FIREFLY - WEB-BASED INTERACTIVE TOOL FOR THE VISUALIZATION AND VALIDATION OF IMAGE PROCESSING ALGORITHMS

A Thesis

presented to

the Faculty of the Graduate School

University of Missouri

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

DANIEL BEARD

Dr. Kannappan Palaniappan, Advisor

December 2009

©Copyright by Daniel Beard 2009
All Rights Reserved

The undersigned, appointed by the Dean of the Graduate School, have examined the thesis entitled

FIREFLY - WEB-BASED INTERACTIVE TOOL FOR THE VISUALIZATION AND VALIDATION OF IMAGE PROCESSING ALGORITHMS

presented by Daniel Beard,

a candidate for the degree of Master of Science and hereby certify that in their opinion it is worthy of acceptance.

Dr. Kannappan Palaniappan

Dr. Yi Shang

Dr. Guilherme DeSouza

Acknowledgments

First and foremost, I would like to thank all of the professors and peers at the University of Missouri who have taught me so much. In particular, I would like to thank Dr. Kannappan Palaniappan as my advisor for his continued patience and guidance in both my undergraduate and graduate studies at the University. I would also very much like to thank my wife, Carolina, who has been extremely supportive of me working late nights to accomplish what seemed like impossible tasks at times. And of course, my little Sophie, who has proven to be one of the best, and most fun, distractions for me. Thanks also to Dr. Yi Shang and Dr. Guilherme DeSouza who have also taught me much and have agreed to review my masters thesis.

Contents

Acknowledgments **ii**

List of Figures **vi**

List of Tables **viii**

Abstract **ix**

1 Introduction **1**

2 Background **3**

2.1 Need for Firefly 3

2.2 Existing Tools 3

2.3 Dataset Types 9

3 Choice of Technology: Rich Internet Application **11**

3.1 Requirements 11

3.2 History and Background 14

3.3 Adoption and use as a RIA technology 15

3.4 Flex Framework Overview 15

3.5 The Elastic Racetrack and the Component Lifecycle 17

3.6 Frameworks for a Framework 21

3.7 AMF and Communication with the Server . . . 24

4 Ground Truth Utility **25**

4.1 Requirements . . . 25

4.2 Implementation . . . 27

5 Firefly **31**

5.1 Overview . . . 31

5.2 Implementation . . . 33

5.3 Implementation Hurdles . . . 35

6 Conclusion **43**

6.1 Future Work . . . 43

6.2 Summary . . . 45

A Computational Provenance **47**

A.1 AVA - Advanced Video Archive . . . 47

A.2 Computational Provenance . . . 49

A.3 Provenance Tools . . . 49

B Entity Relationship Diagram **51**

C Firefly User Manual **53**

C.1 Navigation . . . 53

C.2 Workspace . . . 55

C.3 Accessing GTU and Firefly . . . 62

Bibliography **63**

List of Figures

2.1 ViPER-GT Interface 4

2.2 NeuronJ Interface Interface 6

2.3 NeuronJ Attribute Window 7

2.4 Assigning Attributes to Cells in DCellIQ [4] 8

2.5 Cell classification in DCellIQui [4] 9

3.1 Sample MXML markup 16

3.2 The Elastic Racetrack 18

3.3 Stretched Elastic Racetrack 18

3.4 Flex Component Lifecycle 20

3.5 Cairngorm Event Flow 23

4.1 Sample image from the HeLa dataset 26

4.2 GTU Interface 27

4.3 Assignable Attributes in GTU 28

4.4 GTU System Structure 29

5.1 Organization of the Firefly System 32

5.2 The Firefly System 34

5.3 Sample Firefly Interface 34

5.4 Attributes stored with a Marked Object 35
5.5 Classification Layers 36
5.6 State Diagram of the Firefly System 36
5.7 Firefly Class Diagram 38
5.8 Sample workspace code in Firefly 39

A.1 The tree structure showing the AVA processing chain 48
A.2 Comparing parameters in Vistrails [17] 50
A.3 Data-flow Visualization in Taverna [18] 51

B.1 Firefly Database ERD 52

C.1 Login Page 54
C.2 Choosing a Project 54
C.3 List of possible datasets 55
C.4 Workspace view of Firefly 56
C.5 Frame control widget 56
C.6 Available Tools 57
C.7 Save Control 58
C.8 Firefly Classes 59
C.9 Line Drawing 60
C.10 Box Drawing 60
C.11 Free-Drawing 60
C.12 Attribute Window 61

List of Tables

2.1 Different datasets needed for Firefly 10

3.1 RIA Comparison Parameters 13

4.1 Required classes for the HeLa dataset 26

A.1 Attributes stored in AVA system 48

C.1 Firefly keyboard shortcuts 62

Abstract

Image analysis, in computer science, is defined as the process of extracting useful data from digital images for the purpose of accomplishing a goal. Examples of this might include tracking objects in satellite imagery or finding edges of a vessel network in a microscopic image. Doing this manually can prove to be very time consuming, and is prone to human error. Many advanced techniques and algorithms exist to help automate this process for researchers in the field. By comparing results of an image analysis with ground truth, researchers are able to determine how accurate their algorithms are. Currently there are very few tools to help researchers in gathering and analyzing this ground truth.

We propose a generic, expandable, web based tool called Firefly to help researchers in establishing and visualizing ground truth for differing datasets, and automating the analysis of these datasets. This is done using the interactive and multimedia features that Flash Player provides running in the browser, and using a centrally located database for the storage of the data. The overall goal is to develop web-based services for the analysis of video datasets including data management and image processing of large timeseries datasets with web access to all of the original and processed imagery and results. Firefly is one component of this goal.

Chapter 1

Introduction

Image analysis, in computer science, is defined as the process of extracting useful data from digital images for the purpose of accomplishing a goal. Examples of this might include tracking objects in satellite imagery or finding edges of a vessel network in a microscopic image. Doing this manually can prove to be very time consuming, and is prone to human error. Many advanced techniques and algorithms exist to help automate this process for researchers in the field. By comparing results of an image analysis with the reality of the image, researchers are able to determine how accurate the analysis is. This also allows for researchers to fine tune and enhance the technique to gather even more accurate data. This reality is more commonly known as ground truth for the image being studied.

Traditionally, ground truth would be used in the context of aerial or satellite imagery. While the images are being captured from a distance, there would be individuals on the ground - on location - gathering data. This data is assumed to be accurate, and can then be compared against the captured imagery for verification purposes. Image analysis, however, is not limited to satellite/aerial imagery - it has applications in many fields. For our purposes, we will refer to ground truth as any sort of data that can be gathered and trusted to be accurate for use in validating results generated from a computational analysis of a dataset. Currently, there are no generic tools or systems to help scientists validate their research. In fact, most ground truth is done manually, making it an extremely slow and time consuming process. Any

automatic tools made for measuring ground truth, has been very domain specific.

In this paper, a new system is presented to help researchers establish ground truth for different datasets - most notably in the field of image analysis. This system, Firefly, is centered around the idea of displaying the gathered data visually, and providing the tools necessary for ordinary users to be able to assist in the process of obtaining ground truth for the data. Typically, ground truth in these datasets refers to some type of measured object and it's location relative to the dataset (e.g. points, lines, boxes, polygons) Currently, Firefly is able to display and annotate ground truth for two different types of data. The first type is referred to as a "frameset" in this discussion. A frameset is a set of a finite number of images that go in a particular temporal order. This can be thought of almost as series of frames in a movie. The second data type is includes high-resolution, tiled images. This data type is typically used for satellite imagery. We will use the specific file type made for use with the desktop application, "Kolam". More information on the Kolam file format can be found in [1]

In recent years, web applications have become increasingly more popular and important due to the many advantages that they offer over traditional desktop applications. Most notably, this includes platform independence of the application, and no required software to actually be installed on the user's machine. For these and reasons to be discussed, Firefly was written as a web application. It utilizes relatively recent advancements in the field of rich internet applications, namely Adobe Flex, to provide the level of interactivity needed. Since the datasets are typically very large, Firefly is also able to stream the data needed to the client. We will look into the technology powering Firefly with more people, and also see how the Firefly system is being used now.

Chapter 2

Background

2.1 Need for Firefly

To better explain why such a system would be useful, we can look at the example of HeLa cells, a line of cells used in research by scientists to better help them understand the growth of cancer. A frameset has been taken of some HeLa cells as they change over time. From this frameset, an automated algorithm has been created to determine regions of the cells, as well as classify the cells by their cellular cycle state. In order to validate the results of this algorithm, researchers would need a professional in the field. This person would classify the cells by marking and recording locations where each is located, and then associate it with a state or class. Now consider that there are approximately 10-20 cells in a particular frame, and around 200 frames in the frameset. It becomes clear how difficult and time consuming the task becomes. Doing this manually, using traditional tools would be tedious and prone to human error.

