for display on an output device. This occurs at many levels. In two-dimensional graphics, a structure with a center and a radius representing a circle must be converted into a discrete set of pixels in order to be displayed on a device with a 2-dimensional grid of display elements. In three-dimensional graphics, a structure representing an entire scene must first be oriented relative to a virtual camera and mapped into a two-dimensional space, then rendered into a discrete set of pixels of display elements. Computer graphics is especially interesting because there are many methods to accomplish these goals. These methods each allow different trade-offs between computation time, memory usage, simplicity, generality, and image accuracy.

Students need to understand the fundamental concepts involved in generating three-dimensional imagery before they can begin to fully understand the advantages and disadvantages of different techniques, and before they can begin to formulate their own techniques. Learning these concepts is largely an exercise in rigorous textbook understanding. Once students have grasped these concepts, they should be encouraged to explore the trade-offs between different techniques and create their own techniques. eFuse accomplishes both.

3.2 Seeing the Big Picture

Students need to see the "big picture" in a course in order to best understand how the details fit. eFuse is uniquely designed for this task in courses where the big picture is the functional composition of a set of smaller components.

These components are all visible within the concept graph area of eFuse. The concept graph is a free-form grid which contains the current modules in a system or topic. Each module is visually represented as a box with triangles denoting the input and output ports. Concept graphs can be saved and restored at a later time. For instance, different concept graphs might represent different courses, or different systems built with the modules available in a course.

The classical polygon rasterization pipeline can be considered the "big picture" that students must understand in an introductory computer graphics course. The pipeline is a functional system comprised of smaller algorithmic building blocks.

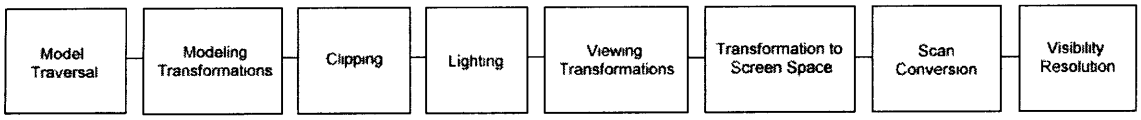


Figure 2: Classical Polygon Rasterization Pipeline (Naïve View)

Generally, the rasterization pipeline works as follows: a model is traversed to generate geometric primitives which are then transformed into the world-space of the scene. These primitives are clipped against the view volume, then lit based on the lighting description for the scene. Next, the geometric primitives are transformed from world-space to eye-space relative to the synthetic camera. The primitives are then projected into screen space, rasterized into device coordinates (pixels) and visibility conflicts are resolved. The final output is an image displayed on the physical device.

Different techniques, occasionally applied in a different order, may fulfill these steps. For instance, some or all visibility resolution could be done at an earlier stage in the pipeline depending upon the constraints of the model. Transformations to screen space are commonly either parallel or perspective projections. Different lighting models exist, some of which might be calculated after viewing transformations have taken place. The eFuse system is capable of representing and effecting all of these different techniques for rasterizing polygons.

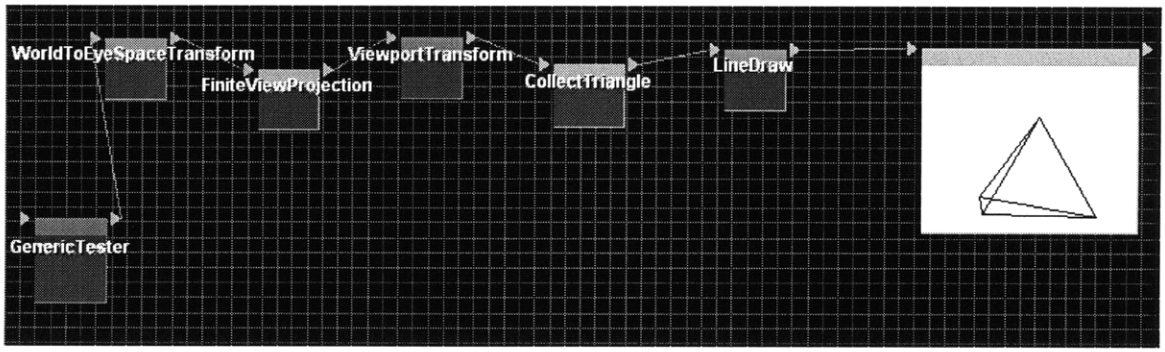


Figure 3: Simple Classical Polygon Rasterization Pipeline Implementation in eFuse

Figure 3 is a screen capture from the eFuse system which displays a fully instantiated graphics pipeline for rasterizing a wire-frame view of a three-dimensional model. This is perhaps the simplest version of a graphics pipeline containing a synthetic camera. As an instructional tool this pipeline can be implemented quickly by students based on a text-book understanding of the material. This pipeline also serves as the basis for further exploration of the pipeline, and can easily be extended with new modules created by students.

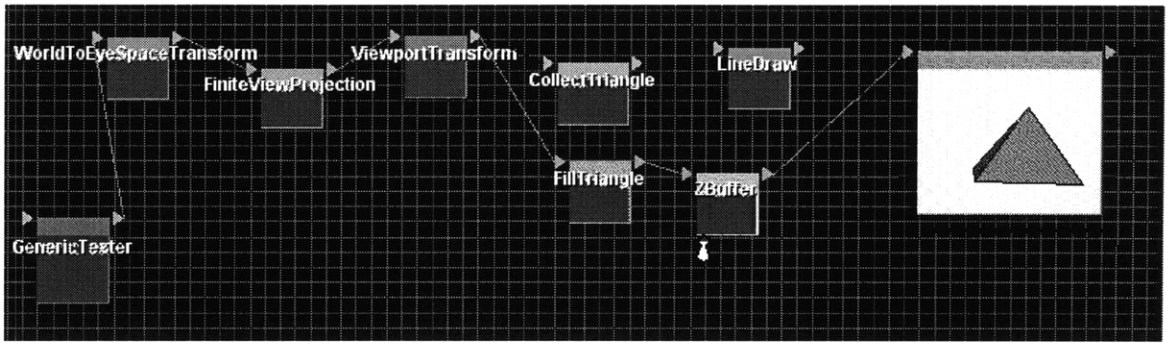


Figure 4: Extended Classical Polygon Rasterization Pipeline in eFuse

Figure 4 illustrates how adding two new modules can change the pipeline from the wire-frame render shown in Figure 3 to a filled polygon implementation. Students can see this difference by simply re-linking the modules.

3.3 Beauty in the Details: Learning by Doing

Clearly understanding the big picture is important. However, the interesting and challenging work is often in understanding the subtle details. Why is clipping a line against a rectangle non-trivial? What is the quickest way to test if a polygon is inside a given volume? What parameters are necessary to completely specify a synthetic camera? What are the differences between Gouraud and Phong shading? One of the best ways to learn and remember these details is to implement them.

Implementing components in eFuse is designed to be as easy and as instructive as possible. Students interact with individual modules in one of four modes: manual mode, implementation mode, reference mode, and difference mode. We will discuss each of these in turn.

3.3.1 Manual Mode

Manual mode allows a student to directly interact with a visualization of the algorithm and challenges the student to use simple mouse interaction – "manual interaction" -- to effect the operations that the algorithm would perform. For instance, the manual mode for Bubble

Sort might request that the student shift graphical bars representing data of various values in the same manner that Bubble Sort would shift them. For other algorithms, the student may attempt to solve the implicit problem of the algorithm through simple visual approximation, without attempting to perform the operations specifically as the algorithm would. Given that certain algorithms grow quickly, manual mode generally contains small inputs so that the student can quickly perform the operations by hand.

In the realm of graphics, the interactive mode of Bresenham's algorithm challenges the student to click on the pixels that Bresenham's algorithm would illuminate. An applet which implements the interactive mode of Bresenham's algorithm is shown in Figure 5.

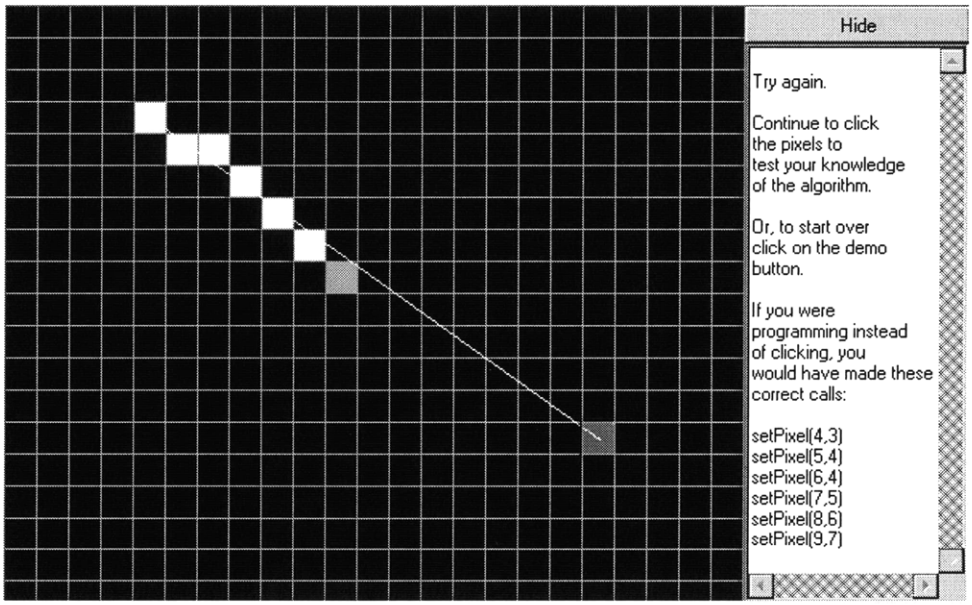


Figure 5: Bresenham's Algorithm in Interactive Mode

3.3.2 Reference Mode

Manual mode allows a student to student understand the basic algorithm behavior. Once that behavior is understood, a student can then enter reference mode and interact with the

algorithm by changing the inputs and viewing the new outputs. This mode is very similar to traditional algorithm animation and visualization systems. This mode also helps a student understand how the algorithm behaves in special boundary cases that the student may not have learned from the interactive mode. Figure 6 is a screen-capture of the Cohen-Sutherland line-clipping algorithm in reference mode. The entire gray line is the original student-specified line. The gray filled box is the clipping region. Finally, the white line and endpoints denote the new clipped vertices.

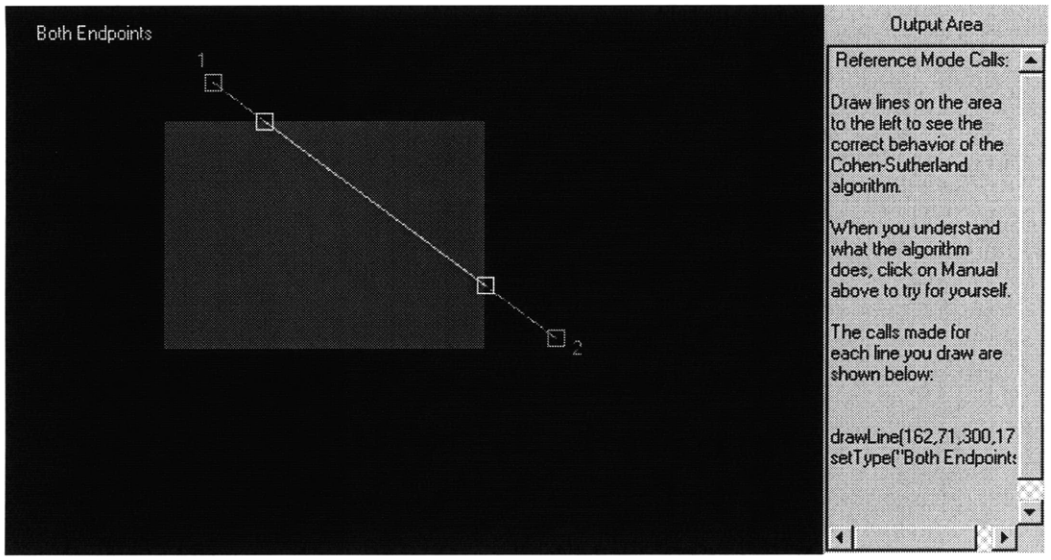


Figure 6: Cohen-Sutherland Line Clipping in Reference Mode

3.3.3 Implementation Mode

Research has shown that students who actively engage in implementing an algorithm understand and retain the concepts better than those who just view a visualization. Specifically, Georgia Institute of Technology published a report stating:

> Allowing students to create their own examples [of algorithm animations] in a laboratory session led to higher accuracy on the post-term examination of understanding of the algorithm as compared to students who viewed prepared examples or no laboratory examples. [Law+94]

In the implementation mode, the student is challenged to implement the algorithm in the Java language. In order to implement the algorithm, the student brings up the code edit window for the algorithm. The module designer is responsible for abstracting any necessary details about the interaction with the eFuse input/output port, the visual algorithm display, and event handling, allowing the student to focus on the algorithm implementation. In the edit window, the student sees an empty method declaration. Necessary information about input and output data types should be provided by the module designer to the student, along with information about the special methods available to the student. For instance, a module for Gupta-Sproull's anti-aliased line-draw algorithm might provide a `setPixelIntensity` routine which the student invokes to set the illumination intensity of a pixel.

