Graphics Performance Benchmarking Framework

ATI

Presented to: Jerry Howard

By:

Drew Roberts, Nicholas Tower, Jason Underhill

Executive Summary

The goal of this project was to create a graphical benchmarking framework. This framework would allow the user to queue up benchmarks for the framework to run in sequential order. After the benchmarks are run, their outputs would be displayed on the screen and saved for later review. The framework would include functionality to let the user view and modify the settings of the graphics card, thus allowing the user to optimize graphics card performance.

To implement this project we used Tcl/Tk to handle the graphical user interface and created a C++ toolkit that the Tcl/Tk code could call. This toolkit handles the actual application list and queue, runs the applications, stores the output files, and generally does all of the behind-the-scenes work.

All of the functionality except the changing and viewing of the graphics card settings has been implemented. The settings functionality was unable to be implemented due to lack of information, but the framework was designed to allow a future programmer to easily integrate that functionality into the framework. The implemented framework allows the user to add and delete benchmarks from a stored application list, queue up applications and run them, display their output, and view the results of and/or delete previous runs for comparison.

Overall we are pleased with our product. We accomplished everything that was possible to accomplish, and produced something not only functional, but useful.

Table of Contents

Executive Summary 1

Table of Contents 2

I. Abstract 3

II. Introduction 4

III. Requirements 5

IV. Design 8

V. Implementation 12

VI. Conclusion 17

I. Abstract

Performance benchmarks are used by many people in the computer industry, from business administrators to gaming enthusiasts. A user will load up a benchmark and run it to see how well their computer configuration performs, and often base changes to that configuration on the results of the benchmarks. The benchmark lets the user know if something needs to be changed or not, whether the computer is running as well as required or not.

ATI is a graphics hardware company looking to expand their software support to the Linux environment. Currently they have little software support for Linux, but are looking to change that due to the increasing popularity of Linux. To this end, ATI has requested that we design a graphical performance benchmarking framework for Linux. This will require us to design a graphical user interface that allows a user to run benchmarking applications and, based on the results, modify the settings of the graphics card.

The user will be able to add benchmarking applications to a list, select applications from that list and then add them to a queue, run that queue, and then see the results from each application run once all applications have finished running. This framework will store all of the results for later review and allow the user to change the settings of the ATI graphics card, thus allowing the user to optimize performance based on benchmarking applications.

II. Introduction

- **Purpose**

  ATI has requested that our team design a graphical performance benchmarking framework for Linux. ATI, a graphics hardware company, currently has little software support for the Linux operating environment. ATI is looking to change that due to the increasing popularity of Linux. To this end, they want to develop a tool that allows a Linux user to optimize the settings of an ATI graphics card to meet their needs using the results of benchmarking applications.

  The purpose of this paper is to explain the goals, requirements, design, and implementation of a graphical performance benchmarking framework that meets the needs of ATI and their customers who use Linux.

- **Goal**

  The goal of this project is to create a graphical performance benchmarking framework that will allow a user on Linux to queue up and run any number of benchmarking applications, review those (and past) results, and, from that information, modify the settings of their ATI graphics card for optimal performance.

III. Requirements

- **Requirements Overview**

  The framework can be broken up into three separate areas, each with its own set of requirements. The general requirements for these are as follows:

- The benchmark application handling needs to include a user-maintained list of available applications and a run queue for queuing up and running the applications.
- The application output handling needs to include fetching and displaying the application output to the user when the application is finished and keep a record of the benchmark outputs.
- The graphics card settings need to be easily modifiable by the user.

The framework must be developed on Red Hat Linux. The framework must be easily portable between Linux and Windows. The framework must be user-friendly, simple and have a graphical user interface, and its users will have high levels of computer expertise. If time permits, the framework could be able to display the output in the form of graphs and could come with an example benchmarking application to demonstrate how the framework functions. The framework could also have a nice color scheme and precise window management. In order to ensure we meet all the requirements in order of necessity, we have arranged them on basis of functionality into primary and secondary requirements.