2.2 Existing Tools

Typically, general purpose image editing tools are used to help researchers gather ground truth in image analysis, At times, a specialized tool is created for the express purpose of gathering ground truth for the specific dataset being analyzed, also.

2.2.1 ViPER

One tool that exists for helping researchers in video analysis was created by the Language and Media Processing Lab at the University of Maryland. This tool is known as the Video Performance Evaluation Resource, or ViPER.[2] This lab was involved with analyzing video for semantic content such as detecting text and tracking people. The lab saw the merit in being able to compare their automated results with human generated ground truth, and wanted to create a tool to help them in doing this. The ViPER tool is specific to the domain of video analysis, but exhibits many useful features, and can be looked at as a model for an even more generic solution.

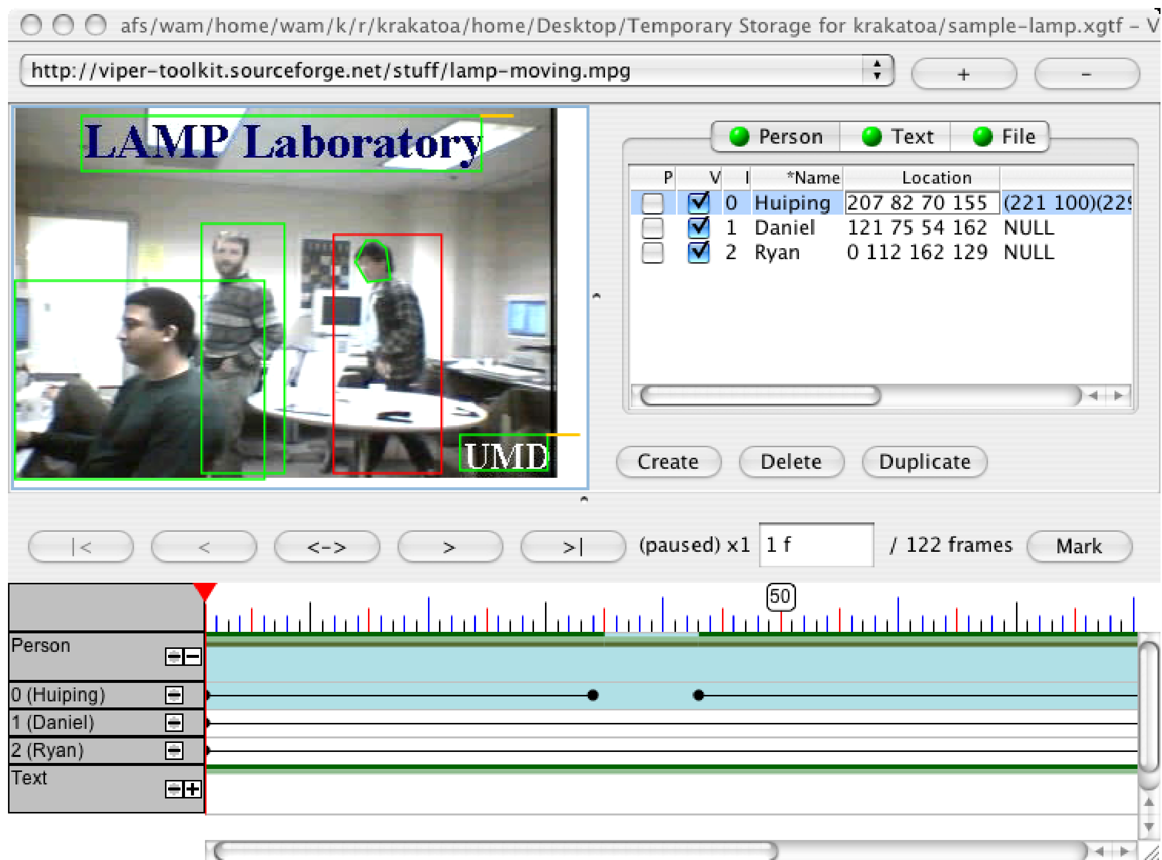

Figure 2.1: Interface of the ViPER-GT - showing regions being marked [2]

In its annotation of the ground truth, ViPER is able to define regions of various shapes, such as rectangles and ellipses. It can also define items that do not necessarily

have a visual element, such as text. This information is stored in an XML file for each video that is annotated. This XML file can then be used as needed by the researchers to compare against their results. A set of rudimentary tools is given to help them in doing this.

There are several aspects of this tool that are important to note. Firstly, it uses a rich set of GUI tools to help the user. The user is essentially able to draw the regions on the image, and easily view their results. Making the tool as interactive as possible is important to help ease the learning curve. Many times, people who will be establishing ground truth for a dataset aren't the researchers who have developed the analysis techniques, but rather professionals in the field who need a tool that allows them to annotate quickly and easily. ViPER also stores it's data in a well documented XML format. This allows for any program to interface with the data for it's own comparison. Researchers could also potentially store their output in the same format, thereby using the ViPER system as a way of displaying their own results as well.

Perhaps the biggest limitation of the ViPER system is the fact that it can only display and mark ground truth annotations for video files. While ViPER was never intended to handle any other file type, there is a need for more generic solutions. ViPER is also a desktop application that forces it's users to install local software, but also have access to the video files as well. When working in a more distributed environment, this can be an issue, especially when dealing with larger images.

2.2.2 NeuronJ

NeuronJ [3] is a tool used to help researchers in obtaining ground truth of elongated structures in images - particularly neurites in fluorescence microscopy images. This tool was created by Erik Meijering of the University Medical Center Rotterdam as part of the Biomedical Imaging Group Rotterdam's research. It was written as a plugin for the Java image processing program OpenJ. This tool, like ViPER, was not

made for use in a collaborative environment. All data is stored in memory, and can be exported at any time to various pre-defined data formats.

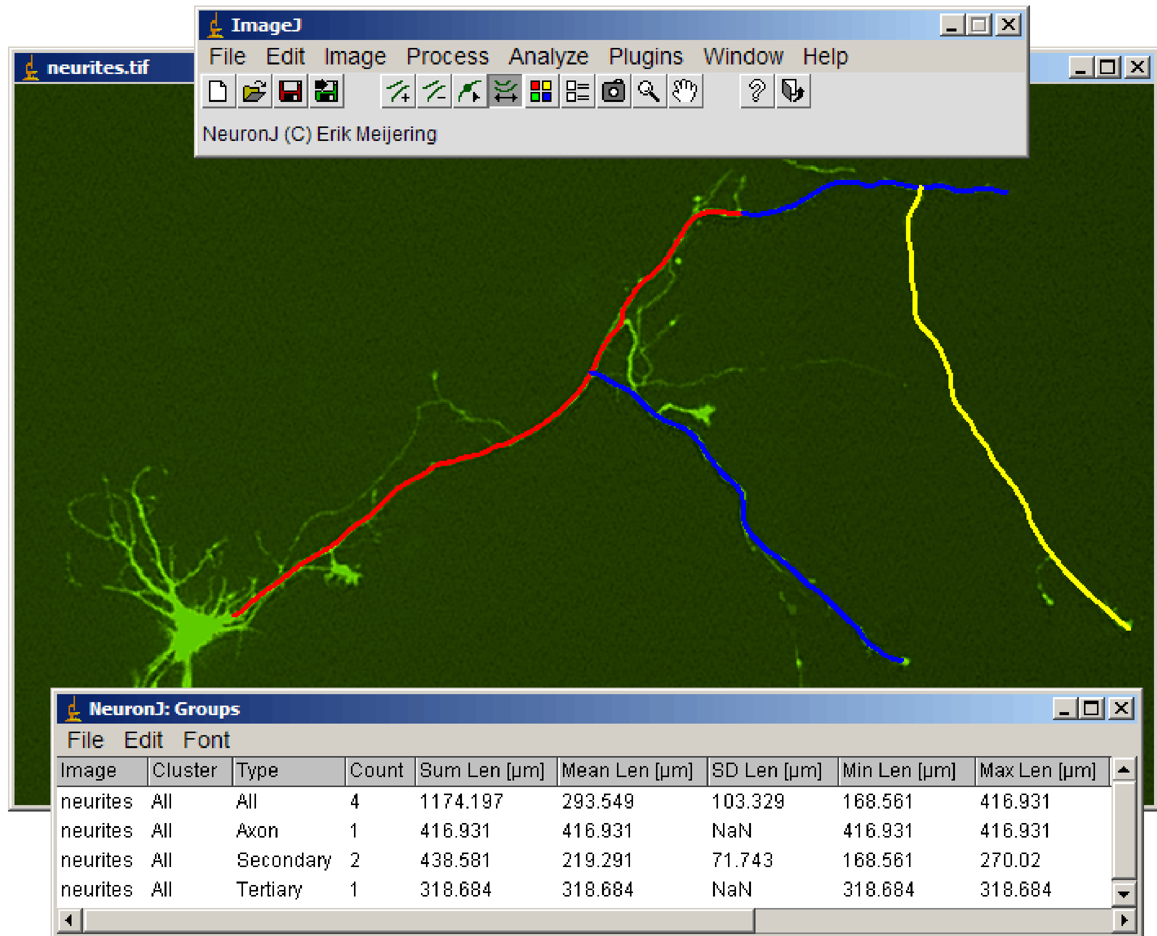

Figure 2.2: NeuronJ Interface - Able to trace neurite structures in images [3]

It is also somewhat limited in it's scope. NeuronJ only allows you to do tracings of images, so all data is stored as points along these contours. Other types of annotations are not possible. It also is limited to working with one image at a time. Users of the system are unable to work with image sets, or any sort of temporal information in the software.