One of the objectives of eFuse is to simplify the process of compilation, execution, visualization, and validation of an algorithm. A student implementing the Bresenham Algorithm begins by implementing at least part of the algorithm in the edit area. In order to test out the current implementation, the student clicks the "Evaluate" button. eFuse then compiles the implementation, stores the latest code as a new version, reports any compilation errors, and attempts to load and execute the compiled code. Figures 7 and 8 illustrate this process. Figure 7 is the empty implementation. No pixels are highlighted in the visualization area. Figure 8 illustrates the results of a student clicking "Evaluate". Clearly, the two endpoints of the line are now illuminated – the student's code is now running. A new revision is listed on the right of the edit area so that the student can return to a previous implementation if necessary.

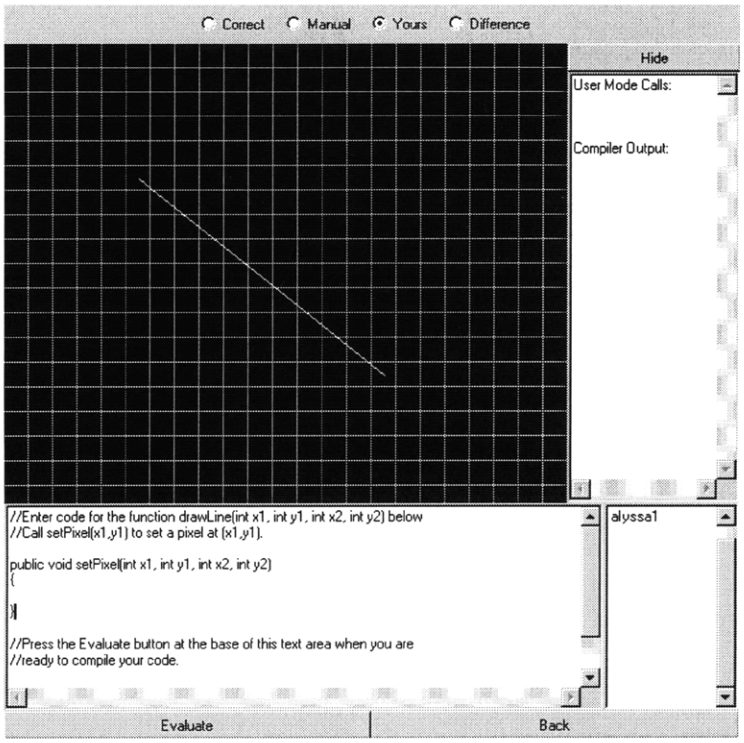


Figure 7: Initial Empty Implementation View

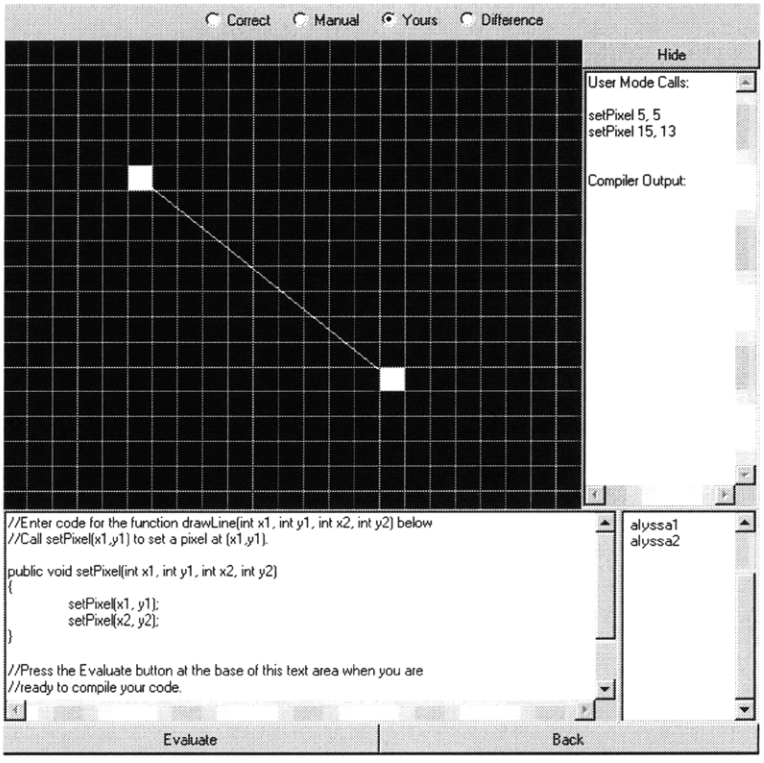


Figure 8: Implementation View After Evaluation

3.3.4 Difference Mode

The difference mode is a particularly useful and unique feature of eFuse. In difference mode, a student can visually compare his or her own implementation with the reference implementation. The module designer must implement a visual difference engine which uses a trace of the key execution events in the reference implementation and the trace of the student's implementation to display inconsistencies visually. Designing an effective difference mode can be a challenging task for the module designer. The effort spent by the teaching staff in implementing the difference mode of an algorithm proves worthwhile. The ability to visually inspect and debug differences in execution greatly improves the learning experience for students. At the same time, the difference mode allows the system to check the correctness of the implementation's mapping of inputs to outputs, aiding the teaching staff in evaluating the student's work.

Figure 9 illustrates the difference mode visualization of an incorrect implementation of the Bresenham algorithm. White pixels are those which the student's implementation correctly illuminated. Dark gray denotes pixels which the student's implementation did not illuminate, but should have. Finally, light gray pixels are those which the student's implementation incorrectly illuminated.

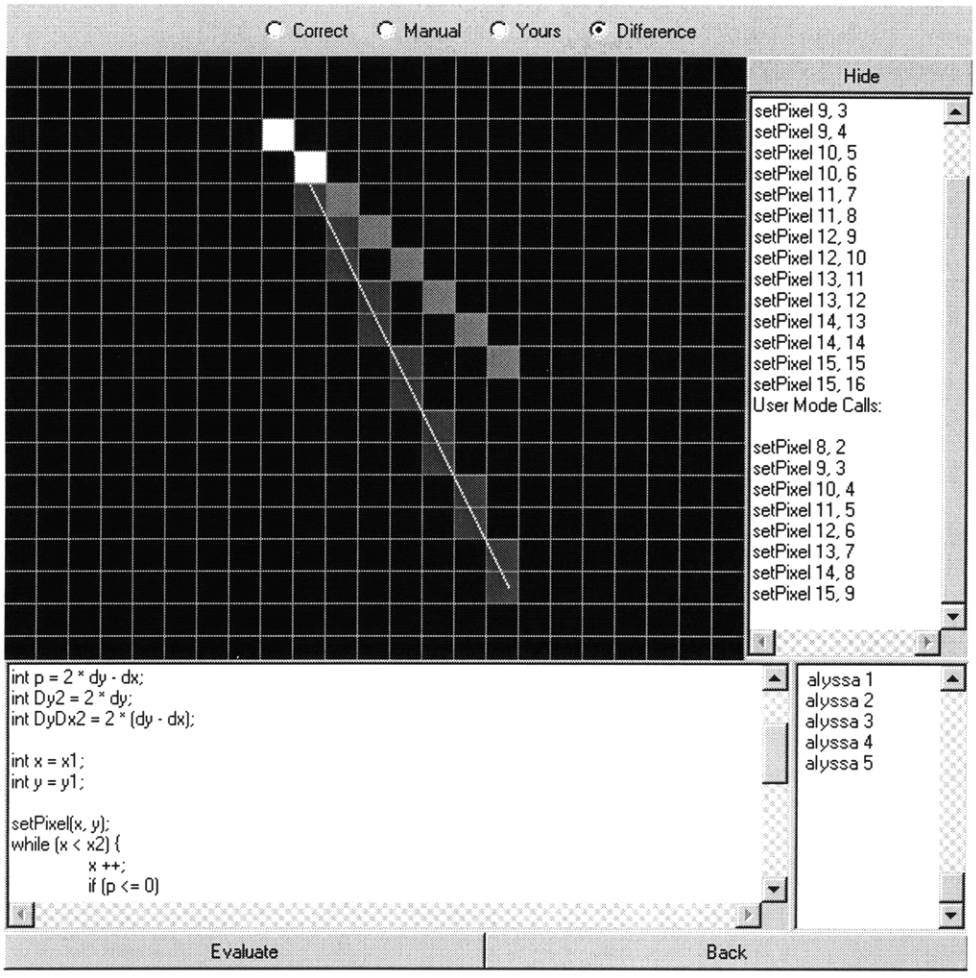


Figure 9: Difference Mode for Bresenham's Algorithm

3.4 Collaboration Between Students

First we looked at how eFuse helps students to see the "big picture" by viewing the components and interconnections between them within the concept graph. Next we looked at how a student can dig into the details of the algorithm's behavior on different inputs and outputs, and how a student can then implement the algorithm. These features help students to understand the material individually. Often, however, just interacting with an algorithm and seeing how it fits into the big picture of the material is not enough to answer all of a student's questions or resolve subtleties. In these cases, collaboration between students is invaluable.

Identifying students who can are working at a similar pace is the first step in effective collaboration. eFuse automatically places small graphical avatars next to the module that each student is exploring (Figure 10). A student can click on an avatar to find out the username of a fellow student who is working on the same module. A student can then send a text message to that fellow student working on the same module. The student can also send a message to all students on that module using the text messaging area. The fact that all messages are logged helps students look back at areas where others had questions, but also discourages students from sending full module implementations to each other in order to finish an individual assignment without doing the work.

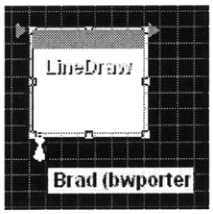


Figure 10: Graphical Avatar Representing a Student Working on the LineDraw Module

The collaboration space allows students to send messages to a single student, a select group of students, or everyone using the eFuse system. Students can also draw on a shared whiteboard. Readers interested in a detailed explanation of the functions and implementation of the collaboration space should see [Boy97] .

3.5 Interactions Between Students and Teaching Staff

Both the students' and the teaching staff's time resources are intrinsically valuable. Currently, in the university environment, teaching staff members give assistance in one of two ways. First, and most commonly, a student requiring assistance must seek out a member of the teaching staff at special times when that staff member is available. Often the teaching staff does not have the time or resources available to pull up the student's current implementation and help resolve questions about the finer details. As an example, it is

difficult for teaching staff to help a student resolve subtle problems in a student's implementation of a ray tracer. Once the large issues have been solved, the student is often left hoping the details are correct because staff feedback is nearly impossible. The second way the teaching staff can help is by interacting with students in an electronic laboratory by looking over the student's shoulder and working with the student to resolve subtleties. This is an expensive proposition because it requires centralized computing resources and a high degree of staff availability.

eFuse addresses these issues by creating a virtual, distributed laboratory. The teaching staff can easily load a student's current module implementation, test it, recommend changes, and view the student's next attempt. The teaching staff can also load the student's current complete concept graph to see the entire system. The staff member may also load a number of students' implementations simultaneously to look for common problems. The students may be across campus, or across the ocean. Currently student-staff interaction is limited to text-based communication, but as increasing network bandwidth becomes available, audio and video may be available. The virtual laboratory allows the staff ultimate flexibility in when they can be available to help students, reducing the need to negotiate specific appointments with students.

Chapter 4

Teaching Staff Perspective

eFuse is not only designed to be beneficial to students and architecturally interesting to developers, but also to maximize the teaching staff's time, availability, and effectiveness. The role of the teaching staff is to select the material that students need to learn, help students learn that material, and evaluate how well the students learned the material. eFuse facilitates each of these objectives.

eFuse is designed to be an ideal platform for creating, distributing, and tracking student course assignments. As described in the chapter on future work, the complete feature set for teaching staff is an active area of research and development. The system currently provides support for administrative facilities such as maintaining student lists, updating assignment descriptions, and logging messages. Instructors are encouraged to proactively select or design modules which students are asked to implement, and to arrange these modules into systems where appropriate, or visual hierarchies representing the "big picture" for the course. eFuse also attempts to simplify the process of evaluating student implementations, often making it feasible to look at a student's implementation in more detail than is usually possible with traditional systems.

4.1 Administering the Course

The teaching staff has a high degree of control over the system Most of these functions are controlled through a simple Web interface. The staff is able to perform such tasks as maintaining user lists, and controlling which eFuse users have administrative access. The system also provides support for the teaching staff to maintain a "Message of the Day" (MOTD) posting to give students the latest information pertinent to the course. Descriptions of the assignments can be posted directly within the system. In order to track possible bugs in assignments, the staff can review the history of public messages posted by students to look for common problems. These features greatly aid in administering the course.

4.1.1 Helping Students

Beyond general administration, eFuse gives the teaching staff the ability to better help students. The staff can also use the same communications facilities to provide individual feedback to students. eFuse also allows the teaching staff to maintain flexible hours of access that better meet student demand, rather than just during "office hours". The teaching staff is able to see avatars representing all the students who are online at one time, while also monitoring questions. Because this can be done with just one window on a Java-enabled Web browser, the staff can be working on other tasks, while monitoring the class in the background. This, combined with the ability to rapidly load a student's implementation and view the implementation details and results, enables "targeted just-in-time instruction". For instance, if a student asks a question about an algorithm, the staff member can quickly pull up the student's code, check for the error, then quickly check if other students appear to have the same error. The staff member can then specifically select those students and provide real-time instruction about areas of confusion in the algorithm. This ability to identify problem areas and address them specifically to those students who are having problems in a matter of moments is nearly impossible with current systems.