- **Functional Requirements**

Primary Requirements

- There must be a user-maintained benchmark application list
  - User inputs program name
  - User inputs program location
  - User inputs output location
  - User can add to list
  - User can delete from list
- There must be a run queue for benchmark applications

    - User can add to queue from application list
    - User can delete from queue
    - Application at the top of the queue runs first
- The application output must be displayed to the user
    - When application is finished, framework must find output.
    - Output is displayed in a user-friendly manner
- The application output must be recorded
    - All output must be stored
    - All output must be easily available for later review
- The graphics card settings must be user-modifiable
    - All graphics card information must be found
    - All relevant graphics card information must be displayed to the user
    - All modifiable settings must be displayed to the user
    - Any modifications the user makes must take immediate effect
- The framework must have a graphical user interface
    - Must be user-friendly

Secondary Requirements

- The framework could display the output in the form of a graph
    - Output parsed into a graphing package
    - Graph displayed to user
- The framework could come with a test benchmarking application
    - Create SPECviewperf suite

- **Nonfunctional Requirements**

Primary Requirements

- The framework must be developed on Red Hat Linux
- The framework must be easily portable between Linux and Windows
    - The framework should use Tcl/Tk to create the graphical user interface

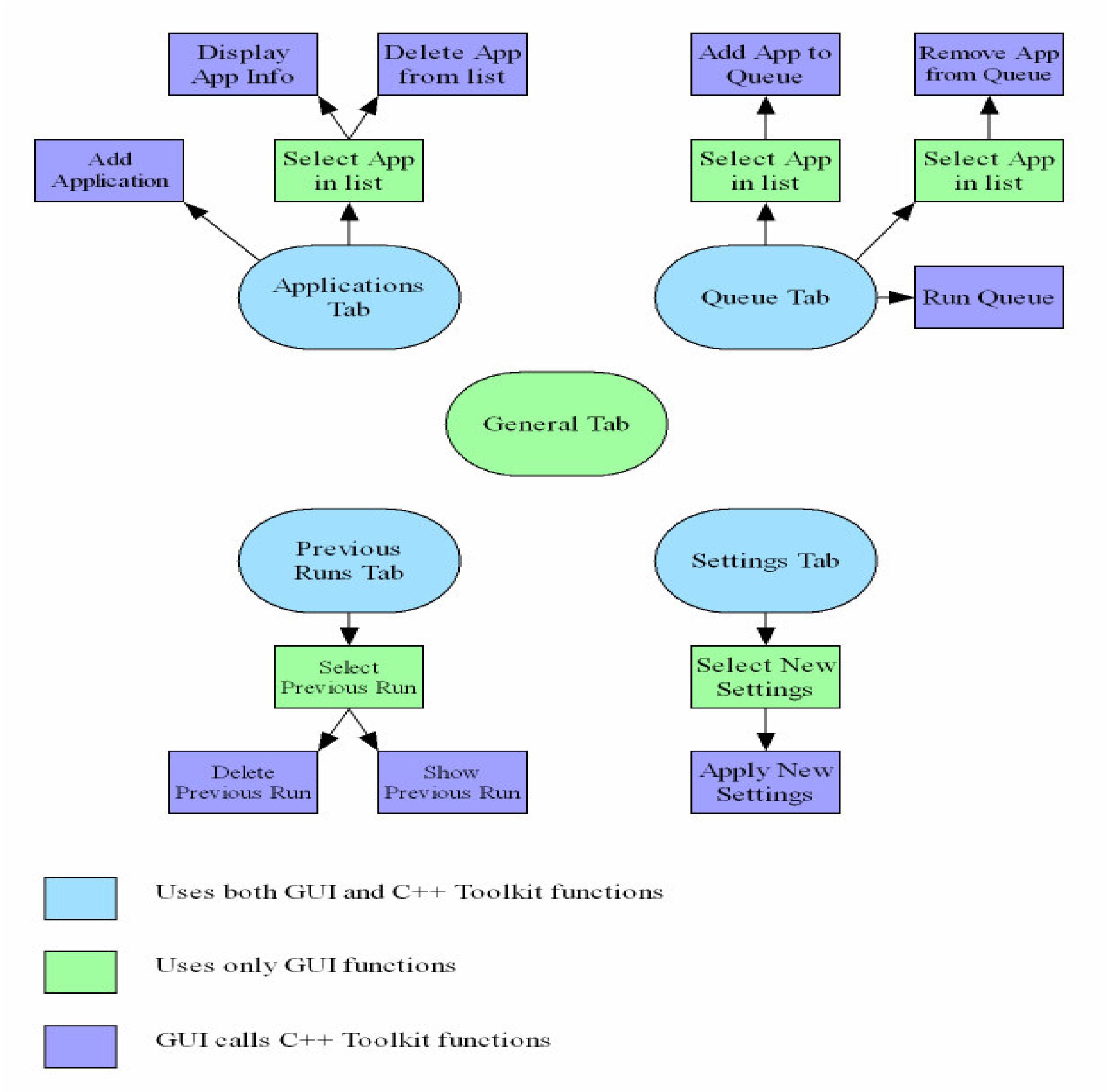
Secondary Requirements

- The framework could look good
    - Color scheme, arrangement, window-creation, etc
- The frame could use the Ant compiler

IV. Design

- **Design Overview**