NeuronJ does, however, contain many features that make it powerful in it's scope. Tracing works as expected in any image processing program, but attempts to help the user by correcting the tracing to what it thinks is a more optimal path, by actually analyzing the image below it. Because of the complexity of the neurites, and using

limited means of input to the computer (computer mouse), this is a very important feature that can greatly speed up the work of the researchers. Of course, manual corrections can be made at any time if the correction algorithm did not work as expected. NeuronJ also allows the user to annotate each individual tracing with various predefined attributes. These attributes also include the idea of a class - which allows users to correlate different tracings in the same group. Currently the system allows for ten different classes, each with an assignable color. This class color is reflected in the GUI, making it easier for the user to see where the different classes are in the image.

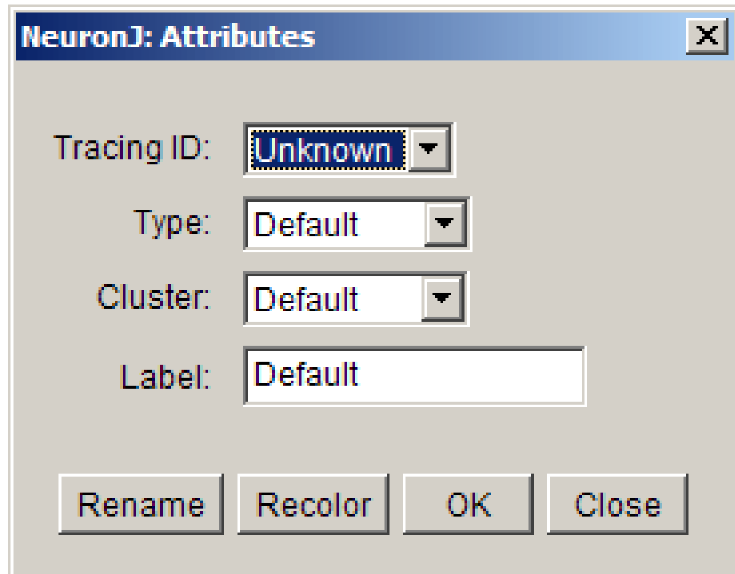

Figure 2.3: Assignable attributes of the NeuronJ system [3]

2.2.3 DCellIQ

DCellIQ (Dynamic Cell Image Quantitator) [4] is a software package created by the Center for Bioengineering and Informatics at The Methodist Hospital Research Institute. DCellIQ was created as a tool to "provide an automated pipeline for quantitative, reproducible and accurate interpretation of cell dynamic behaviors using time-lapse cellular images". The time-lapse cellular imagery is put through a pipeline

of processes that includes detection, segmentation, tracking, feature extraction and classification. This is done in a very straightforward manner, where the user merely needs to load the raw data in the program. All data extracted is stored locally in the same location as the raw data.

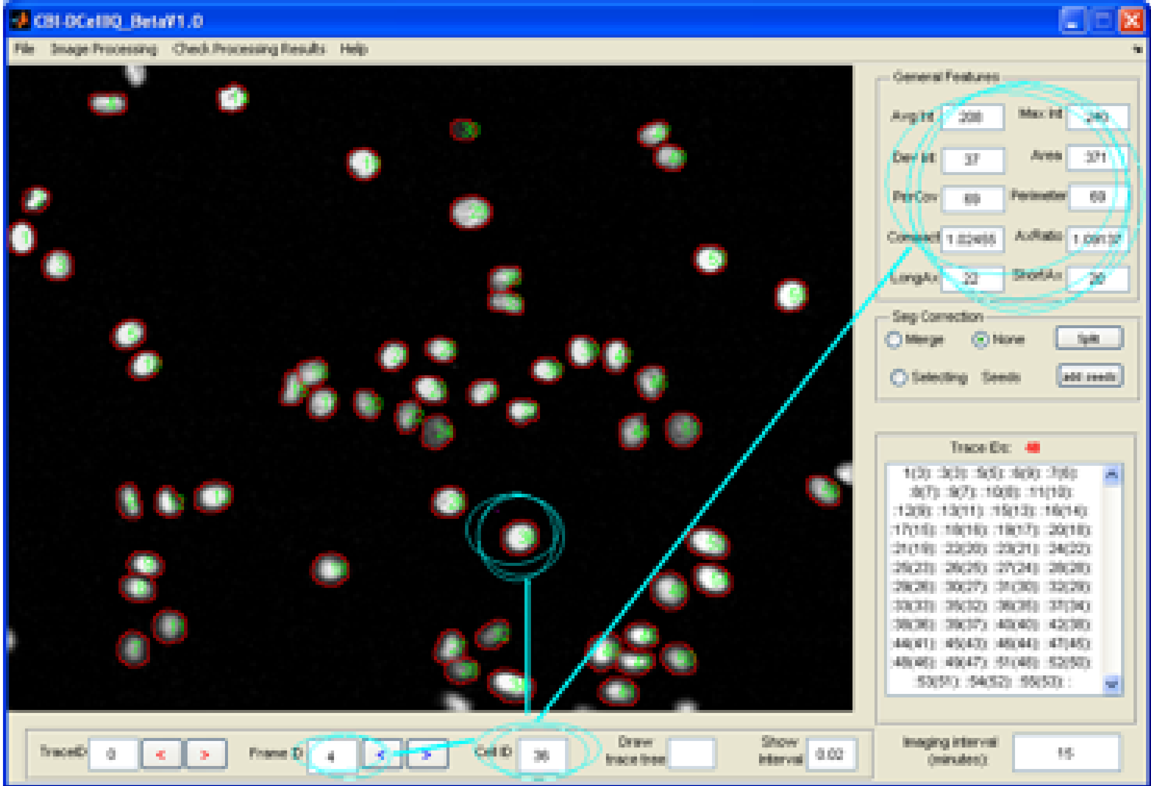

Figure 2.4: Assigning Attributes to Cells in DCellIQ [4]

After the pipeline is done, users are able to gather ground truth, and also correct errors made by the algorithms. Each cell in the system is assigned a certain amount of attributes in the system, and also introduces a way to quantify the cell cycle, or parent-child relationships between the cells. All cells are assigned an ID, that can then be tracked across the different frames of the time-lapse data. Since cells split over time, this is an important feature, and something that needs to be tracked. The user is also able to manually fix this data, by either merging or splitting cells that were auto-detected by the system.

The software package also includes DCellQui, which allows researchers to validate

and correct classification results. This is similar to how NeuronJ handled classes, where a set of predefined classes are able to be assigned to the different cells, and for feedback to the user, they are assigned a color when viewed by the user.