4.1.2 Selecting, Designing, and Organizing the Material

One role of today's educator is to select the most appropriate material to instruct students in a field of endeavor. In designing eFuse, we sought ways to encourage reuse of course material, while provide an educator with the ability to select that material from pre-existing modules or create new material when the pre-existing material is inadequate. Modules attempt to break concepts into small, self-contained, highly modular, algorithmic components. These components can be and should be reused. For instance, if a professor at one university has created a compelling reference implementation of view-frustum culling, a professor at another university should be able to use that implementation in his or her course. Currently, assignments generally cater specifically to the environment at a particular university. We hope, as eFuse grows in popularity, universities will share a common platform for teaching algorithmic concepts. In such a situation, a market may arise for modules, creating an environment in which professors can easily select from a large body of modules when creating a course.

Once the modules have been selected or designed, the teaching staff is responsible for arranging them in a way that gives students a high-level view of the material. In courses such as computer graphics, a complete running implementation of the rasterization is just one possible organization. Ultimately, we feel students will gain the most if they can take the modules they have implemented, new modules they create, and reference modules borrowed from other sites to build a system.

4.1.3 Evaluating How Well Students Are Learning the Material

The ability to load a student's work, view it, and execute it enables the teaching staff to quickly evaluate an implementation, focusing on details that the student may have missed. The ability to quickly load a student's implementation, view the code, and examine it in "difference mode" is largely unique to eFuse and allows the teaching staff to examine each student's submitted work in much more detail. The teaching staff can also design tests and

specialized visualizations of the algorithm as modules that connect to the input and output ports of the student's module. Consequently, a programmatic batch of tests can be hooked to the inputs of an algorithm, and a module attached to the output port can examine the results and flag inconsistencies. A web-based grading form is designed by the staff for each assignment which writes the grades into each student's directory for referral in the event that grading questions arise. Each of these features improves the quality and equity of the grading and feedback process.

Chapter 5

The Developer's Perspective: System Implementation

This chapter describes the eFuse implementation. In particular, we focus on the architecture of the data-flow subsystem while describing the intrinsic functions of a module. For an in-depth discussion of the implementation of the visual aspects of the concept graph, the administrative interface, and the collaboration components, please refer to *A Platform for Distributed Learning and Algorithmic Concepts* [Boy97].

Initially, the Educational Fusion project was focused on providing a framework for instruction on individual modules. Pedagogically, realizing a system that helped students learn the fundamental building blocks was very important. As the project progressed, it became apparent that while this was compelling and important, the system did not give students ample opportunity to explore and create. We had developed a system that could teach the fundamental building blocks in a field of study, but these building blocks could and should be used to build a larger system. Students should be encouraged to create, explore, and extend systems created from these basic building blocks. We have enabled this capability in eFuse by allowing modules to be combined into a dynamic data-flow system.

This chapter first explores the actions that users perform in the system and describes how the system processes those actions. Attention is paid to the details of loading the system and

creating, loading, linking, and executing modules. Next, this chapter describes in detail the implementation of the client-side data-flow architecture. Initially, we describe why a data-flow model was chosen, and how it is implemented in eFuse. Similar attention is paid to the process of transferring data between modules and maintaining the interconnections defined by the user. Finally, we present an example of the system in action by describing the implementation details of the classical polygon rasterization pipeline that we have instantiated with the system.

5.1 Overall System Architecture

This section describes the overall system and discusses the specific implementation details that allow the system to remain completely location independent.

5.1.1 Client-Server Architecture

eFuse is implemented as a client-server system. The client application is comprised of a HTML environment and set of advanced Java applets for developing modules and collaborating with peers. The complete client application is loaded into a Java-enabled Web browser from the server each time a user enters the system, which means users always receive the latest version of the eFuse client.

The server system is comprised of a traditional Web server, a registry server written in Java which tracks users in the system, and a messaging server, also written in Java, which handles text-based message delivery. The Web server is enhanced with a library of Common Gateway Interface (CGI) scripts which perform the essential system functions such as generating the HTML code for the client, and compiling users' modules.

5.1.2 Location Independence

eFuse was designed to support completely location-independent access to user data. All user-specific, course-specific, and system information is stored persistently on the server.

Users access their persistent information by supplying a username and password that uniquely identifies them.

A single directory acts as the root for all persistent data in eFuse. This root is specified as the applet "codebase" for the Java client. This allows the system to dynamically load any *Java class file* in any subdirectory of the *root directory*, provided the class's *package name* matches the directory. For instance, a graphics module stored in the subdirectory of the root named `/projects/graphics/modules` must be a member of the Java package named `projects.graphics.modules`.

Subdirectories from the root directory are created for each course and each user. Each course directory contains information about the assignments that students have been asked to complete, and the modules associated with those assignments. Each user's directory stores the user's profile, all the versions of the user's modules, and any persistent concept graphs that the user has created. The client either explicitly allows the user to save information in this directory on the server, or it is done implicitly without user intervention. For example, a student, teaching staff member, or developer can save a concept graph encapsulating the modules for a specific system by clicking on the "Save CG" button. The client sends a binary image of the state of the concept graph to the server along with the any information specific to the user. When the user requests the saved concept graph again, the server searches the user's directory for that file and returns the binary image of the stored state of the concept graph back to the client. Information such as the user's profile, which is created when the user creates an account is implicitly stored on the server. The user enters the pertinent data into an HTML form, and the *CGI script* writes that information into a user profile in the proper directory.

5.2 Implementation of User Actions

Users enter the system by loading the client application from the server into their Java-enabled Web browser. Users can then create, load, link, compile, and execute modules. The user-interface for these actions is designed to be straight-forward and elegant. The implementation of the underlying system intentionally abstracts the details from the user. In this section, we explore those details.

5.2.1 Entering eFuse: Loading the Client

In order to load the eFuse client and establish a connection to the server, the user directs a Web browser to load the eFuse home page. Upon entering the system, the server generates an HTML form which requests authentication from the user. When this is complete, the Web server generates a user-specific Web page that contains the Java client.

All eFuse client Web pages are generated on the fly based on templates. This is accomplished by the server using a CGI script which takes information from three sources in order to generate the appropriate custom Web page. The three sources are the eFuse system profile, the user profile, and the current state of the application. The eFuse profile stores information about the location of client components on the server and the properties of the system. For instance, the root directory for the Java client is specified in the eFuse profile, as is the location of the Java compiler. This allows the system maintainers to change the arrangement of files on the server and update eFuse by editing the system profile. The user profile stores information about the user such as whether the person is a student, teaching staff member, or system administrator, along with the user's email address and name. Information about the current state of the application includes information about the previous operation performed and the results of that operation. For instance, the server CGI script uses form data that the user enters in order to determine the next appropriate action.

5.2.2 Creating a Module

Creating a module is a relatively simple process for the user because, using common design patterns, the basic functionality of every module can be coded automatically. In order to create a module, the user enters the name of the module, the input types that the module can receive, and the output types that the module generates into a simple dialog box. Based on standard design patterns, a CGI script generates a Java source file representing the module, and compiles it into a class file. The eFuse client then requests that that class be loaded into the current application, and creates an instance of the new module. The new module generated by the script is fully functional.

The newly generated module contains a single method for each input port that the module supports. The module designer enters the code that specifies the actions that the module takes when new data arrives. The module designer can issue data on an output port simply requires making one method call in the module, `doFusionAction`. The generated module also contains standard Java `paint` and `handleEvent` methods which the designer can use to implement the module's graphical display and event handling routines. Appendix C exhibits the full source code of a module for Bresenham's algorithm created through this process.

5.2.3 Loading a Module

Once a module is created or selected for inclusion in the current concept graph, it must be loaded. In Java, the `Class.forName` method provides the ability to dynamically load a class into the running applet based given a string of characters representing its name. This class can be located on a server at an address relative to the applet code's *Universal Resource Locator (URL)* address. If a class of the same name has been loaded once already, a locally cached version is automatically loaded by the browser's *Java Virtual Machine.* Once loaded, eFuse creates a new instance of the object. The following two lines of code perform this operation in Java:

```
Class c = Class.forName("projects.graphics.modules.SimpleTestModule");
SimpleTestModule stm = (SimpleTestModule)c.newInstance();
```

A graphical representation of the module component is then added to the concept graph grid.

5.2.4 Linking Modules

A user graphically links modules by dragging a link between the output of one module and the input of another module in the concept graph. In eFuse, the data-flow component which generates data is considered the *source*. The recipient data-flow component is termed the *listener*. Generally, these data-flow components act as both listeners waiting for inputs and as sources generating outputs. When a link is completed by the user, the *source module* in that link is notified that a new *listener module* has been added. The source module keeps a list of all listeners on each outgoing data port. In this arrangement, each source keeps track all of its listeners. The listeners have no information about which sources are connected to them, which reduces the system complexity.

5.2.5 Recompiling a Module

One of the unique aspects of eFuse's data-flow system is that module implementations can be changed and recompiled dynamically. The recompiled class file is automatically incorporated into the running system. This allows for rapid development and debugging of modules within an integrated system.

When the user clicks "Evaluate" in the module edit window, the client sends the entire source code for the module back to the server. A CGI script on the server takes that information, generates a Java source file, includes the appropriate package name into the source file, and puts it into the appropriate users directory. The CGI script then executes a Java compiler on the server to create a corresponding class file containing appropriate bytecode for the Java source file. This compiled class file is located in the same directory with the source file. The client application can then load that class file by specifying the full

package and class name and requesting that the client Java Virtual Machine dynamically load that class.

5.3 Client-Side Data-Flow Implementation

In section 5.1 we described the implementation of the actions that occur in response to user interaction with eFuse. This sections describes the underlying implementation of the client-side data-flow system.

5.3.1 The Data-Flow Model

eFuse uses a simple data flow model of sequential execution. Software applications can be created by linking the output(s) of stateless components to the input(s) of other stateless components. In this situation, an application can be modeled as a *directed acyclic graph (DAG)* of stateless components. Because the input(s) and output(s) of the components in a data-flow system are defined explicitly, an individual component can be treated as a true *black box* abstraction of the operation it performs. As a result, data-flow systems are highly modular and, in eFuse, can be formed, changed, and extended quickly. Data-flow systems also naturally extend to parallel operation, as each component processes data independently of the other components in the system.

Textbook algorithms are generally expressed as functional units that given a particular input, produce a particular output. These algorithms are generally expressed as stateless, self-contained units. Consequently, textbook algorithms naturally fit well as components in a data-flow system.

In order to dynamically create applications using the data-flow model, an underlying architecture must support the ability to load modules, manage interconnections, and route information from one output to the connected inputs. This underlying architecture needs the ability to inspect a component to determine what inputs the component can receive, and what outputs the component generates.

We have developed the capabilities to do all of this inside the eFuse client -- a Java applet that can run inside any Java-enabled Web-browser. The architecture described here allow students to create applications by dynamically linking modules in a run-time data-flow system. Individual components have graphical representations that can be arranged and linked by the students. Links are typed and can be created, modified, or destroyed as an application is running. Modules run in separate threads, properly handle queuing of incoming data and delivery of outgoing data, and can be dynamically recompiled and automatically inserted into the running application. All of this functionality is automatically incorporated when each new module is created. The user is left to focus entirely on the module algorithm and graphical display.

5.3.2 Anatomy of an eFuse Module

In the eFuse data-flow model, there is no global scheduler, nor is there a global routing table, or even a global list of components. Instead, the module itself keeps track of those modules that listen for its outputs. *Source modules* handle delivery of output data to all attached *listener module*s. Modules also each run as a thread and handle their own queuing of input data. Each module also has a *graphics canvas* for drawing and receiving user interface events. On top of all this, each module supports the capability to dynamically swap in a new implementation of itself. The special implementation of a module described here, along with the automatic generation of code for each new module, allows modules to provide all of this functionality while still remaining easy to implement.

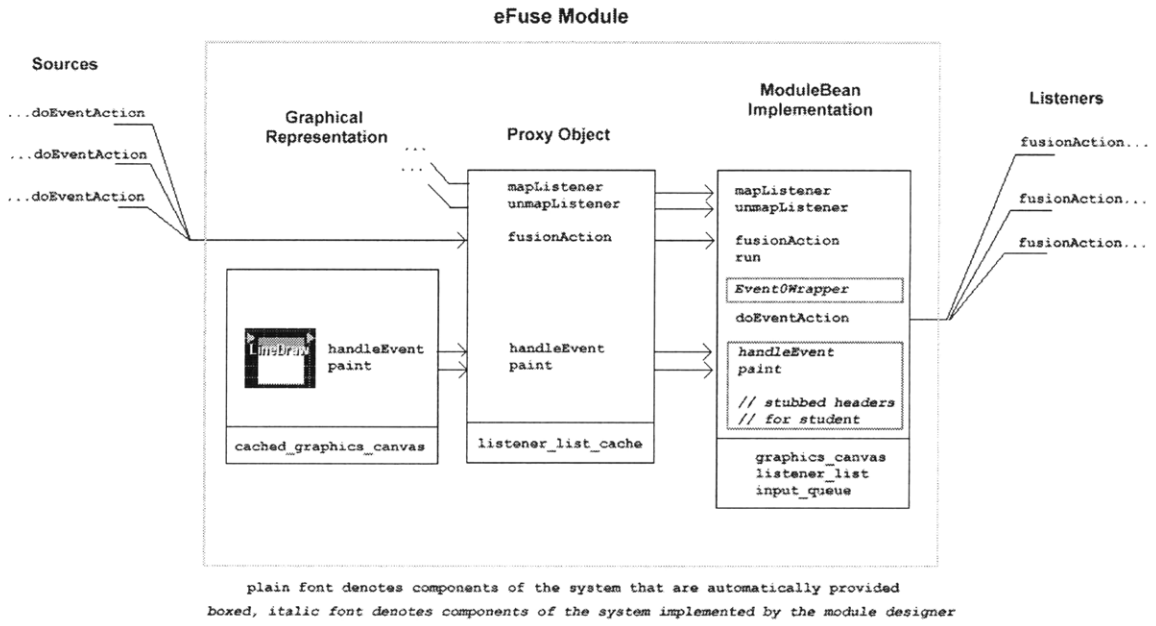


Figure 11: Anatomy of an eFuse Module

Modules can be viewed are three-layered structures (Figure 11). The right-most layer is the individual mutable portion of the implementation. In the actual implementation, this layer of code implements the `ModuleBean` interface. Each module contains a unique implementation of the mutable portion – hereon referred to as the *`ModuleBean` implementation*. Much of the `ModuleBean` implementation is automatically generated for the developer, as described in section 5.2.2. The module designer is left with the task of implementing any necessary initial handling of input data through the `EventWrapper` methods, handling user-interface events (`handleEvent`) and drawing on the canvas (`paint`), and creating function stubs for the student to implement. For instance, a module designer developing the Bresenham module might add the empty method stub `void Bresenham(int x1, int y1, int x2, int y2)` and provide a method called `setPixel` which draws a pixel into the canvas. The student is then left with the task of implementing the last piece of the module – the Bresenham algorithm itself.