Due to the complex nature of our project, we decided to split it up into the four components that it naturally divided into: managing the application list, managing and running the program queue, storing and displaying the previous runs, and modifying the settings of the graphics card. Each component has its own flow and use cases, which have been summarized by the diagram below. The user interface was designed with this division in mind, and thus each component has its own tab, along with a 'General' tab which is mostly for aesthetics.



- **Design Components**

Applications

The user must be able to both add and delete applications from the stored list, and it would be helpful if the user could see what was entered. Therefore, our design prominently displays the list of the applications the user has entered displayed, with the ability to delete and/or show the info of a current selection. Adding a new application requires information from the user; our program requires the user to input the location of the application executable and the location of the output file that the application will generate. The user is also required to give the application a description as an identfier.

Queue

The user must be able to add and delete applications to the program queue from the application list. Therefore, our design shows the application list to the user and has them select an application in that list, then copy it into the queue. For deleting, the user selects a program in the queue and deletes it. The user also must be able to run the queue; our design has a 'Run Queue' button that does exactly that, with a warning mentioning that running the queue will probably take a long time.

Previous runs

The user must be able to view the results of previous runs which have been stored. Therefore, our design copies the output of each application into a folder, then displays the files in a list for the user to select from. The user selects a previous run and has two options: display or delete. Delete will delete the file permanently as well as removing the entry from the list. Display will show the user the output of the selected run.

Settings

The user must be able to modify the settings of the graphics card. Since details are largely unknown, our design is rather vague. The options will be presented to the user and the user will be allowed to modify and apply settings as they choose.