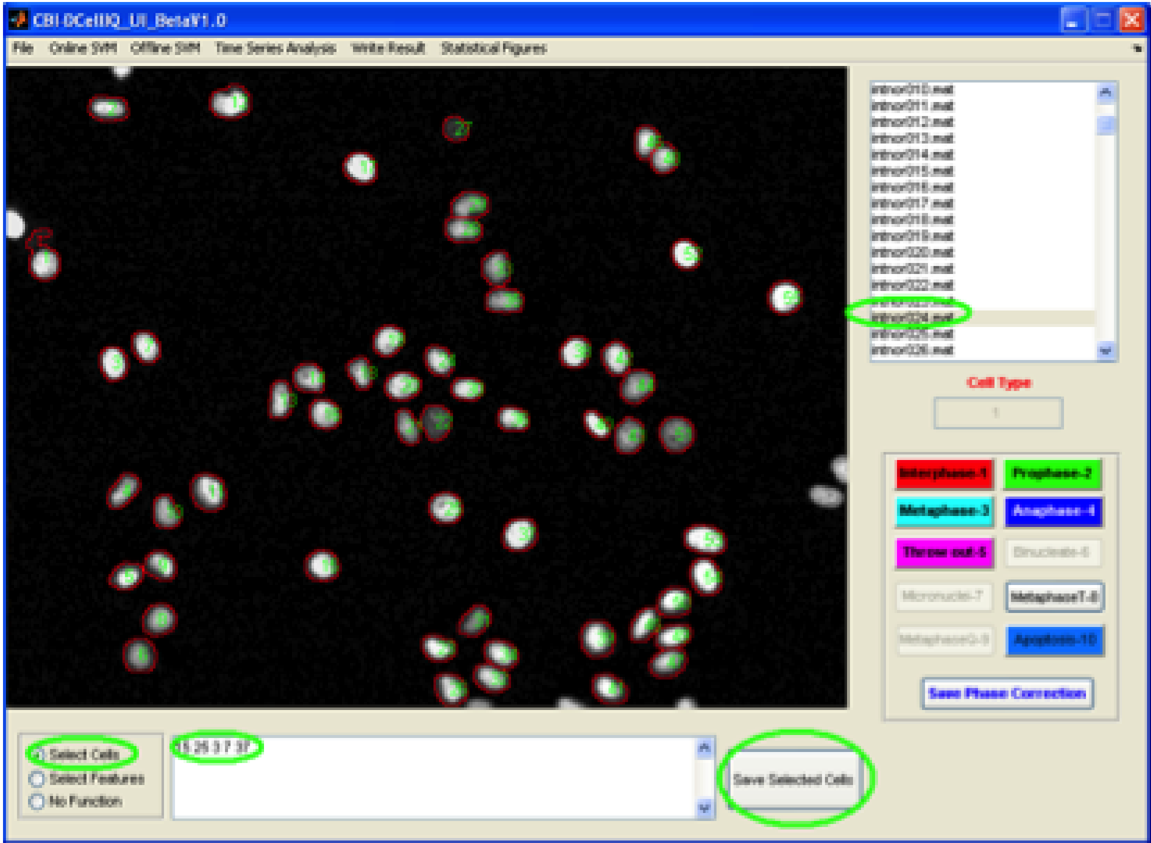

Figure 2.5: Cell classification in DCellIQui [4]

Like most packages, the extracted data is stored locally in various Matlab and Excel files, which can then be used by the researchers however needed. Once again, this makes it difficult for researchers to collaborate on a single dataset.

2.3 Dataset Types

In attempting to make a more generic tool, one must begin to understand the complexity and variety of domains that it would need to be able to cover. In doing this, we will focus on the applications that were initially required by the Multimedia Communications and Visualization Laboratory at the University of Missouri. The Firefly

Datasets	
Cell Labeling	Points
Vessel Segmentation	Lines, Contours, Boxes
Histopathology	Filled Polygons, Contours
Cell Segmentation	Contours
Satellite	Points, Boxes, Lines

Table 2.1: Examples of different datasets, and the tools needed

system has currently implemented the Cell Labeling dataset type, and the Satellite Imagery/Kolam dataset type. Each dataset type listed requires the ability to draw different types of objects within the image.

It is relatively easy to add new data types to the system in order to provide a more generic solution.

Chapter 3

Choice of Technology: Rich Internet Application

3.1 Requirements

In deciding on the technology that should be used for any application, the requirements should be analyzed, and weighed against the different options. The biggest deciding factor for the Firefly system was the need for a web based application due to the makeup of the lab. The initial users of the system were using different operating systems (Macintosh OS X, Windows and Linux), and access to the files needed to establish ground truth were scattered across different machines in the network. Using a web application was the obvious answer for several reasons. Browser-based applications offer the following advantages:

- Standardized tags/scripts are easy to develop (Rapid development, low development cost)
- No installation, updates or patches are necessary (Low delivery and maintenance costs)
- Applications are accessible from networked computers (Availability, flexibility)
- Applications can run on different operating systems (Platform independence)
- User interface (UI) is simple and standardized (Low learning curve for end users)

[5] However, traditionally, web applications have been extremely limited in it's user interaction as compared to desktop applications. Traditional web applications have the user perform an action that then typically corresponds to a page refresh with the updated information. This is a stark contrast to the rich interactive experiences that desktop applications can give us. In recent years, there have been advances in technology to attempt to bring the rich interaction of the desktop to the web.

In 2002, Macromedia talked about some of the features that would be needed for future web applications, coining the term Rich Internet Applications (RIA) [6]. Rich Internet Applications are described as a new model that combines the media-rich power of the traditional desktop with the deployment and content-rich nature of web applications. [reference]

Since then, many technologies have come out to fill that need. These include AJAX, Adobe Flex, Java and Silverlight. In comparing these technologies [7] has determined important areas of any RIA technology that should be considered.

Ultimately, in the end Adobe Flex was chosen as the main technology to present the Firefly system to the user. Using traditional HTML/Javascript/AJAX can make some very nice RIAs, but poor cross browser compatibility makes this extremely difficult to implement correctly. This especially becomes a problem with Firefly, since we need to be able to display data in precise locations within the application, so having the exact same experience across all browsers is crucial. Silverlight doesn't have complete cross-browser support yet, and the tools required to develop in it, are still costly - requiring a version of Microsoft Visual Studio.

Flex excels in all of the areas listed in [8]. Due to the fact that it runs as a plugin (through Flash Player), we can expect it to behave the same across different browsers and operating systems. Flash player's market penetration of almost 100 percent insures that practically anyone will be able to run the content. Also, having it's roots in flash content, it excels in interactive content, especially in it's ability to

Interaction	Possibility to specify the user (active) behaviors
Multimedia	Possibility to support the representation of graphics, audio, video, streaming and live multimedia
Visual continuity	Possibility to avoid screen refreshments and blink experiences
Synchronization	To provide an active (related with user interaction) and a passive (related with predefined behaviors) representation of interface elements
Dynamic Data Retrieval	Possibility to carry data to/from the server at run time
Parallel Requests to Different Sources	Possibility to retrieve data from one or more simultaneous sources, both in a synchronous and asynchronous way
Personalization	Extension for internationalization and localization, accessibility, multi-device access, etc.
Interactive Collaboration	It allows real-time interactive collaboration between different users in order to work together on the same task.

Table 3.1: RIA Comparison Parameters

draw vector graphics - which is used widely in the Firefly system.

3.2 History and Background

Adobe Flex began in 2004 as Macromedia Flex Server, a J2EE application using Flash Player as a means to display information in an interactive way to the user. Adoption was fairly small due to it's closed and costly nature. It is important to note that this was a complete server-side solution. While running the presentation in Flash Player gave the developer certain interactive attributes to the application, the actual application was still compiled on the server side. When Adobe completed it's merger with Macromedia in 2005, it released Flex 2 as the first rebranded application from the newly formed company. Flex 2 brought many important changes to the Flex framework, and began a much larger adoption of the technology by the web development community. The most notable addition was the introduction of ActionScript 3. ActionScript is the main scripting language used for developing Flex applications. ActionScript 3 is based off of the ECMAScript specification, and is a powerful, object-oriented language that can be leveraged for creating much more robust and modern web applications. With Flex 2, Adobe also began to reach out to more developers by significantly lowering the development cost associated with Flex apps. Flex 2 no longer required the very expensive J2EE application server to run the apps, and Adobe also released the SDK as a free download for developers to create Flex applications free of charge. Flex 2 applications also were compiled once, instead of being compiled on the fly by the server, meaning less costly servers could now be used to serve a Flex application.

Flex 3 continued Adobe's new commitment to open source, by releasing the entire SDK as open source under the Mozilla Public License [9]. While many of the concepts remained largely untouched from Flex 2, it did include a number of enhancements and performance increases over Flex 2. Adobe has recently announced it's intent to release Flex 4, code named Gumbo, in early 2010. Flex 4 continues on Flex 3's framework, with enhancements to help design applications.

3.3 Adoption and use as a RIA technology

With the shift towards providing a new open source toolset, and reaching out to more developers, Flex has seen an increase in adoption. In January 2009, Adobe announced that there had been over a million downloads of its developers tools, with major adoption as a rich internet technology. Adobe also opened up it's data transport technology specification, AMF (Action Message Format), so that it could be implemented across several different backend technologies, including Java, PHP and Python.

3.4 Flex Framework Overview

When talking about the Flex framework, it is important to understand that Flex at it's core is still a flash application. Flash's traditional goal was to provide a way to bring vector-based animations to the browser through the use of a lightweight plugin. As an animation framework, it's whole execution is based around the idea of frames. One can think of frames in the same way as frames in a traditional animation. An image is drawn on a particular frame, and then modified slightly on each following frame. Combining all of these frames together in a fast sequence, gives the appearance of movement to the user. Flash displays these frames at a rate defined by the user. Flex applications by default run at 24 frames per second. Understanding how code execution works within the frames is very important to building a Flex application.