The middle layer is the *proxy object*. The proxy object's purpose is to act as a consistent proxy between other modules in the system and the current implementation of the mutable layer of the module – the `ModuleBean` implementation. When a method is called on the proxy object, it in turn calls that method on the `ModuleBean` implementation for which it is the proxy. The proxy object also caches the list of listeners for a module.

The left-most layer of a module in the eFuse data-flow system is the graphical representation of the module. This layer of the module handles drawing the box, drawing the connectors, responding to resize and move events, and informing the proxy object when the module's graphics canvas has changed or when events have arrived inside the module's graphics context.

5.3.3 Transferring Data Between Modules

In the eFuse data-flow model, data is transferred by method calls. Method calls provide a fast, robust delivery mechanism. Specifically Sending data output from a source module is accomplished by calling the `doFusionAction` method with the data to be sent and the unique identifier representing the correct output port. The `doFusionAction` method automatically takes care of sending the data to all registered listeners in the `listener_list` by calling each listener's `fusionAction` method with a cloned version of the output data. `fusionAction` places the new data on the input queue and resumes the thread's execution. The main loop of the thread iterates over all input queues and checks for available data to process. If new data is available, the thread pops the data off the queue and calls the appropriate `EventWrapper` method written by the module designer. Event queues are checked in a round-robin fashion. If all input queues are empty, then the module thread suspends.

A module can generate many output actions for just one input action, and these actions can occur throughout the execution of the algorithm. There is not a one-to-one mapping

between the number of inputs and the number of outputs, nor does the execution have to end before an output can be sent to the next module. This allows module designers to control the granularity of the data elements sent through the system.

With the development of Java *Remote Method Invocation (RMI)*, the same delivery mechanism can be used across applications running on different machines [SM97]. For instance, a module that a student is implementing could be linked across the network to a module in a teaching staff's concept graph, which would allow the member of the teaching staff to view the outputs that the student's implementation is generating.

5.3.4 Loading a New ModuleBean Implementation

When the user clicks the "Evaluate" button in the edit window in order to compile any changes to the implementation, the new implementation is compiled as a new class file with a unique name based on the current version that the developer created.

As mentioned before, Java can dynamically load class files over the network, but if a class of the same name has already been loaded, then the Java Virtual Machine's class-loader will load the class from the cache. Thus, it is impossible to load a new implementation of a class with the same name as a previously loaded class. In order to resolve this, the eFuse system automatically renames the class to the next available version number before compiling. For example, a LineDraw module might be automatically renamed to `LineDraw_3` if the developer is working on the 3rd revision. The developer is able to see the list of revisions and reload any previous revision by simply clicking on it.

Each revision of a module implements the same basic interface, the `ModuleBean` interface. When a new revision is loaded, it is treated as an object of type `ModuleBean`. This allows the new module, even though it is a new, different type, to be treated in a fashion identical to the previous revision.

A module is incorporated into the existing system in the same way it was loaded. The class is requested by name (`LineDraw_3`) and a new instance of the object is created. In this case, however, the proxy object and the graphical representation of this module already exist in the system.

To understand the process of loading the new implementation at a high level, imagine removing the right-most layer of the module shown in Figure 11 – the `ModuleBean` implementation. All the connections into the `ModuleBean` implementation have been broken, as have the links to the external outputs. Note, however, that all the input links coming from source modules and the graphical representation are preserved. Loading the new implementation now simply involves re-establishing the links from the proxy object to the new `ModuleBean` implementation, and re-establishing the links to the listener modules.

In order to re-establish the links to the proxy object, the proxy object's reference to the old module implementation is replaced with a reference to the new implementation. Replacing the old reference involves first telling the old module implementation's thread to die, then reassigning the reference, which causes the garbage collector to remove the object associated with the old module. Now, data flowing from sources to the proxy object is passed to the new `ModuleBean` implementation.

Re-establishing connections from the module to other *listener* modules is more complicated. When a link was originally created with a call to `mapListener`, the proxy object for the source module first cached the information about the unique identifier of the outgoing port, the reference to the listener's proxy object, and the unique identifier for the listener module's port, before passing that information to the module itself. When a new `ModuleBean` implementation is loaded in place of the old version, the proxy object enumerates its cached list of listeners and tells the new implementation to add those listeners. The new implementation has been completely swapped in place of the old implementation.

5.3.5 Support for Multiple Inputs and Outputs

As shown in Figure 11, each input port can receive data from many sources. Similarly, each output port can send data to many sources. While the module illustrated in Figure 11 only shows one input and one output, each module can have multiple inputs and multiple output ports. These input and output ports are identified by unique identifiers. Internally, the module keeps track of which action occurs on which port by internally passing the unique identifier that refers to that port. Internal data structures, such as input queues and lists of listeners, are replicated in arrays and indexed by the unique identifier. For instance, a module with three output ports will contain three lists of listeners, one for each port. Similarly, a module with three input ports will contain three input queues for storing incoming data that is yet to be processed.

5.3.6 Type-Checking Inputs and Outputs

The output type of the source module must agree with the input type of the listener module. In order to help users see which output types match which input types, different types are distinctly colored. When a user clicks on an output arrow of a module, eFuse can optionally display a graphical image representing that data type.

The data-flow system also automatically checks on-the-fly if a module can receive certain data types. This is important because certain data types may be subclasses of another. For instance, a listener module may be able to handle any object of type Vector and a source module may generate an object that is a subclass of Vector. The listener's output port would have a different color because the input type to the listener differs from the output type of the source, but the listener is still capable of processing information from the source. In order to dynamically perform this type checking, each source sends an instance of the data type it generates to the listener. The listener then returns true if it can handle data of that type. This allows run-time type-checking, while supporting arbitrary Java objects as data types. No global registry of data types is necessary, nor does this require a specialized object

naming pattern. Instead, type checking is done solely by inspecting an example input object and determining if it is of the correct type. Module designers are required to pass only data of the specified type through a given output port, but can then guarantee that any connected listener is capable of processing it.

5.3.7 Parallel Execution of Modules

Each module runs in its own thread. Consequently, multiple modules can be processing data simultaneously. Incoming data is queued in each module until it can be processed. When a module is not processing any data, it suspends itself until new data arrives. The system does not make any guarantees about how quickly data will be processed by any individual module, though such a real-time scheduler is one possible extension [Har97]. Adding data to the incoming queue is a synchronous operation. Which thread model is in effect depends on Java's underlying threading architecture. If the Java threading architecture supports multiple processors, then data-flow applications created with eFuse systems will transparently execute on separate processors.

5.4 Data-Flow in Practice: A Classical Polygon Rasterization Pipeline

Using the data-flow model for application building in eFuse, we designed, implemented, and tested the classical polygon rasterization pipeline as a way of teaching students about the field of three dimensional computer graphics. This section describes our design process, the data types defined as inputs and outputs to the pipeline, the implementation of the modules, the creation of the full system, and what we learned from this experience.

5.4.1 Designing the Pipeline as a Data-Flow Application

We began by brainstorming all the modules in the graphics pipeline. We listed all the input data necessary, the transformations on the scene data, the clipping and visibility operations, and our options for rasterization. The input data for the pipeline consists of a scene description, information about the synthetic camera, and lighting information.

Transformations include operations such as transforming the polygons from world space to eye space, converting vertices to homogenous coordinates, performing a perspective or parallel projection into two dimensions, and converting the two-dimensional projection into screen space. Clipping and visibility checks included back-face culling, view-frustum clipping, and Z-buffering. We arranged these in a data flow organization. During this process, we visited several common permutations of the model. For instance, there are different methods for performing visibility resolution, such as scan-line algorithms or z-buffers.

As we organized the modules, we roughly described the information passing between each module. As you can see in Figure 12, the data-flow pipeline we designed by hand was more involved than the naïve classical rasterization pipeline presented in Figure 2.

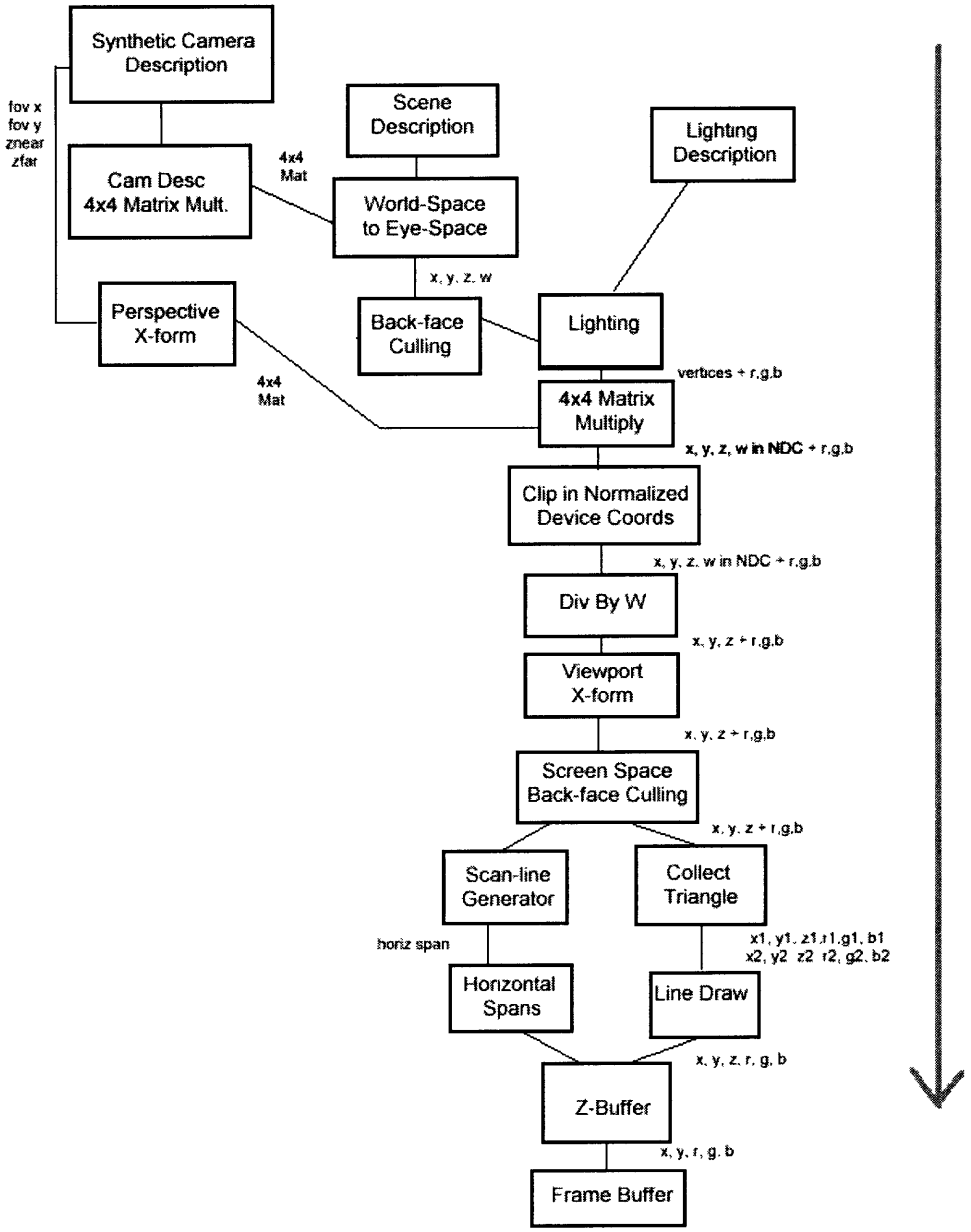


Figure 12: Rough Design for a Classical Polygon Rasterization Pipeline as a Data-Flow

Finally, we stepped back and determined that the best data type for this particular data-flow application was a series of tokens which each module could optionally process, collect, or pass down the pipe. This became the ThreeDToken.