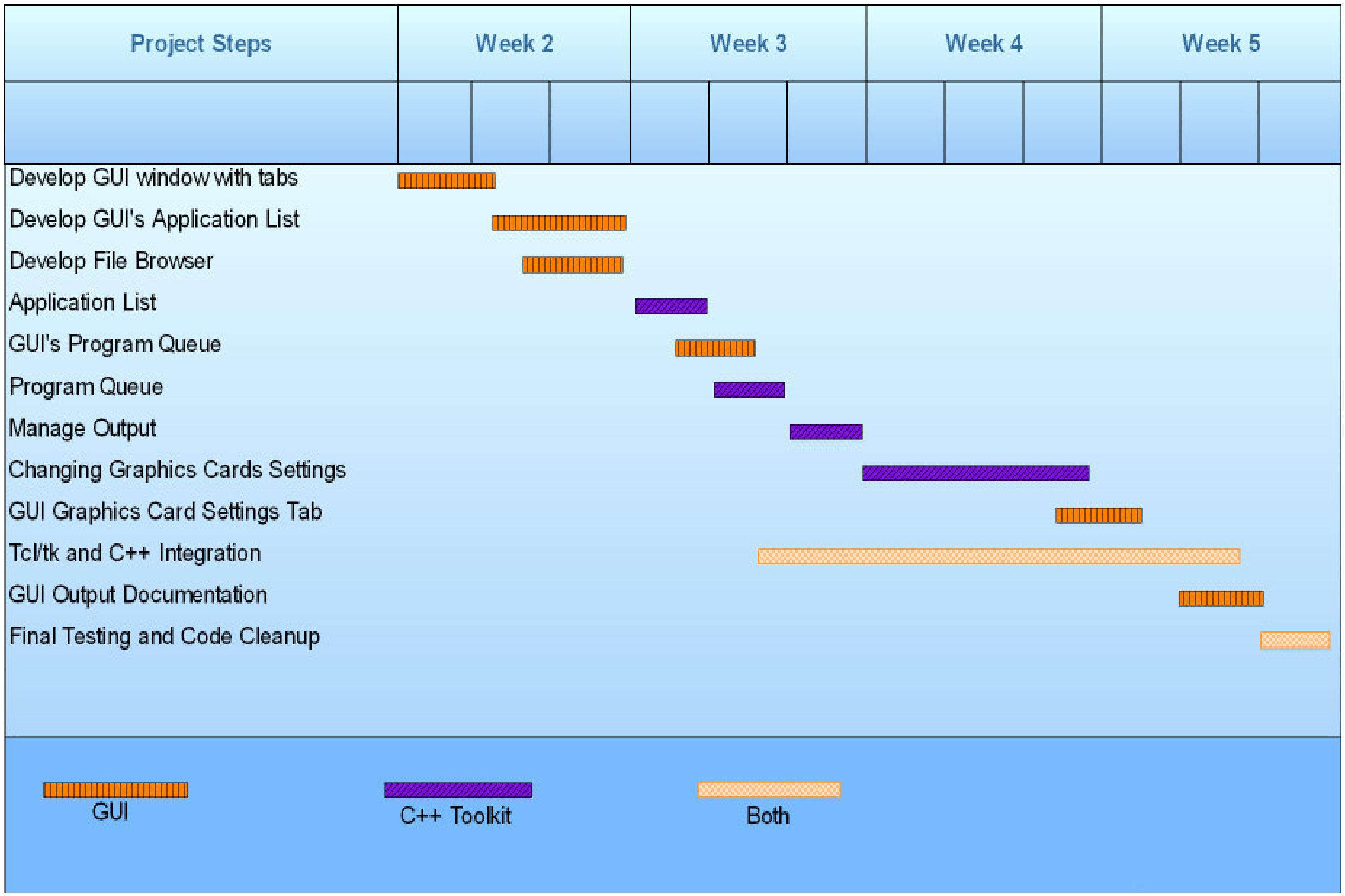
- **Implementation Design**

  Due to the desire for easy portability, it was suggested we use Tcl/Tk to implement the graphical user interface. However, Tcl/Tk is not well-suited to maintaining lists and running programs. We decided to create a toolkit, written in C++, which would take care of many of the list-handling, file storage, and program-executing tasks. Tcl/Tk is also not well-suited to make the system calls we thought necessary to modifying the settings of the graphics card, thus we decided the C++ toolkit would take care of that as well.

- **Schedule**

  A schedule was created showing how long each task is expected to take. Below is a chart displaying each task and how long we anticipate each task will take to complete.

Project Development Schedule



V. Implementation

- **Implementation Overview**

  The implementation of this project can be broken up into two major sections: the user interface, and the underlying toolkit. The user interface was coded using Tcl/Tk with the Iwidgets package, the underlying toolkit was coded using C++, and the entire project was developed on Red Hat Linux. The user interface and the underlying toolkit were integrated using SWIG. The application runs on Linux, but is easily portable to most other operating systems, especially Windows.

  The user interface allows the user to communicate with the toolkit, which is actually doing the work. The toolkit provides and manages a stored application list and implements a scheduled queue. The toolkit runs that queue on command and copies the resulting output file to a folder of stored previous runs so that it can be accessed later.