Flash content is created by compiling ActionScript code to a SWF file that contains flash bytecode. This works in much the same way that Java applications are compiled down to java bytecode. The SWF file is executed directly by Flash Player, using the ActionScript Virtual Machine (AVM).

On the web, Flash is widely used for creating simple animations and simple interactive content such as games, audio and video. By itself, it is very difficult to

make more traditional web applications - which is where Flex comes in hand. Flex, provides an additional toolset and extra classes that help in the development of rich internet applications. This includes more robust ways of running remote procedures, a huge set of GUI components, and new language constructs to help in the layout of the application.

One of these tools is the use of an XML based language, MXML. MXML is used to declaratively lay out the interface of the application. This basically helps separate the code needed to layout the application from the logic that defines the interactions between different components within the Flex application. However, the MXML specification does allow for some logic by itself, since all MXML code is compiled to ActionScript during the compile process anyways. Sample MXML code is given below showing text being displayed in the middle of the application:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">
        <mx:Label text="Sample Label" horizontalCenter="0" verticalCenter="0"/>
</mx:Application>
```

Figure 3.1: Sample MXML markup

Flex is a highly event driven programming environment. Objects are created within Flex, and then listen for events from other objects within it. For example, when creating a button in a Flex application, one the more commonly used events is the "click" event. When the user clicks the button, a certain action should happen. As more complex objects are made, custom events can also be made as well that are dispatched when the user specifies. This model allows for classes to become more reusable, and less coupled, since they are not concerned with how other objects handle the different events that are dispatched.

Related to the idea of being event driven is the idea of data binding. Data binding

is the process of tying the data in one object to another object. Essentially, what this allows is to update a target property of an object given a source property. One can think of an example of using a slider interface control where you want to change the location of an object on the screen based on the current value of the slider control. The location of the object should be changed every time the slider's position is changed as well. Data binding implicitly or explicitly creates events that are fired every time the source property changes. When these events are dispatched, Flex knows that the target properties should be updated as well. This is very useful, and eliminates much of the boiler plate code that could exist otherwise.

Flex also provides a wide array of standard user interface elements, or components, that can be used to create rich desktop-like interfaces. These include datagrids, buttons, sliders, menus, etc.

3.5 The Elastic Racetrack and the Component Lifecycle

In looking at the Flash player execution model, it is important to remember that everything is executed on a speed that is based on the framerate defined by the developer. That is to say that either code execution and/or rendering occur depending on the current frame in execution. Ted Patrick, an Adobe employee, refers to the Flash Player as an "Elastic Racetrack" [10]. He describes a racetrack with two main sections, one section for code execution, and another section for graphics rendering. In every frame in the SWF file, the Flash player loops this "racetrack" performing the code execution first, and then renders any graphical changes that need to occur based on the the code that that was executed during that frame. The player will attempt to accomplish everything specified by the SWF file in the loop in the shortest amount of time possible, but not going faster than the speed designated by the framerate.

As is expected, there will be times that there is a lot of data being processed,

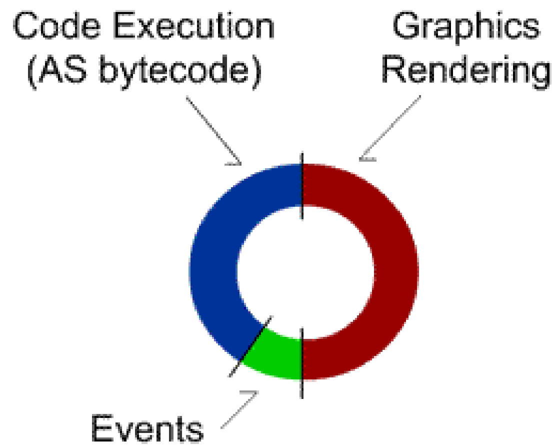

Figure 3.2: The Elastic Racetrack [10]

and there are times when a lot of graphical information is being drawn to the screen. The player will essentially stretch the racetrack to allot more time for either code execution, or graphics rendering, or both. This is why the racetrack is referred to as being elastic.

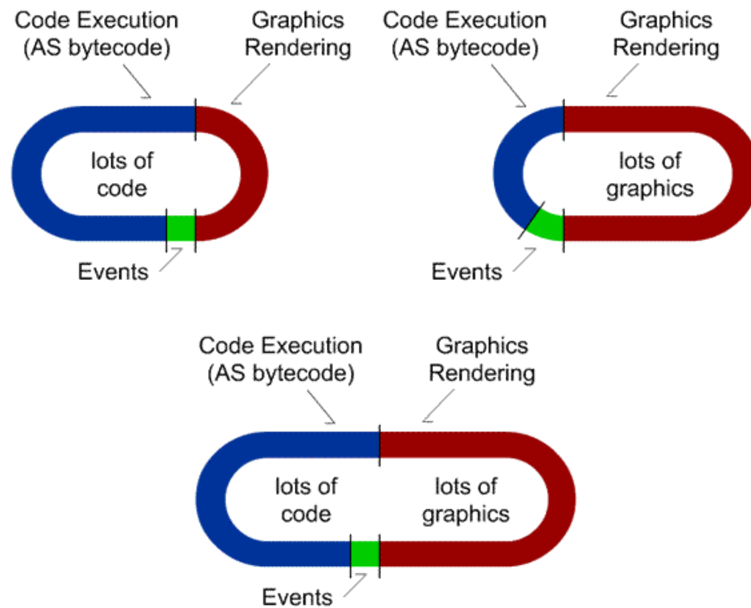

Figure 3.3: The racetrack stretches to accommodate is occurring during the cycle [10]

All of this is important to understand when developing applications to run in the ActionScript virtual machine, because trying too much code execution or too much

graphics processing during a single frame can greatly slow down the performance of the application, since it has to complete all actions before moving to the next cycle. To better optimize code, we can postpone execution of processor intensive code to happen at the last possible time, to increase performance - or as Ted Patrick puts it - "Just wait a frame!" [10].

The Flex framework is designed in such a way to make this easier to accomplish through the use of what is known as the component lifecycle during execution. The component lifecycle is a set of distinct steps that are followed for every visual object within the display tree and each step is executed to allow for them to happen at the most efficient time. They follow a component through it's construction, updating, and eventual removal from the display tree. While all of these steps are important to understand, the most important part for a developer to understand is the invalidation, validation and update states.