We determined that data would flow through the pipeline largely as triangles. A START_TRIANGLE token would denote that all incoming data until the END_TRIANGLE token was part of a single triangle. This way, modules working only on the vertices of the triangle could process those vertices directly, but those modules requiring an entire triangle could keep a stack of inputs and perform the processing when the END_TRIANGLE token arrived. This meant that the data passing between modules in the pipeline was fine-grained, but modules could accumulate data to process large-grained data structures.

5.4.2 Defining the Data Types

Once we had the initial design, we began by implementing the ThreeDToken. Each ThreeDToken has an integer representing the type of data contained in the ThreeDToken, and a reference to the data. For this application, we chose not to use the automatic typing-checking features of the system and instead use information contained within the ThreeDToken to help the application identify the type. This model was chosen because it allowed us to design modules that could filter based on the data type, potentially passing data right through a module without any processing. We could have defined specific types for each data item flowing between modules, but certain modules would require multiple inputs and interconnections that did not flow as a natural pipeline. Given the intent to make this particular example appear to the students as much like a direct pipeline, we chose to use the same data type for all module inputs and outputs.

Once the Java object representing the ThreeDToken, and the possible data types that the ThreeDToken might contain (Camera, Vertex, Fragment, etc.) were created and compiled, we added them to the `projects.graphics.modules` package and placed them on the server for eFuse developers to use. We also added the `projects.graphics.modules.ThreeDToken` object to the list of available input and

output types. Now, when a module developer created a new module, the ThreeDToken object was listed as an available input or output data type.

5.4.3 Instantiating the Modules

Implementing the modules was done entirely in the eFuse environment. We began by identifying the core set of modules required to make a very simple rasterizer. We chose to rasterize only a wireframe representation of the scene, eliminating the need for visibility resolution. We decided to support a full synthetic camera with perspective projection from the beginning.

Creating a new module was simply a matter of clicking the "New Module" button, selecting ThreeDToken as the input and output of a new module, and entering the module's name. The automatically generated module already compiled. Each module in the pipeline simply checked to see if the data was a type that it wanted to process. If so, then the algorithm for that module was performed on the data and new modules were generated.

We built the system by first starting with the last module in the pipeline, the ViewPort, and incrementally added functionality to the front. The ViewPort was responsible for listening for ThreeDTokens representing fragments – that is, data with an x, y, and Color component. The ViewPort then displays this information inside the module's graphical space. Next we implemented the LineDraw routine, which, given an EndPointPair object, determines the appropriate pixels to illuminate between those two points. This module generates fragments that are sent to the ViewPort. CollectTriangle accumulated the three vertices passed between the START_TRIANGLE and END_TRIANGLE tokens. Upon reception of the END_TRIANGLE token, CollectTriangle generates three EndPointPair objects that could be sent to the LineDraw module.

The ViewportProjection, FiniteViewProjection, and WorldtoEyeSpace projection modules all perform operations directly on vertices. These projections first wait for a camera token

and then store all the camera information. The projections are applied to incoming vertices based on the current camera information; the transformed vertices are then passed to the next module. Ultimately, the ViewportProjection passes fragments to the triangle routines. Finally, the pipeline was completed with a single module, GenericTester, that generates scene and camera tokens that comprise a tetrahedron. This simple classical rasterization pipeline was implemented in seven modules. By far, the most extensive module was the LineDraw module with approximately 100 lines of code. The other modules required much less code.

Debugging was greatly aided by the fact that we could hit the "Evaluate" button and our bug fixes were immediately incorporated into the running system. We also chose to implement each module with its own internal test cases. For instance, by clicking in the graphical area representing the LineDraw module, the module performs an internal test of the algorithm which causes the fragments for a test line to be drawn in the connected ViewPort. Similarly, the CollectTriangle module had an internal test case which sent a series of tokens to its own routines which represented random triangles. If the module was working correctly, the LineDraw routine would be called, which would lead to random triangles appearing in the ViewPort.

Since then, we have implemented a ZBuffer module and a FillPoly module which handles visibility resolution and drawing of filled polygon models instead of a simple wire-frame rendering. These modules have been similarly simple to build and design.

5.4.4 Putting the System Together

Combining the modules was simply a matter of drawing the links between them. Once the links were connected the system automatically ran. In fact, we were able to have multiple versions of the system running on the screen simultaneously given that each module runs in

its own thread. When the system was complete, we pressed "Save CG" and the concept graph was saved onto the server so that we could recall it later.

Building the initial modules and putting the system together took a couple students well versed in computer graphics only a few hours to complete.

5.4.5 What We Learned

We were impressed by the speed of implementation in our own system (we initially allocated much more time to completing the pipeline than was necessary). We were also impressed by the robustness of the architecture. The data-flow model was a natural fit to the graphics pipeline, and those students who implemented parts of the pipeline have expressed that even though they already understood the pipeline, they had a much better sense for it after implementing it in eFuse.

We learned that adding internal test routines for modules was a particularly helpful technique that we had never considered in detail while building eFuse. We also learned that the limitations of a poor editor hinder debugging. The lack of line numbers made it especially difficult to find compiler errors.

Overall, the data-flow model and the eFuse architecture made the process of building this system efficient and highly instructional.

Chapter 6

Results: eFuse Trial in a Large Course

In the fall of 1997, we deployed two modules to the students in MIT's introductory computer graphics course, 6.837, taught by Professor Seth Teller. We desperately desired real-world feedback on the system. The initial results were very encouraging, and the system stood up well to the activities of over 100 students.

6.1 Selecting and Designing the Trial Assignments

We selected two modules for the students to implement: Bresenham's line drawing algorithm, and the Cohen-Sutherland line clipping algorithm. We designed module implementations that included interactive, reference, implementation, and difference modes. The modules were placed in the concept graph, but with no apparent hierarchy and no data-flow interaction (the data-flow aspect of the system was under development). These two modules were selected because they were two early algorithms and important in computer the computer graphics curriculum. The algorithms were also considered relatively straight-forward, though not trivial. We anticipated that students would be able to implement both algorithms quickly, but chose to err on the side of extra time in the event that students had difficulty acclimating to eFuse, or eFuse was unavailable for significant periods of time.

Students were given one week to complete each assignment, which required implementing the entire algorithm in Java in eFuse. The course staff (one professor and three teaching

assistants) and the eFuse developers were online to help answer student's questions and make sure the system was available. When students completed the assignment and used the "Submit" button to turn in their final version, they were presented with a form which asked them to anonymously rate the eFuse experience.

6.2 Evaluating the Effectiveness of the Trail Run

Overall, we were very pleased with the initial run. The system performed well, though there were difficulties. In general, the students gave the assignments high reviews while providing plenty of constructive criticism.

Judging from the evaluations students submitted, they were pleased with the system and appreciated the features available. At the same time, the students found certain aspects of the system frustrating. The survey results are listed in the following two tables.

Bresenham Assignment (121 responses)	**Avg. Rating (5 is the highest, 1 is the lowest)**
How well do you feel you learned the material?	4.37
How well was the assignment organized?	4.00
How well was the staff able to answer your questions?	4.19
How difficult did you find this assignment?	2.25

Table 1: Student Survey Responses for the Bresenham Assignment

Cohen-Sutherland Assignment (109 responses)	**Average Rating (5 is the highest, 1 is the lowest)**
How well do you feel you learned the material?	4.80
How well was the assignment organized?	4.08
How well was the staff able to answer your questions?	3.92
How difficult did you find this assignment?	3.20

Table 2: Student Survey Responses for the Cohen-Sutherland Assignment

Overall, students felt they learned the material well, found the assignment well organized, and gave high marks to the staff for answering questions. We were curious about how difficult students felt the assignment was, as it was designed to be straight-forward, but the eFuse system might impose added difficulty. The results told us that students found the assignments to be relatively straight-forward to mildly difficult, which was again encouraging. The results of the question asking how long each student spent on the assignment backed this claim. Most students selected the lowest time bracket for both assignments: 1-3 hours.

Students provided constructive feedback in their comments. Some students were affected by the few times that the system crashed, and found this highly frustrating. eFuse successfully maintained the state so that students could start back where they left off when the system crashed. For some students, the Java implementation of the browser they were using was broken, requiring them to find another machine. At the same time, some students complained of long startup times, which was related to the large number of class files that the client downloaded. These have since been archived into one file, greatly reducing

startup time. While most gave praise to the simple compiling mechanism, many complained that the simple Java text editor was difficult to use, and compiler feedback was almost useless because line numbers were not consistent with the student's view of the code. Students appreciated the collaboration functions, but felt the interface could be improved. Overall, the students found the manual, reference, implementation, and difference modes useful in helping them achieve a correct implementation.

For the teaching staff, the ability to load a students implementation was the most useful feature, but often the most confusing. The eFuse implementation did not initially high-light cases where a student's code did not compile successfully. When the code did not compile, the previous version was left running. This occasionally made it unclear as to which version was running. Once this was resolved, the system was easier to use. The staff found that having all of the student's revisions available was useful in helping to determine the course of events that the student pursued, but occasionally left them wondering "what in the world was this student doing?". The integrated grading form made it easier to provide feedback, but the inability to review the grades was a limitation. Administering the assignments and setting up the user lists was easy, largely because it was handled by the eFuse developers. Overall, the staff enjoyed the ability to monitor student activity and quickly load a student's implementation in order to provide help, or evaluate for grading.

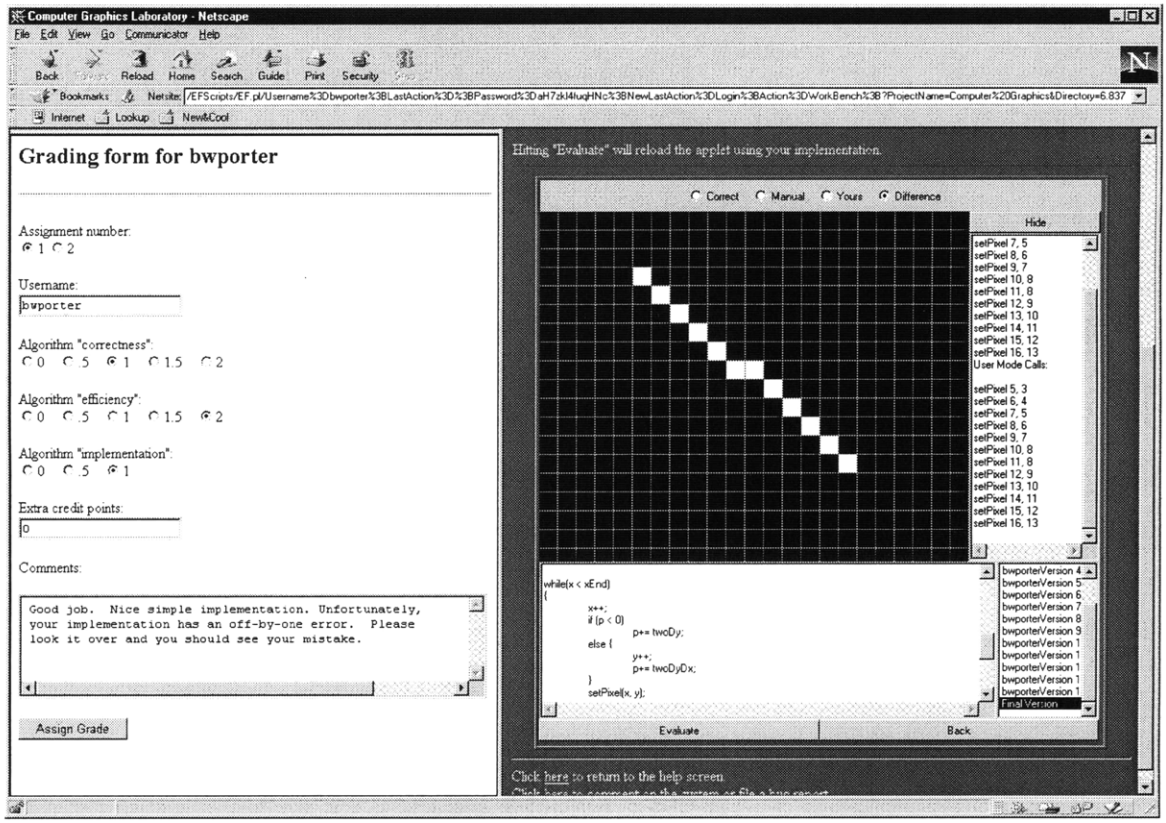


Figure 13: Staff Evaluation Space Used the Fall of 1997

From the eFuse developer's viewpoint, the client-server architecture held up well and, with the exception of one glitch, was very robust. We designed the system to be available 24 hours a day, 7 days a week. For the two weeks we ran the system, it was available 95% of that time, occasionally with more than 25 students online at once. This exceeded our conservative expectations. Unfortunately, the 5% that the system was down was obviously very frustrating to the students, and caused some lost sleep for the eFuse developers. The system failed briefly approximately every 36 hours. The point of failure the same each time and was related to the implementation of the Java Virtual Machine for Windows. We believe has since been resolved. All other components of the system worked well. Given that the compile and save process is separated from other components of the server infrastructure, students could continue to do some work in the midst of the failure and we are not aware of any student who lost work due to the failures.

In retrospect, the initial deployment was an encouraging success that provided us with invaluable feedback.

Chapter 7

Future Work

Even though eFuse has a tremendous amount of functionality and accomplishes our objectives well, we continue to see more and more opportunities to add capabilities to the system.

In this chapter, we describe several possible future directions for eFuse. These active areas of research and development can be generalized as: improving the overall integrated environment for the students; developing new capabilities for the teaching staff to track student progress; and building upon the architecture to extend the data-flow model.