- **User Interface Implementation**

  The user interface was implemented using Tcl/Tk. This language was chosen due to its portability and feasibility, as it is both platform-independent and fairly simple. The Iwidgets package was used to provide the tabbed-window interface, as well as the various popup windows.

  The user interface was designed in two stages. The first stage involved designing and building a framework to attach the toolkit to. The selected design was a tabbed window with four tabs, each with its own purpose. The 'General' tab provides a brief overview of the program's status. The 'Applications' tab manages the list of applications. The 'Queue' tab manages the queue and allows the user to run the queue. The 'Previous Runs' tab contains a list of previous executions of applications for which output is stored. It contains a selectable list and a button to display the stored output file for the selected run. The output from each application is displayed in a popup window after the queue of applications has run. The

Iwidgets package provides a ‘tabnotebook’ widget and a ‘dialogshell’ widget, which provide the windows where the rest of the Tcl/Tk code is. Iwidgets also provides the output reading. The Tcl/Tk code framework (without any toolkit integration) was tested on both Linux and Windows platforms, and performed equally well on both.

The second stage involved integrating the toolkit into the Tcl/Tk interface. The toolkit was coded in C++, leaving two choices: write wrapping code around the C++ functions, or find a package that would automate the process for us. As writing code around C++ functions to allow Tcl to use them would have taken too much time to learn, we instead used the SWIG package. SWIG is an interface compiler that connects programs written in C and C++ with scripting languages such as Perl, Python, Ruby, and Tcl. Our implementation uses it to construct a scripting language extension module, as C++ is more suited to list-handling and system calls than Tcl. With SWIG, we created a shared object of wrapped C++ code that was loaded into and called from the Tcl/Tk user interface.

The biggest challenge in implementing the user interface was correctly creating the shared object. To create a shared object of wrapped C++ code, SWIG requires a correctly-formatted interface file written in pseudo-C++. SWIG must then be run on the interface file (in the correct mode, in this case –tcl –c++) converting it into a wrapped cxx file. That file and the original cpp must then be compiled into a shared object which will only be functional if the cxx and cpp files are synchronized and the interface file was written in the correct format. This is also where much of the porting effort would be required, as creating shared objects is done differently in each operating system.

- **Toolkit Implementation**

  The toolkit is a group of functions written in C++ which Tcl/Tk can call as needed. We selected to code in C++ because of its capabilities for both list handling and file handling, its known ability to interface with Tcl/Tk, and our familiarity with the language.

Development of the toolkit had two stages. In the first, we developed the underlying data structures. The code maintains both a list of applications as well as a queue describing which of those applications to run. Both of these are implemented using the Vector class. In this stage we created functions for maintaining the list and queue, for running the queue, and for initialization and cleanup of our data. We created code which saves the application list to disk and which can load the same list on startup. Here are some details of what we created:

- Program_struct

  We created a structure which defines the description of a program. This contains three string elements, one for the program's location, one for the location of the program's output, and one which is a name describing the program. Using this structure lets us group the relevant information of each application.

- The List

  The list is a collection of all the applications which the user has entered. The information in this list can be saved to disk and restored for the next time the program is run. The list is implemented with the standard template library's Vector class. The Vector class is a template class and allows us to dynamically change the size and contents of the list and has functionality for adding and deleting entries.

- The Queue

  The queue stores an ordered list of which applications are to be run. Every application in the queue also exists in the application list, but depending on which applications are selected, the two lists will not be the same. Only when the user selects the "Run Queue" button does the queue get filled with information. Up until that point, the user can add and delete to the queue maintained by the user interface. When the run command is executed, the queue in the C++ code fills with the list of which applications to run. The C++ code then steps through each application in the queue and runs it. While an application is running, there can be no interaction with the framework.

- Saving Output

  After each application in the queue runs, the resulting output file is copied into the subdirectory "PreviousRuns". The files are saved with a new filename indicating the time they ran and what application created them. A portion of the C++ code is responsible for checking this directory to determine what data is available for previous runs. This data allows the GUI to display a list of which files are available.