[11] Flex uses what is known as deferred validation. Aspects for each component related to code processing and rendering are deferred until the appropriate validation function is called. The three main validation functions are commitProperties(), measure(), and updateDisplayList(). CommitProperties is called to synchronize properties across the application. Measure is used to set the width and height of the component, while updateDisplayList is for updating all other graphical elements of the component, such as laying out it's children in the display tree.

This not only allows for there to be tighter control over when certain types of updates happen, but also allows code to be executed the least amount possible [11]. This is done since the functions are scheduled to be executed when their corresponding invalidate functions are called: invalidateProperties, invalidateSize and invalidateDisplayList. For example, when the height of a flex component is set, then the validation function measure() should be run as well. We could potentially run the function explicitly every time the height is changed. However, this doesn't seem very efficient

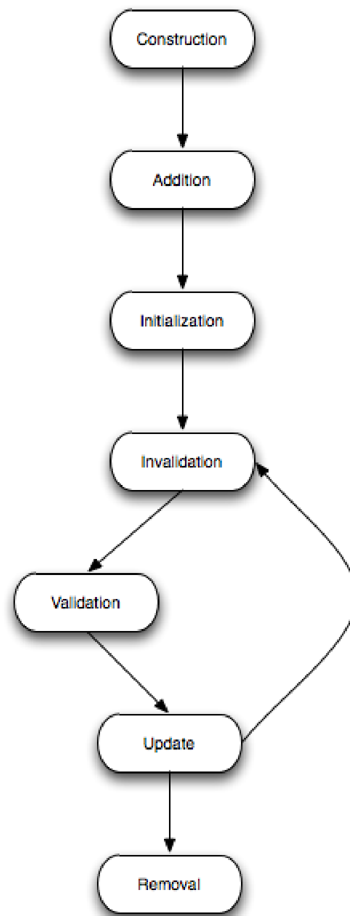

Figure 3.4: Flex Component Lifecycle

considering that this function could potentially be called multiple times before the current frame is even rendered. To stop this, the measure function is only scheduled to be called once by using the invalidateSize function. Then before the component prepares to render again, the measure function will be called. This way, no matter how many times a height was set during a frame cycle, the actual updating only occurs once. It also important to note that it does not necessarily have to run every frame cycle. It is only run if the size had been invalidated using the invalidateSize() function during that particular cycle.

3.6 Frameworks for a Framework

By definition, Flex is essentially a web application framework that extends the base functionality of Flash Player. It provides a collection of common code that can be extended or overridden to create powerful web applications. However, as Flex began to grow in popularity, the need for more defined best-practices within the Flex framework grew, as applications began to be created in parallel by large development teams. With this need, arose several different micro-architectures within Flex, that help encourage best practices, and design patterns when creating an application. These micro-architectures are more commonly known as Flex Frameworks.

By using a Flex framework, developers not only develop using tested design patterns in their applications, but also create more reusable code that can be changed and expanded by anyone else familiar with that framework. Probably the most important design pattern followed in these frameworks, is the Model-View-Controller (MVC) design pattern. The main idea behind this design pattern is to separate the underlying data of an application from how the data is ultimately displayed to the user. As the name suggests, there are three main parts in a MVC framework, the model, the view and the controller. The model represents the actual data, the view is the presentation to the user, and the controller helps in the interaction between

the view and model. Using this approach, the application can relatively easily change where the data is coming from, or how the data is presented, by only changing the affected portion of the application. Each part should still be making the same calls through the controller. Flex applications are not able to implement the MVC design pattern in the traditional sense however, since the application normally doesn't have direct access to the data source. Since the SWF runs locally in the Flash player, it has to make remote procedure calls back to the server any time it want to access data. This leads to what is knows as a dual-mvc deployment where models and controllers exist on both the server and the client [12]. The models that exist in Flex are typically mirrored from the data that is received back from the server after a remote call.

Two frameworks were ultimately used in the development of Firefly, Cairngorm and Swiz. Cairngorm was used during initial development, but was ultimately changed to the more flexible structure that Swiz allows.

Cairngorm is one of the oldest frameworks, and probably the most widely used at the enterprise level. It was developed by Adobe and works well in enterprise situations due to it's highly structured organization. It follows an event flow model, where any interaction with the controller/server happens by firing events that the controller is listening for.

One of the biggest complaints of the Cairngorm framework is the amount of "boiler plate" code that has to be written in order to use the framework. As shown in the Cairngorm event flow, even in the simplest example, one would have to write five classes just to send and receive data from the server. Cairngorm also relies on a global singleton, known as the model locator. The model locator is essentially a place for global storage of variables that are used to update the view. This makes modularization of the application even more difficult, and code reuse much more difficult.

The Swiz framework, attempts to simplify the whole process by merely providing

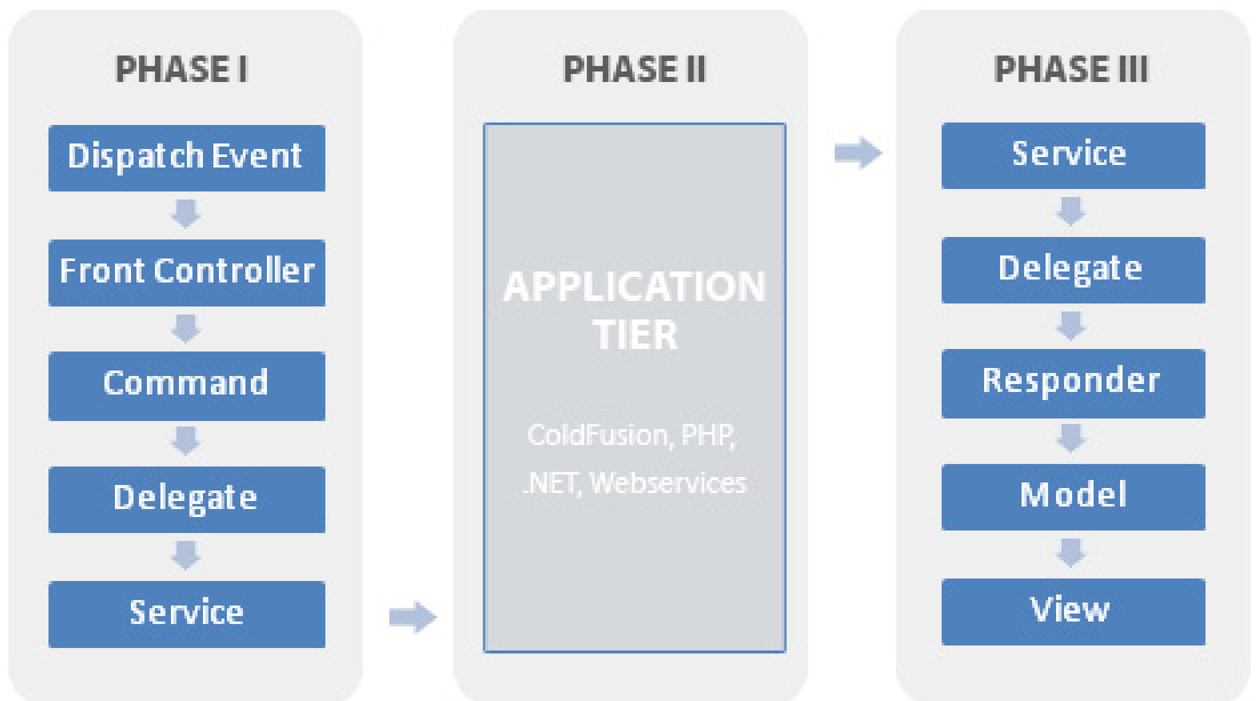

Figure 3.5: Cairngorm Event Flow [13]

a set of classes and tools that can be used to help in the process of making a MVC framework. Swiz is an inversion of control framework that uses the idea of dependency injection. Essentially what Swiz does, is provide all data that is needed by an object for successful execution. This is sometimes called the Hollywood Principle - "Don't call us, we'll call you". Swiz uses the idea of a factory that contains a collection of singletons that are then able to be set as properties of different objects without the objects having to explicitly call for them. This leads to a cleaner implementation of the MVC design pattern, by strictly keeping the different parts separate. As explained earlier, Swiz does not enforce a specific code structure like Cairngorm, which is seen as a weakness by some, but does allow for more flexible code depending on the project at hand. [14]