7.1 Student Environment

Much of our research and development efforts have focused on providing students with a platform for collaborative pedagogy with the facilities to develop, execute, debug, and verify algorithmic modules. The basic components of this vision are in place. Future work in this area will focus on improving these capabilities. Specifically we hope to improve aspects of the development environment such as the editing capabilities and debugging capabilities. We also hope to support automatic documentation of code developed in eFuse. Finally, we would like to extend the communications capabilities available to students.

7.1.1 Improving the Editor

In our initial trial run, students felt the editor was one of the weakest components in the system. Basic functionality such as search and replace, and line numbers was not incorporated into the simple editor. Developing a Java editor that supports these features, along with advanced features such as auto-indentation of code, color high-lighting of keywords, and keyword-enabled context-sensitive help will be included in future versions of the editor. Much of this work involves simply selecting an incorporating a Java-based editor into eFuse.

Saving files based on a specific name instead of purely by version numbers is another feature that students requested of the eFuse system. Developing this capability is complicated by the strict Java class naming conventions and the special source rewriting techniques that eFuse uses to give each class a different name. Determining the most effective way to accomplish this is another possible area of development.

7.1.2 Improved Debugging Capabilities

Currently, eFuse only supports simple debugging based on embedded print statements. Ideally, these modules are simple enough to implement in their own right that advanced debugging capabilities are not as necessary. The strict modularity between modules should further ease debugging.

Simple editor features such as color high-lighting and the ability to click on a compiler error and go directly to the appropriate line number will improve debugging of simple syntactical errors. Unfortunately, we do not have the ability to control the Java Virtual Machine, so stepping through code line by line is a difficult process. Research could be done in ways to do this, potentially through automatic code rewriting or integrating a better virtual machine.

7.1.3 Automatic Documentation

The Java Development Kit includes the JavaDoc utility which parses comments from a Java source file and automatically generates an HTML page with the class specifications in the source code. Integrating this as a feature of the automatic compiler on the system's back-end would help students understand modules that are provided for reference, and help the teaching staff understand a student's code. Integrating links to the automatically generated documentation with keyword-specific help in the editor would allow students to easily find help on any Java class in the system. The eFuse architecture supports this type of change easily.

7.1.4 Improving the Communications Capabilities

Integrating the collaboration facilities into the concept graph will improve the overall interface. A student should be able to click on an avatar and begin sending messages back and forth with a fellow student or a member of the teaching staff. More advanced capabilities could be added to the white board to allow more expressive communications. 280, Inc.[1] has done pioneering work in this area.

Improvements in bandwidth and streaming technologies increase the viability of one- or two-way voice and video. Simple text-based communications have the advantages that they can be multicast easily and processing the information visually is quicker in some respects than hearing it. However, text-based communications can be limiting when a member of the teaching staff is trying to explain the intricacies of a concept.

Network collaboration is an interesting area of research especially as it applies to distributed learning systems. We feel eFuse provides a solid framework for designing and researching these technologies.

---

[1] http://www.280.com

7.2 Course Administration

While eFuse contains many features to aid the teaching staff, this area is an important focus of our future work. The most common feedback from the teaching staff was that they needed a way to create grading criteria, easily generate Web-based grading forms, and automate distribution of grades to students. In general, we want to improve the staff's ability to track student progress throughout the course, and improve the general administrative features.

7.2.1 Improving the Grading

eFuse is an excellent platform for loading, viewing, and evaluating a student's implementation. Currently, however, the forms for assigning grades to students were generated by hand for each assignment. Grades were placed in a directory and automatically emailed to students in a single batch. This process was not integrated into the system. The staff was not able to easily recall these grades in the event of discrepancies. In the future, the staff should be able to easily create grading criteria documents which can be converted into grading forms, printed, emailed, archived, and retrieved. Overall, while evaluating the assignments was greatly enhanced by eFuse, we still have work to do to make the administrative process of grading easier.

7.2.2 Better Student Progress Tracking

In the future, we would like the teaching staff to be able to click on a student's avatar and see a complete HTML report of the student's progress in meeting checkpoints, the grades the student has received, and information about areas that the student has found confusing in the past. This will greatly help the teaching staff track each student's progress, and cater responses to each particular student. For instance, a student may not have any trouble with the algorithmic material, but this same student may have weaker programming skills relative to other students in the class. By tracking the student's progress and archiving comments

about the student, the teaching staff can customize the help they provide. In order to accomplish this, the system needs to include specific checkpoints that students must pass. These checkpoints can be simple "problem sets" or fine-grain markers such as implementation of components in an assignment. The system also need to support annotation of a student's record with information about performance, questions, and other student-specific information. Detailed reports will help the staff get to know the students in a large class better, and provide a record that can be used in final grading and when writing recommendations.

Specifically, we would like to be able to track progress at a level such that we can tell whether or not the student is implementing an algorithm by successive approximation, or through true understanding of the material.

7.3 Development Directions

The data-flow architecture opens the door to further research directions. The most active area of further research is in our work on allowing modules to connect to other modules on remote machines. We also feel that providing basic data-flow constructs will help students and teaching staff build more complicated systems. Abstracting these systems as components in their own right is another important innovation in the system that we are working towards. Finally, determining the best way to integrate manual interaction with the data-flow system and the resultant visualization is an open question.

7.3.1 Data-Flow Components Across the Network

We envision a system in which students can send the output of their module to a member of the teaching staff. The student's output could then be linked across the network into the staff member's own module, perhaps to perform a simple test. This would also enable a number of students to simultaneously connect the incoming data from a member of the teaching staff to an output module on the student's screen. This would allow the teaching

staff to demonstrate how a system works to all students simultaneously. Students could also collaboratively design modules in a system and build a system from the modules running on each student's client.

Currently, the technology exists to do this in the Java Remote Method Invokation (RMI) technology. This technology allows one class to call a method on a remote class through a proxy wrapper. This matches well with the eFuse data-flow design model. This work requires designing a method for specifying that an output can be streamed across the network and then announcing the availability of that output to other students or teaching staff members in the system. A layer that automatically instantiates the RMI connection is feasible, but non-trivial.

We believe this type of collaborative execution of a system distributed across a network will be a powerful instructional tool.

7.3.2 Built-in Constructs

Building a library of basic data-flow structures is another direction for future work on eFuse. A simple switch that users can click to reroute data is one simple example. More advanced components include loop constructs and data filters. eFuse modules automatically support multiplexing of outputs, but simple filtered multiplexers, or modules that guarantee in order delivery of events might be provided to all eFuse users. Diagnostic modules that display information about data types as they flow across the system would also be useful. Clearly, a number of simple data-flow components would be generally useful and will be developed as future enhancements.

7.3.3 Abstraction

Once a user has developed a system, that system has the same properties as a module – a set of inputs and outputs with a specific interface definitions. This system of modules could be abstracted into a single module. For instance, the transformation stages in the classical

polygon rasterization pipeline might be abstracted into a single module representing all the transformations in the system. The challenge in implementing this is determining a way to graphically represent the action of abstracting a module, then assigning the abstraction a name, maintaining information about the input and output ports, and serializing the abstraction for storage on the server. The advantages to this abstraction are clear and the best design for this abstraction is under consideration.

7.3.4 Overlaying the User-generated Input with System-generated Visual Output

One of the open questions for eFuse right now is how best to represent areas in which students should provide inputs from the user into a full system and how best to display the outputs. For instance, a voice-based or haptic interface could be used for user input into the system, and sound or tactile response might be the form of the output.

Currently the inputs to the system are either programmatically generated, or generated through mouse actions by the student. The outputs are displayed graphically. Traditionally, interactive programs are interesting because the user interacts directly with the graphical representation. In certain cases, the input is best represented as separate from the output. In either case, the user interaction is the input into the system and the display is the output. Representing this visually can be confusing. If a module both displays output and takes input, then the system becomes circular, which tends to break the clear model of a data flow. Overlaying one module on top of another allows for separated input modules and output modules, but can lead to the same circularity. In eFuse right now, there is no clear indication to the user where mouse input can be given by the student. Resolving these issues is an important area of future work.

7.4 Module Development

Developing compelling modules is an open area of research. Effective visualizations of complex algorithms can be challenging. eFuse pushes into new territory in algorithm visualization with both the interactive modes and the difference modes.

Creating interactive versions in which students can change the inputs and view the outputs has been an active area of research in systems such as BALSA, Zeus, and the *Interactive Illustrations* project[Bro+84][Bro91][Try97]. Besides creating this graphical visualization of the "reference" implementation of an algorithm, we expect module designers to create visualizations in which the student must interactively *perform* the algorithm – the algorithm's interactive mode.

We also expect module designers to create visualizations that high-light the "difference" between a correct and an incorrect algorithm. These difference visualizations have proved to be useful for students when attempting to find problems in their own implementation. Determining the best way to visually portray these differences is not a trivial task for many algorithms, and an important area of future work.

We foresee developing a number of these modules as part of the eFuse project. We also expect that as eFuse become more widely used as a platform for collaborative pedagogy, researchers, students, and educators will contribute modules to the system, thus creating a large shared public-domain library of instructional algorithmic modules.

Chapter 8

Conclusion

eFuse is an instructional, collaborative, Web-based, platform for learning and teaching algorithmic concepts through implementation and verification. We were inspired to begin this project based on our observations that more powerful personal computers and faster networks would allow location-independent access to instructional material, enable more interactive instructional environments, and facilitate collaborative learning.

Specifically, for students we set out to design a system that simulated a programming laboratory that: provides an intuitive visualization of course material; allows location independent access; hides the details of compilation, execution, visualization, and validation of an algorithm implementation; and enables and encourages collaboration. For the teaching staff, we set out to design a system that allows simple centralized administration of system resources, permits instructors to select the material that they felt was most important and organize it effectively, and finally to allow the staff to easily access student work in order to provide assistance and evaluate a student's progress. From the developer's perspective, we set out to create a platform independent system on both the client and server that allowed students to work within the bounds of a textbook algorithm but also creatively build systems from these components.

The eFuse system accomplishes these goals. Students interact with and implement algorithms in a virtual programming laboratory, or workbench. Students can implement

individual algorithms, or build larger systems by combining modules into data-flow systems. Modules are designed to support advanced techniques for algorithmic understanding, including the interactive, implementation, reference, and difference modes for each. The server compiles and tracks all code revisions automatically, while the client is capable of dynamically loading new implementations into the running system. Students can see visual representations of peers in the system and use text along with graphics to effectively collaborate. The system has remained server- and client-independent through the use of a Web server as the primary server platform along with a concerted effort to use languages available on many platforms, including Java and Perl. The data-flow architecture enables students to easily create new modules and extend existing modules in order to build and extend their data-flow systems. The teaching staff can effectively manage student lists and assignments. They are also able to instantly load a student's implementation and view the function of the algorithm while also viewing the implementation directly. The teaching staff can then use the system for either communicating with a student, or for evaluating the final implementation and providing feedback. Initial feedback from students and staff who used the system has been extremely encouraging.

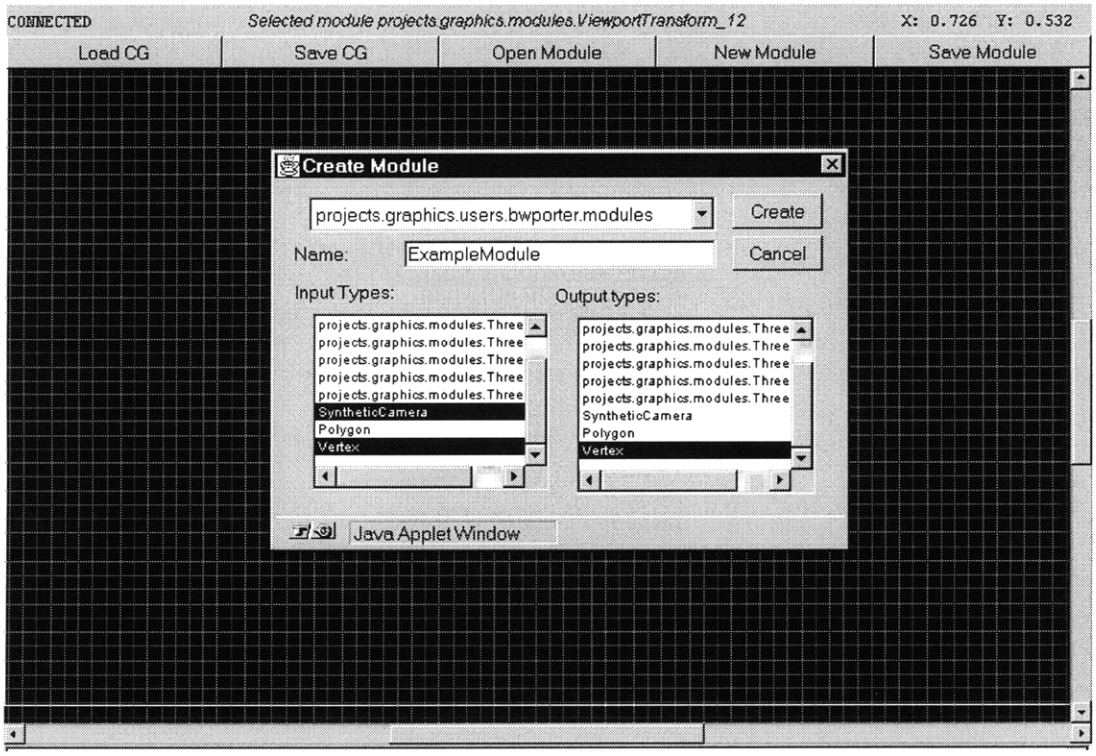
While eFuse goes a long way toward meeting our objectives, there is still much to be done. We hope to improve the student interface, develop a methodology for automatically tracking and reporting student progress, and increase the capabilities of the data-flow system to provide simple data-flow structures and abstraction. We also need to continue to develop modules which support the full interactive, reference, implementation, and difference modes. At the same time, we need to encourage others to use the system and develop new modules.