  In the second stage, we created accessors and modifiers which the Tcl/Tk code could use. The creation of these functions was largely driven by what the GUI needed to do. As we added more things to the interface, we added more functions which would allow the Tcl/Tk code to do what it needed. We created functions for adding to and deleting from the queue and list and for returning how many elements are in each.

- List Modifiers

  These functions allow the Tcl/Tk code to modify and manage the list of programs. Programs can be added and deleted from the list with the use of these modifiers, and that list can be saved and loaded from a file.

- List Accessors

  These functions allow the Tcl/Tk code to retrieve information about the contents of the list so that the GUI can correctly manage the list through the use of the modifiers. With these functions the GUI can get the name, file path, and the output for any entry that is in the list.

- Queue Modifiers

  These functions work just like the modifiers for the list with some slight differences. There is no 'delete'; instead there is a clear function which clears the whole queue of all its entries. Also, there is no ability to save or load the queue from a file.

- Queue Accessors

  There are only two accessors for the queue. There is one that returns the program name and there is another that returns the location for the output of a program. These functions exist so that the GUI can find the output files for programs and display them.

Originally, the most challenging portion of our code was the portion which handles the output from the applications. Our initial plan was to have the C++ code read from the output file and return its contents as a string, which could then be displayed in a scrolling text box window. This code would perform error checking to confirm that the file existed and report errors to the Tcl/Tk code as required. During our coding of this portion, we discovered a widget in Iwidgets which performs the entire process for us, loading text from a file and displaying it without requiring any help from C++.

One part of the specification for the project was that the user be able to modify the settings of the graphics card. Unfortunately, this portion of the project was unable to be completed. To implement the settings, example code was needed so that we would know how to talk to the graphics card. Our client gave us a collection of code which we hoped would contain the functions we need to query the graphics card and make changes to its performance oriented settings. After looking through the code we were given, we were unable to find suitable functions to accomplish this. In addition to looking at the code, we also attempted to find this information online. Unfortunately, we were not able to find the methods we needed within the time available, and it is for this reason that this portion of the project is incomplete.

VI. Conclusion

- **Final Results and Testing**

  We have successfully completed a basic version of the framework. However, there is still room for expansion of the project in the future, so we have done our best to leave areas in the code for later developers to add to what we have done.

  The framework currently has the following things implemented. It allows the user to add to and delete from a list of benchmark applications. This list is preserved between sessions, so when the user re-opens the program, the same list will be available. The framework allows the user to queue up a selection of applications from the list and execute them sequentially. The framework then displays the output of each benchmark in the queue and saves a copy of that file for later review. The framework allows the user to review the previous output files, arranged by date, and display them to the screen.

  Testing of our framework consisted of running the applications and entering different things to see how it responded. For example, we entered sample applications which had invalid path names, or ones which specified a non-existant output file. We used each of the buttons in our interface to make sure no unusual behavior happened when the user tried doing things in non-intuitive orders.

- **Future Plans**

  The most significant thing which is missing from the current framework is the ability to view and change the settings on the graphics card. This would be an excellent thing to add to the application to make it an effective tool for calibrating a computer's graphics card under Linux. The user interface has a tab for the settings, but no code currently exists for this portion of the project. With the addition of this feature, it would be beneficial to change the way that output files are archived in order to include information on what the settings were at the time it was run.

There are many other small things which could be done to improve the framework. One possibility would be to have a way for the user to customize their view of the previous settings. This could include different sorting options, an interactive cell-style system, or any of a number of other options.

- **Knowledge Gained**

  The knowledge of Tcl/Tk that we gained will be valuable any time we want a cross-platform GUI in the future. The increased knowledge of how to use Linux and how Linux works will be valuable as Linux is becoming more and more popular. The knowledge of using C++ to make command-line calls will be valuable in any C++ project. Possibly the most useful knowledge gained was the experience of creating a large, multi-module program. This includes a great many minor things such as ensuring that everything is synchronized, that there are no conflicting variable names, checking that everyone has the most recent code, etc. That is the knowledge that will be applied to every project we work on in the future, and thus we believe it will be the most valuable.