3.7 AMF and Communication with the Server

As with most rich internet application technologies, Flex is a ”fat client”, where most of the interaction with the application happens client side, and then only sending and receiving data to the server when needed. What this means is that Flex or Flash has no way of interacting directly with the data being stored on a server, such as a database, or the filesystem. All interaction with the server has to take place with some sort of service call to the server.

Flex has two main ways of interacting with the server: HTTP service calls and AMF streaming. HTTP service calls interact by making HTTP requests to the server using GET and POST variables, and then receiving certain information back. Typically when dealing with flex, you will want to receive the data back as XML, since flash player is able to natively parse the XML quickly.

AMF, or Action Message Format, is a binary format that can be used to serialize and transfer ActionScript objects. This has several advantages when dealing with Flex applications. Firstly, when dealing with large amounts of data, this can prove to be much faster than typical HTTP requests. The serialized object is compressed using zlib before sending it over the wire, and doesn't need to send the headers that HTTP responses require. Another major advantage is it's ability to automatically serialize objects into strongly typed objects between both the server and client. This is especially useful since Flex frameworks use a dual-mvc setup, where models exist on both the server and client. Now, for example, if we had a Java class User, and sent it to Flex via AMF streaming, Flash player can automatically serialize the object into the corresponding ActionScript class User. This can be a huge help in dealing with complex data that is being sent back and forth between the client and server.

The AMF specification was opened to the general development community in 2007, with most major server side languages now having bindings to it. Firefly uses AMF streaming to communicate with the server using PHP.

Chapter 4

Ground Truth Utility

4.1 Requirements

The Multimedia Communications and Visualization Laboratory at the University of Missouri was involved with analyzing and classifying cancerous cells, known as HeLa cells. This was done in collaboration with a lab in Europe, making it difficult at times to work together on parts of the project. The data analyzed comprised approximately 200 images taken in sequential order composed in a frameset. As the cells would progress through the sequence, they would change state, including splitting during the mitosis stage. The cell images were analyzed using various algorithms to determine not only regions of the cells, but also the class, or state of the cell during it's life cycle. As discussed previously, obtaining ground truth of the cells locations and classes was needed in order to verify the results. In order to classify these results, a professional in the field was needed to mark on every image in the sequence the current state of each cell. Obtaining this ground truth, would prove to be very difficult to do, manually.

From these requirements, an initial prototype was created, known as the Ground Truth Utility or GTU. This was a a very dataset specific application for the HeLa cells dataset. Since the regions had already been determined for the cells in the sequence, only a point within the region was needed in order to identify them. That is to say, the researchers merely needed to mark one point inside of a cell to determine what

Cell Classification	
MI	Mitosis
G1	Gap 1
SE	Synthesis Early
SM	Synthesis Middle
SL	Synthesis Late
G2	Gap 2
AP	Apoptosis
?	Undecidable

Table 4.1: Required classes for the HeLa dataset

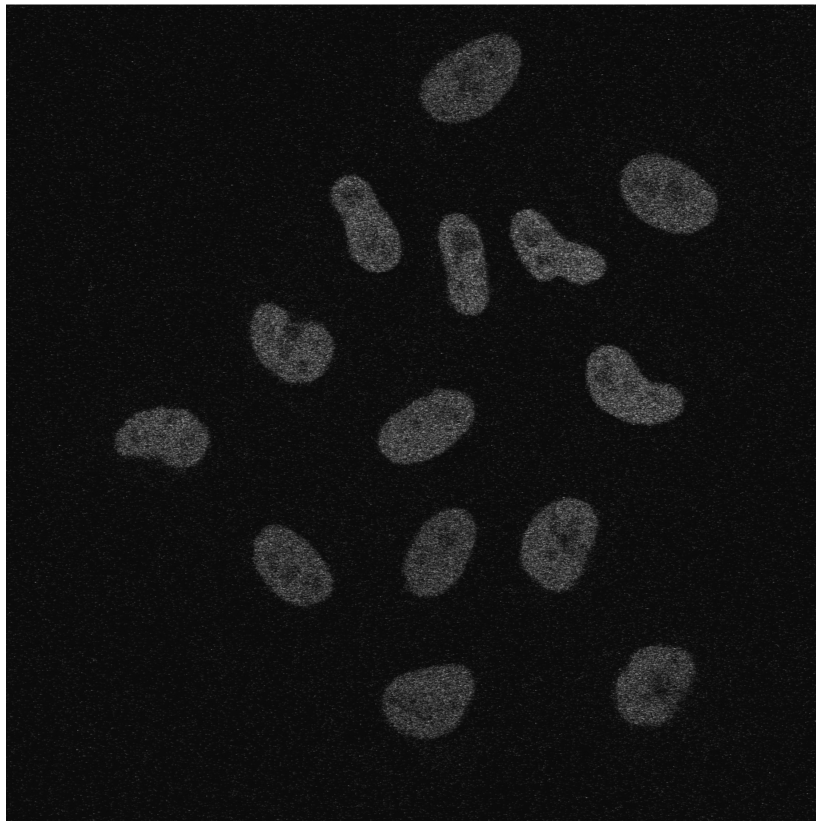

Figure 4.1: Sample image from the HeLa dataset

cell is being described. Once a point has been marked, then different attributes, and most importantly the classification, can be associated with it. Since this was a collaborative work, it was decided that web-based tool would be ideal as discussed earlier, using Adobe Flex as the main technology.

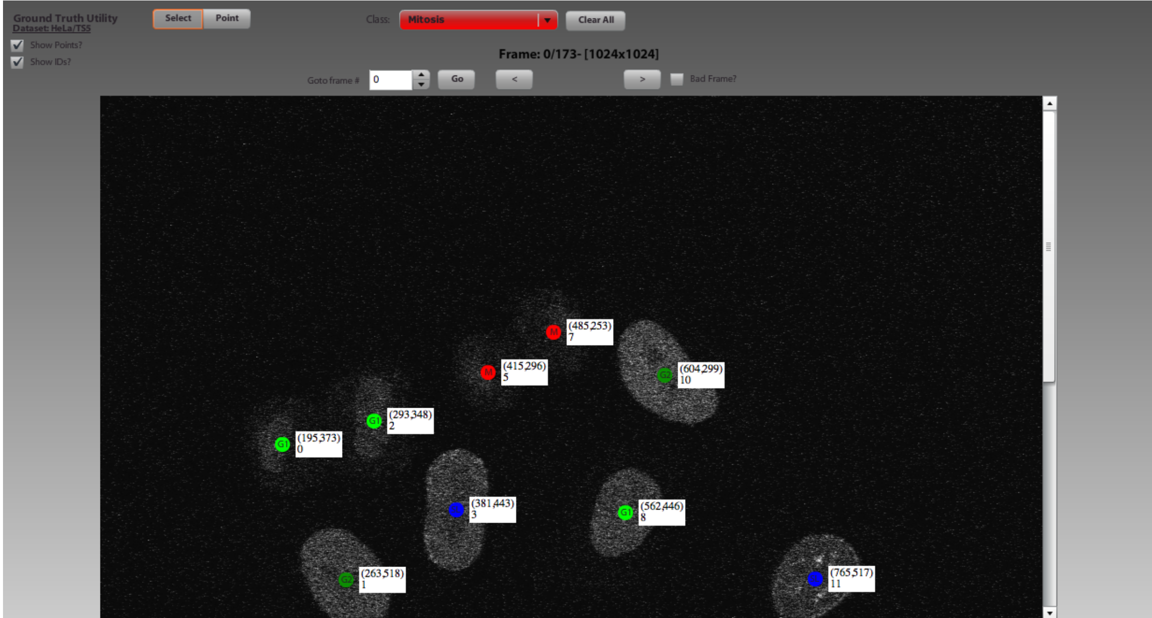

Figure 4.2: GTU Interface

4.2 Implementation

GTU was a very basic tool that only used data files on the server to store and retrieve the data associated with the sequence sets. All interaction between the Flex application and the server took place by passing POST variables to PHP scripts residing on the server. The server had two main folders, data and images. By passing in the identifier "HeLa.TS5" as a POST variable, the system would then know to look in the HeLa project folder and then the TS5 dataset folder. So, the images would be stored in "images/HeLa/TS5", and the data files being stored in "data/HeLa/TS5". This approach, while rudimentary, does allow the user to add and change the datasets very easily. The format of the data files was a plain text comma delimited file, with

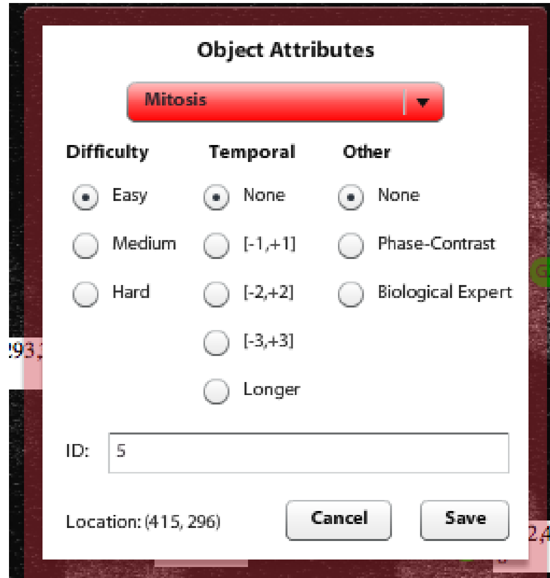

Figure 4.3: Assignable Attributes in GTU

each row corresponding to a cell in the image. This was predefined to allow for other programs such as Matlab to easily read it in. The real power of the application was in the interactivity of Flash player. One of Flash's biggest strengths is it's ability to handle and draw vector graphics within itself. The user was able to interactively draw points on top of the current image in the sequence, and then assign it's class and attributes.

GTU was used successfully for several datasets, but eventually some of the weaknesses of it began to become more apparent. Since all users of the system, were basically accessing the same filesystem, there began to be issues with concurrent editing. The data files were completely rewritten back to disk anytime a change was done. This was fine as long as there was only one user, but if two users were annotating at the same time, a race condition occurred with the possibility of data loss. Essentially, if two users were actively editing the same frames in the system, you have no way of merging their information into the file. Since the data is saved on exit from the frame, the last user to exit is the only one that has data saved.