In conclusion, eFuse is a dynamic new system which greatly extends traditional software development environments, on-line instructional systems, and techniques for algorithmic visualization. eFuse offers an instructional environment unlike any other.

Appendix A

Creating a Module

This document describes the process of creating a module. Creating a module is meant to be a simple task that allows the user to focus on the module implementation rather than the details of supporting the data-flow architecture. Each step is illustrated in detail. As an example, imagine creating a module that takes a synthetic camera and vertex, then generates a vertex transformed into the a world space vertex into a vertex in eye space. First, click "New Module" in the concept graph toolbar.



Next select the location for the module. Modules can be placed in a number of places depending upon the user's permissions. Students can only create modules in their own local directory. Teaching staff can also place modules in the project directory. eFuse administrators can also place modules enter the name of the module. In this example, we chose the user's local directory.

Next, determine the name of the module. Module names should match Java class file naming conventions. In other words, they can not contain spaces, should begin with a capital letter, and should roughly describe the function of the module. Our example module is generically named "`ExampleModule`".

Finally, multi-select the inputs and outputs. The list of available inputs and outputs is maintained in the datatypes.list file on the server for each project. Select the input data type or types and the output data type or types. We chose `SyntheticCamera` and `Vertex` as the input. The outputs is a single `Vertex`.

Upon clicking "Create", eFuse will automatically create a module with the name, location, and input/output ports specified. The module designer can now focus on implementing the routine. To edit this module, right-click on its graphical representation.

For each input, the module creation process creates a method named "`Event[n]Wrapper`" where [n] represents the port number. This method accepts an object of the type expected on the input port. Each time data arrives on that particular input port, this method is called automatically. In our case, whenever a new `SyntheticCamera` is sent to the module, the following routine is called:

```
public void Event0Wrapper(SyntheticCamera e)
{
    // This gets triggered when data arrives on port 0
    // You should make the appropriate call to an algorithm
    // here for this module.
}
```

Inside this method, you must make the appropriate calls to execute the algorithm based on this data. When you are ready to push a piece of data out along the output port, you must pass an object of the proper type to the "doEventAction" while also specifying the output port. In this example, once the eye point Vertex is calculated, you would call `doEventAction(myLocalVertex, 0);` where `myLocalVertex` is a variable representing a `Vertex` object, and 0 is the index to the first port.

You can also control the graphics display and event handling for events which occur inside the module space. Each module has a local variable name `_graphics` which is a standard AWT Graphics object which represents the screen space for that module. You can perform standard Graphics object calls on the `_graphics` module such as `drawRect`, `drawLine`, etc., then call the "`update`" method to force the module to redraw (for instance to show the result of the latest algorithm execution). Each time the system requests that the module redraws itself (for instance, when the concept graph becomes visible), the module's "`paint`" method is called. You can write code for the `paint` method which properly redraws the module.

```
public void paint(Graphics g)
    {
        // This implementation of paint automatically performs
        // double-buffereing

        // Call update if you want the module's drawing area to be refreshed
        // update();

        // EXAMPLE (Draws a red rectangle):
        //        Color c = new Color(255, 0, 0);
        //        g.setColor(c);
        //        g.fillRect(2, 2, 14, 14);
        //        update();
    }
```

Similarly, whenever a user-generated interface event, such as a mouse click, occurs within the module, the "`handleEvent`" method is called. The `handleEvent` method for a module has exactly the same form an properties as the `handleEvent` method in a standard Java

graphical component. Returning "false" will cause the concept graph to attempt to handle the event. Returning "true" informs the applet that this event has been fully processed and should not be processed by any other component. The default `handleEvent` method that is automatically generated looks like this:

```
public boolean handleEvent(Event evt)
{
    if(evt.id == Event.MOUSE_DOWN)
    {
    }
    if(evt.id == Event.MOUSE_UP)
    {
    }
    if(evt.id == Event.KEY_PRESS)
    {
    }
    return false;
}
```

Once the changes to the automatically generated code are complete, you can test those changes by simply clicking "Evaluate" in the edit window. Compiler output is shown in the display window in the lower portion of the edit window. Once the module compiles successfully, it will automatically begin running within the concept graph and you can begin debugging.

To recap, creating a module simply requires defining the location, name, input, and outputs of the module. The `Event[n]Wrapper` is called whenever a new input is received. Calling `doEventAction` with the output object and port number pushes data along the output port. Performing methods on the `_graphics` variable and calling the update method will redraw the graphics area of the module. Finally, handling user events is simply a matter of implementing the `handleEvent` method. Have fun building your modules!

Appendix B

eFuse Operator's manual

This document covers the steps necessary to install eFuse on a Web server and the location of specific system resources. We assume a standard Windows NT setup, though eFuse is designed to work with any platform that supports a Web server, Java, and Perl. We also assume that you have received this distribution as a compressed archive that preserves our directory structure. This is a dynamic document; the latest version will be available off the project home page found at ***http://graphics.lcs.mit.edu*** . In the future, we hope this process will be completely automated by an install script.

1.0 Installation

Installing eFuse involves setting up a Web server, installing Perl and Java, creating an EF directory for all of the files, updating the Web server to point to these files for its scripts and documents, installing two services, and updating the configuration file for your system.

1.1 Setting Up a Web Server

Please follow the directions for setting up a Web server on your platform. We have tested eFuse with both Netscape's FastTrack server for Windows NT, and Microsoft's IIS server.

1.2 Install Perl and Java

Both Perl and Java are available on the Web. Perl is available from `http://www.activestate.com/software/Perl_for_Win32.htm` . Java is available

from `http://www.javasoft.com/` . Installation is generally a matter of simply following the instructions. Perl should be installed such that applications with a .pl extension will automatically be executed by the Perl interpreter. Instructions are provided with the installation kit to explain this procedure. Please make sure your system class path points to the location of the Java files so that scripts will be able to find them.

1.3 Create the EF directory

The base directory for eFuse can have any name, but we recommend `D:\EF`, as the configuration files assume this as the default. Unpack your distribution into the new directory and be sure to expand all the path names.

1.3 Install the Services

Two batch files should be included with your distribution. `EFMsgSrv.bat` runs the eFuse Messaging Server. `EFRegSvr.bat` runs the eFuse Registry Server. Both are written in Java. Test both batch files from your command prompt. If either does not work, it may need to be modified to use your Java virtual machine, which was installed when you installed Java. Please correct each batch file such that it executes.

The next steps will install these two programs as system services in NT, which means they will start automatically when the machine boots up. The included file `SRVANY.WRI` explains explicitly what you have to do to install a service, though we explain the basic steps here.

First, find the `INSTSRV.EXE` and `SRVANY.EXE` applications available from Microsoft. Perform the following operations from the command-line:

```
INSTSRV eFuseMessaging c:\srvany.exe
INSTSRV eFuseRegistry c:\srvany.exe
```

Now, run the registry editor (`REGEDT32.EXE`). Under the key HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\eFuseMessaging:

create a 'Parameters' key and create and 'Application' key of type REG_SZ and specify its value as `D:\EF\EFMsgSrv.bat`

Do the same for the eFuseRegistry under the eFuseRegistry key with the command-line as `D:\EF\EFRegSrv.bat`.

Now reboot the machine. Open the control panel and select Services. Find the two new services and turn them on. Specify that they should be started automatically.

Your registry server and messaging server should be running.

1.4 Specifying the Mapping Between Directories and URLs

Using your Web servers administrative interface, specify the following things:

1. The `D:\EF\Documents` directory should be served as a document directory that has the URL: `http://yourserver/EduFuse/`
2. The `D:\EF\Root` directory should be served as a document directory that has the URL: `http://yourserver/Root/`
3. The `D:\EF\Scripts` directory should be served as an executable scripts directory that has the URL: `http://yourserver/EFScripts/`

1.5 Updating the Configuration Files

Most of the system configuration information is stored in the file: `D:\EF\Scripts\EF.config`

In this file, you must specify the name of your Web server and the path to the eFuse files. If you chose to install in `D:\EF`, then the defaults should work. In general, the defaults will be correct for all other fields. You may also need to specify the location of your Java compiler.

2.0 Maintaining the Server

Once set up, the server should run with little or no intervention. This section will provide an overview of the functions available to administrators, the location of advanced configuration files, and simple trouble-shooting tips.

2.1 Administrative Functions

By logging in as Administrator (the default password is fusionAdmin), you can create and modify accounts. The simple web-based form is relatively straight-forward to follow. A "Student" do not have access to another student's implementations, where as a "Teaching Assistant", "Recitation Instructor", "Professor", or "Administrators" does.

Creating a new Concept Graph allows the administrator to define a new project or course with the system.

2.2 Advanced Configuration Files

Each project has its own directory in `D:\EF\Root\projects`. Within the project-specific directory, the .projectInfo file specifies information about the project such as the name of the course and the course homepage. The datatypes.list file contains a project-specific list of input and output data types available for building modules.

The `ConceptGraph` directory in each project contains the default concept graph that is copied into each user's directory when a user enters the system for the first time. Any project-specific module layout can be written into this directory. Currently, the best way to do this is to create a concept graph, and then copy it from your own `ConceptGraph` directory on the server into the project's `ConceptGraph` directory.

Each project also contains a Users directory. In this directory, the .userdb file contains a set of usernames and passwords, similar to the UNIX `/etc/passwords` file. The Users directory also contains the file authorized. This file contains a list of usernames of students

who are allowed to create new accounts. Within each user's directory is the file .userInfo. This file contains specific information about the user, such the student's name and whether or not the user is a student, TA, or professor. This file is generated by the Administrator's create/modify Web interface, but can be edited directly if necessary.

2.3 Trouble-shooting

This section describes how to fix a couple problems that might come up while running the system. Hopefully these will be few and far between.

If either the registry server or the messaging server dies, you will generally see a failure error box on the server machine and users will not be able to connect. To fix this, enter the Control Panel and find the eFuseRegistry and eFuseMessaging services. Click to "Stop" each service, then click to "Start" each service again. This should fix any problems. No pertinent user state is lost in this process.

Another possible bug is that a user's default concept graph becomes corrupt. In this case, the user may not be able to log in at all because the registry server may disconnect immediately. The client is designed to recover from this and allow the student to load or create another concept graph, but in the event that this is not possible, the administrator should perform the following action. Copy the `Default.cg` file from the project's `ConceptGraph` directory over the `Default.cg` file in the student's directory. This should resolve the problem.

Appendix C

Implementation of a module for Bresenham's algorithm. This module was only designed with the implementation mode enabled.

Definition of styles:

```
Automatically generated code
Code written by module developer
Code written by student
```

```
package projects.graphics.modules;

import edufuse.cg.*;
import java.util.*;
import java.awt.*;
import java.awt.image.*;
import jgl.Queue;

public class LineDraw_24 implements edufuse.cg.FusionListener, edufuse.cg.ModuleBean
{

// Place local member variables here


// SYSTEM VARIABLES DO NOT MODIFY
    Thread thisThread = new Thread(this);

    int numInputs = 1;
    Vector listenerNames = new Vector(1,1);

    int numOutputs = 1;
    Vector sourceNames = new Vector(1,1);

    Queue eventQueue[] = new Queue[numInputs];
    Vector listeners[] = new Vector[numOutputs];
    ImageObserver _imageObserver;
    Image _thisImage;
    int _x, _y, _width, _height;
    Graphics _graphics;

// SYSTEM INITIALIZATION DO NOT MODIFY
    public LineDraw_24()
    {
        for(int i = 0; i < numInputs; i++)
        {
            eventQueue[i] = new Queue();
        }
        for(int i = 0; i < numOutputs; i++)
        {
            listeners[i] = new Vector(3, 5);
        }
        thisThread.setDaemon(true);
```

```
        thisThread.start();

sourceNames.addElement("projects.graphics.modules.ThreeDToken");

    listenerNames.addElement("projects.graphics.modules.ThreeDToken");

    }

    public void killThread()
    {
        thisThread.stop();
    }

    public Vector getSourceNames()
    {
        return (Vector)sourceNames.clone();
    }

    public Vector getListenerNames()
    {
        return (Vector)listenerNames.clone();
    }

    public boolean mapListener(FusionListener listener, int localindex, int remoteindex)
        {
            boolean result = false;
        switch(localindex){
            case 0:
                        if(listener.listensFor(new projects.graphics.modules.ThreeDToken(),
                                               remoteindex))
                        {
                            result = true;
                        }
                        break;

        }
        if(result)
        {
            listeners[localindex].addElement(new FusionListenerPortPair(listener,
                                                                        remoteindex));
            return true;
        }
        else
        {
                return false;
         }
        }

        public boolean unmapListener(FusionListener listener, int index, int remoteindex)
        {
            listeners[index].removeElement(new FusionListenerPortPair(listener,
                                                                    remoteindex));
         return true;
        }

    public void doEventAction(Object obj, int index)
    {
        if(obj == null || listeners[index] == null)
                System.out.println("Something is null here");

         switch(index)
         {
             case 0:
                {
                 projects.graphics.modules.ThreeDToken eventObject =
                           (projects.graphics.modules.ThreeDToken)obj;
                 // for each listener
                 //     call the notify event with the ColorPixelVector data type
                 try{
                       synchronized(listeners[index]){
                     for(Enumeration e = listeners[index].elements();
                         e.hasMoreElements();)
                     {
```

```
                        FusionListenerPortPair f =
                               ((FusionListenerPortPair)e.nextElement());
                        if(f == null) System.out.println("f is null");
                        synchronized(f){
                        synchronized(f.listener)
                        {
                        if(obj == null)
                        {
                               System.out.println("object is null");
                        }
                         f.listener.fusionAction(eventObject.clone(), f.port);
//                      System.out.println("event was fired");

                        }
                      }
                        }}
                }
                catch (Exception e)
                {
//                      System.out.println("I have an exception here: " + e);
                }
               }
               break;

        }
    }


    // report whether or not we support an event
    public boolean listensFor(Object o, int index)
    {
        switch(index)
        {
            case 0:
                if(o instanceof projects.graphics.modules.ThreeDToken)
                   return true;
                else
                    return false;
        }
        return false;
    }

    // handle generic notifications
    public void fusionAction(Object evt, int index)
    {
        boolean valid = false;
        switch(index)
        {
            case 0:
                if(evt instanceof projects.graphics.modules.ThreeDToken)
                {
                    valid = true;
                }
                break;


        }
        if(valid)
        {
            eventQueue[index].push(evt);
               synchronized(thisThread)
               {
                   thisThread.notify();
               }
        }
    }

    public void run()
    {
        while(true)
        {
            boolean end = false;
            while(end != true)
```

```
            {
                end = true;

                if(!eventQueue[0].isEmpty())
                {

Event0Wrapper((projects.graphics.modules.ThreeDToken)eventQueue[0].pop());
                    end = false;
                }

            }
        synchronized(thisThread)
        {
        try{
            thisThread.wait();
        }
        catch(Exception e)
        {
                System.out.println("Exception: " + e);
        }

        }
         }
    }

    public void setImage(Rectangle r, Image aImage, ImageObserver aImageObserver)
    {
        _x = r.x;
        _y = r.y;
        _width = r.width;
        _height = r.height;
        _imageObserver = aImageObserver;
       _thisImage = aImage;
        if(_thisImage != null)
        {
            _graphics = _thisImage.getGraphics();
            paint(_graphics);
        }
    }

    public Image getImage()
    {
        return _thisImage;
    }

    public int getNumOutputs()
    {
        return numOutputs;
    }
    public int getNumInputs()
    {
        return numInputs;
    }

    public void update()
    {
        _imageObserver.imageUpdate(_thisImage, 0, _x, _y, _width, _height);
    }

    public String toString()
    {
        return ("LineDraw_24");
    }



// IMPLEMENTATION -- ENTER YOUR CODE BELOW THIS LINE
// ------------------------------------------------

// Painting
// Call paint(_graphics) if you want your own algorithms to update the canvas

    public void paint(Graphics g)
```

```
    {
        // This implementation of paint automatically performs double-buffereing

        // Call update if you want the module's drawing area to be refreshed
        // update();

        // EXAMPLE (Draws a red rectangle):
        //        Color c = new Color(255, 0, 0);
        //        g.setColor(c);
        //        g.fillRect(2, 2, 14, 14);
        //        update();
    }

// Event handling

    public boolean handleEvent(Event evt)
    {
        if(evt.id == Event.MOUSE_DOWN)
        {
               Fragment f = new Fragment(3,3,0,Color.black);
               Fragment f2 = new Fragment(10,10,0, Color.black);
               Event0Wrapper(new ThreeDToken(ThreeDToken.ENDPOINT_PAIR,
                             new EndpointPair(f, f2)));
               System.out.println("Fired Bresenham");
        }
        if(evt.id == Event.MOUSE_UP)
        {
        }
        if(evt.id == Event.KEY_PRESS)
        {
        }
        return false;
    }


    public void Event0Wrapper(projects.graphics.modules.ThreeDToken e)
    {
        // This gets triggered for the event action of the 0 input port.
        // You should make the appropriate call to an algorithm here for this module.
        // That algorithm should fire an event or update the display or both.

       if(e.getTokenID() == ThreeDToken.ENDPOINT_PAIR)
       {
               EndpointPair pair = (EndpointPair)e.getTokenValue();
               Bresenham(pair.start.x, pair.start.y, pair.end.x, pair.end.y);

       }
       else
       {
               doEventAction(e, 0);
       }
    }

    public void setPixel(int x, int y)
    {
       doEventAction(new ThreeDToken(ThreeDToken.FRAGMENT, new Fragment(x, y, 0.,
                     Color.black)), 0);
    }


    public void Bresenham(int x1, int y1, int x2, int y2)
        {
       // drawLine (int x1, int y1, int x2, int y2);
       // handles all 8 octants, all special cases
       int x, y, dx, dy, npix;
       int xinc, yinc, xup, yup; // advance step; error step
       int e, einc, emax;        // error; err inc; err limit

       dx = x2-x1;  dy = y2-y1;  // x,y deltas and sign bits
       xinc = (dx >= 0) ? 1 : -1;
       yinc = (dy >= 0) ? 1 : -1;
```

```
        if ( (xinc * dx) > (yinc * dy) ) { // loop across x
                  npix = (dx * xinc) + 1;          // note:  non-neg
                  emax = dx * xinc;
                  einc = (dy * yinc) << 1;
                  xup = 0;
                  yup = yinc;
                  yinc = 0; // adv, err steps
             }
        else {                                // loop across y
                  npix = (dy * yinc) + 1;          // note:  non-neg
                  emax = dy * yinc;
                  einc = (dx * xinc) << 1;
                  yup = 0;
                  xup = xinc;
                  xinc = 0; // adv, err steps
             }

        x = x1;
        y = y1;
        e = 0;    // error initially zero
        while ( npix-- > 0 ) {
                  setPixel (x, y);
                  x += xinc;
                  y += yinc;
                  e += einc; // advance step
                  if ( e > emax ) {
                    x += xup;
                    y += yup;
                    e -= emax << 1; // error step
                  }
          }
    }
}
```

Glossary

avatar:	a graphical representation of another eFuse user.
black box:	a system in which only the inputs and the outputs are known, but the implementation is not visible.
CGI script:	a script, generally on a Web server, which implements the Common Gateway Interface for passing data to and from a Web browser.
data-flow component:	a component of a system that accepts data inputs and generates data outputs.
data-flow application:	systems comprised of modules linked together to perform an action.
concept graph:	the workspace in which users can move, resize, and link components while also viewing avatars representing other eFuse users.
design patterns:	common ways of specifying method names and variable names. In eFuse, design patterns allow the system to automatically generate modules based on the name of the module and its input type(s) and output type(s).
difference mode:	the state of a module in which the difference in the output of the algorithm written in the implementation mode and the output of the algorithm in the reference mode is visually displayed.
directed acyclic graph (DAG)	a structure containing directed edges between vertices which does not contain any connected cycles.
eFuse client:	the HTML and Java environment that runs in any Java-enabled Web browser and provides the ability to work with modules, create data-flow systems, and communicate with others.

eFuse server: a standard Web server that has been enhanced through CGI scripts to server the eFuse client to users, compile code, and maintain persistent state information for all the users.

graphics canvas: the space in which a module can draw lines, rectangles, images, etc.

implementation mode: the state of a module in which a student implements an algorithm programmatically.

Java class file: a compiler-generated file of bytecode instructions that perform the actions specified in the Java source code.

Java Virtual Machine: an interpreter that performs the instructions in Java class files, handles multi-threading, access to system resources such as networking, and garbage collection.

link: a connection between the output of a source module and the input of a listener module.

listener module: a module that waits for input from another module.

manual mode: the state of a module in which students attempt to manually solve the implicit problem that the module solves, or attempt to perform the functions of the module on a small number of inputs.

module: the fundamental component of the eFuse system which optionally incorporates the manual, reference, implementation, and difference modes and behaves as a data-flow component. A module can be a source and a listener module.

module designer: a person who designs and implements the basic functionality of a new module for which students will later be asked to provide the complete implementation.

`ModuleBean` *Implementation:* a layer of a module that is unique for each module in the system. `ModuleBean` implementations can be replaced with newer versions dynamically.

package name: the name of a group of similar objects organized into a single location in the Java programming language. (i.e. java.Math is the package name for Java's math objects).

proxy object: a layer for a module that serves as a static reference for other modules in the system and passes requests from other modules to the `ModuleBean` implementation. Allows new `ModuleBean`

	implementations to be loaded into the running system without breaking references.
reference mode:	the state of a module in which a student can view the proper output of a module on different user-specified inputs.
user:	any student, teaching staff member, developer, or module designer using the eFuse systems.
Remote Method Invocation:	specification that allows objects distributed across a network to perform remote procedure calls on the methods in other objects.
Root directory:	the base directory in which all persistent state on the eFuse system is stored.
source module:	a module that produces output and sends it to listener modules.
system developer:	a person who is interested and understands the underlying architecture and implementation of eFuse.
universal resource locator (URL)	an address for an object or document on the network. Generally of the form: protocol://hostname/location/filename.ext

Bibliography

[Ber97] Tim Berners-Lee. "WWW Addressing Overview." *http://www.w3.org/pub/WWW/Addressing/*. World Wide Web Consortium, 1997.

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

[Bro+84] Brown, Marc H. and Robert Sedgewick. "A System for Algorithm Animation." SIGGRAPH Proceedings: Volume 18, Number 3. July 1984. Pages 177 – 186.

[Bro91] Brown, Marc H.. "Zeus: A System for Algorithm Animation and Multi-View Editing." In IEEE Workshop on Visual Languages, pages 4-9, October 1991. Also appeared as SRC Research Report 75. (Postscript).

[Bro+93] Marc H. Brown and Marc Najork. "Algorithm Animation Using 3D Interactive Graphics." In ACM Symposium on User Interface Software and Technology, pages 93-100, November 1993. Also appeared as SRC Research Report 110a, with color pictures. There is an accompanying videotape, Report 110b. (Postscript).

[BGG98] Brown Computer Graphics Group. "Exploratory." *http://www.cs.brown.edu/research/graphics/research/exploratory/exploratory.html* 1998.

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

[Dol97] Dollins, Steven C. "Interactive Illustrations." *http://www.cs.brown.edu/research/graphics/research/illus/*. Brown University Department of Computer Science, 1997.

[FDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Huhes. *Computer Graphics: Principles and Practice.* 2nd ed., Addison-Wesley Publishing Company, 1990.

[Gol+96] Murray W. Goldberg, et. al. "WebCT – World Wide Web Course Tools." *http://homebrew.cs.ubc.ca/webct/*. University of British Columbia Department of Computer Science, 1996.

[Gos+95] Gosling, James and Henry McGilton. "The Java Language Environment: A White Paper." Sun Microsystems. October 1995.

[Hae88] Haeberli, Paul E. "ConMan: A Visual Programming Language for Interactive Graphics." SIGGRAPH Proceedings: July 1988. Pages 103 – 111.

[Har97] Harrison, Timothy, David Levine, and Douglas Schmidt. "The Design and Performance of a Real-time CORBA Event Service." OOPSLA '97. Atlanta: October, 1997. Washington University Department of Computer Science Technical Report #WUCS-97-31.

[Hic97] Hickley, Tim. "Jscheme: an Applet for Teaching Programming Concepts to Non-Majors." *http://www.cs.brandeis.edu/~tim/Packages/Jscheme/Papers/jscheme.html* Brandeis University, Michtom School of Computer Science, 1997.

[IBM97] International Business Machines, Inc. IBM Visualization Data Explorer: Programmer's Reference. *http://www.almaden.ibm.com/dx/docs/html/proguide.htm* September, 1997.

[IBM98] International Business Machines, Inc. IBM Data Explorer (DX). *http://www.almaden.ibm.com/dx/* 1998.

[Law+92] Lawrence, Andrea W., Stasko, John T, and Eileen Kraemer. "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.

[Mor86] J.H. Morris et al., ``Andrew: A Distributed Personal Computing Environment," Comm. ACM, Vol. 29, No. 3, Mar. 1986.

[Sal88] Jerome H. Saltzer. Project Athena Facilities--an Overview for Faculty. M. I. T. Project Athena, 1988. Reprinted in Project Athena: The First Five Years, Volume 1. Digital Equipment Corporation, Hudson, Massachusetts, 1988.

[SM97] Sun Microsystems, Inc. "RMI Documentation." *http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/*. Sun Microsystems, Inc., 1997.

[Sta96] Stasko, John T., " Using Student-Built Algorithm Animations as Learning Aids", Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-GVU-96-19, August 1996.

[Sta+92] Stasko, John T and Eileen Kraemer. "A Methodology for Building Application-Specific Visualizations of Parallel Programs." *ftp://ftp.cc.gatech.edu/pub/gvu/tech-reports/92-10.ps.Z* Technical Report GIT-GVU-92-10. Georgia Institute of Technology College of Computer Science. 1992.

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

[Try97] Samuel Trychin. *Interactive Illustration Design.* Brown University Computer Graphics Lab, 1997.

[Zac+97] Joe Zachary, et. al. "Hamlet Project." *http://www.cs.utah.edu/~hamlet/*. University of Utah Department of Computer Science, 1997.