Other limitations of the system related to the interface. Users were unable to

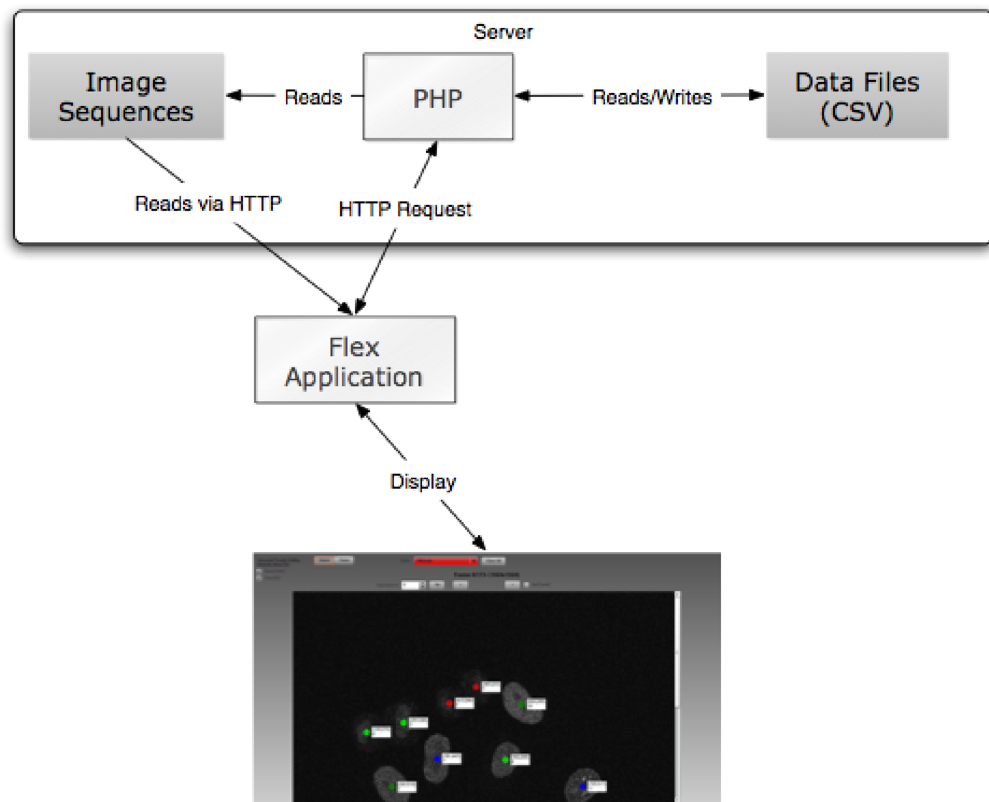

Figure 4.4: GTU System Structure

move points once they were drawn on the image, meaning the point would have to be completely erased, and redrawn to move it. Also, there was no buffering solution created, meaning that every time the user advanced to a following or previous frame, they would have to wait for the image and data to load again, which could take up to several seconds depending on the size of the images. GTU proved that the concept was valid, and that a more robust, generic tool could not only help in HeLa cell classification, but also be used in future datasets, where ground truth was needed.

Chapter 5

Firefly

5.1 Overview

The Ground Truth Utility was very useful for the HeLa cell image sequence described earlier, but was made specifically for that dataset. In reality, a more generic solution would be needed to provide the ability to mark ground truth on different types of datasets. We have proposed a generic solution that is able to work with multiple different data types, including images, image sequences, and high resolution tiled pyramid files.

One of the main issues encountered with GTU was it's inability for different users to have concurrent read and write access to the data files on the server. This is important as many times, two different people will work on the same dataset. To solve this, a centralized MySQL database was setup to store all of the data associated with the different datasets. The database encompasses the new organizational structure of the entire system.

One of the biggest new features was to have some sort of user management built in to the system. Firstly, all users of the system are required to login in order to use the system. By tracking what user is logged in, we can now enforce access control on the datasets in the system. For example, by limiting the amount of users who should be able to annotate images in a dataset, we can avoid different users putting conflicting annotations, accidental deletion of annotations, etc. We are also now able

to track which users are responsible for annotating and creating datasets for auditing purposes.

When creating the Firefly system, it was also designed with the intent that it could also eventually be used in touch/tablet type environments. Being able to use a tablet computer to make annotations on datasets has proven very useful for professionals to easily mark and annotate.

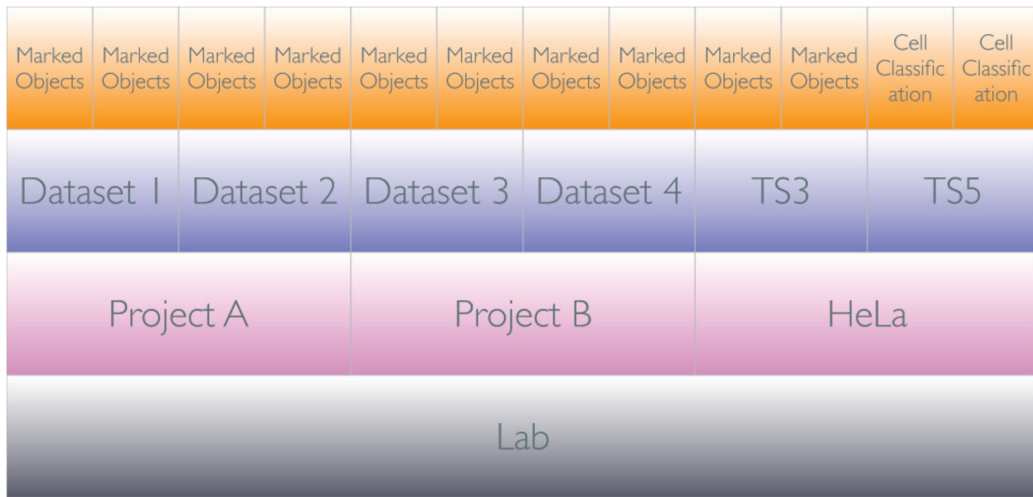

Figure 5.1: Organization of the Firefly System

The general organization is ultimately up to the end users, but Firefly allows for several levels. At the top level, there are labs. Labs are any group of people that work on many different projects together. By using labs, Firefly could run at a larger organizational level, such as a university, which would then have multiple labs underneath. Underneath labs, are projects. An example of a project would be the HeLa cells that were dealt with in the earlier examples. Projects also dictate the type of information that is associated with each dataset (e.g. the classes defined to represent the different states of the cells). Projects typically contain multiple data sets. Datasets refer to the actual data that is being analyzed - whether it be particular sequence of ordered images, or a high resolution image file.

Going back to the whole reason for the Firefly system, we are wanting to annotate the datasets with ground truth, so that the data can be used to verify any sort of image

analysis done. In the original GTU system, these annotations were encompassed in the points that were drawn on top of the different images in the sequence. In an attempt to make the system even more generic, we use the idea of a ”marked object”.

A marked object refers to any piece of related information in a dataset. Typically, these have some sort of location associated with it. So, for example, we have the points in the HeLa set, that then have a class and other different attributes associated with it. However, in making a more generic system, we cannot just limit marked objects to points. For the initial system, we have expanded that toolset to include points, lines, bounding boxes and rasterized free form drawing. Marked objects are not limited to only visual objects either - there are non-visual attributes as well that can belong to an entire image for example. This information is fairly straightforward to represent in the database implementing a one-to-one inheritance relationship for the different marked object types.

5.2 Implementation

The general implementation of the Firefly system fixes many of the issues that GTU suffered from. As a whole, the system still uses a Flex front-end, but now utilizes the Zend Framework running on PHP to create a more robust, and expandable data-access layer. All communication is done through AMF streaming to the PHP backend, which in turn uses a MySQL database for data storage.

The most noticeable changes for the user occur within Flex. Once a user chooses a dataset to work with, the main focus of the entire workspace is on the data actually being displayed. All of the interface elements have been designed to either be hidden when not needed, or merely float over the actual images. With the addition of the new tools (Point, Line, Box, Free Draw), the users now are able to change resize, and move all marked objects within the system. This can help greatly in reducing the amount of time spent working in the system.

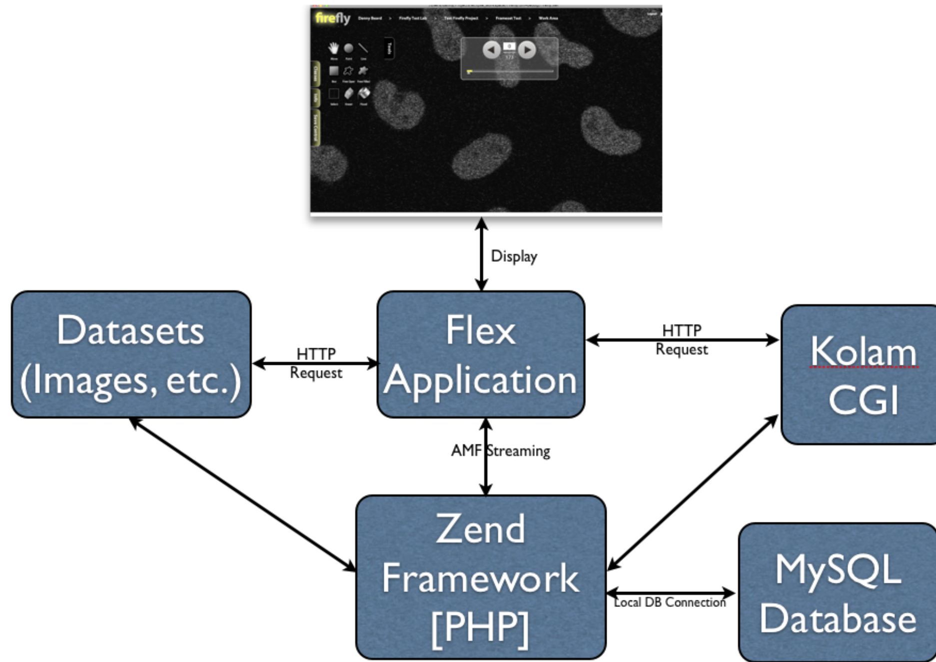

Figure 5.2: The Firefly System

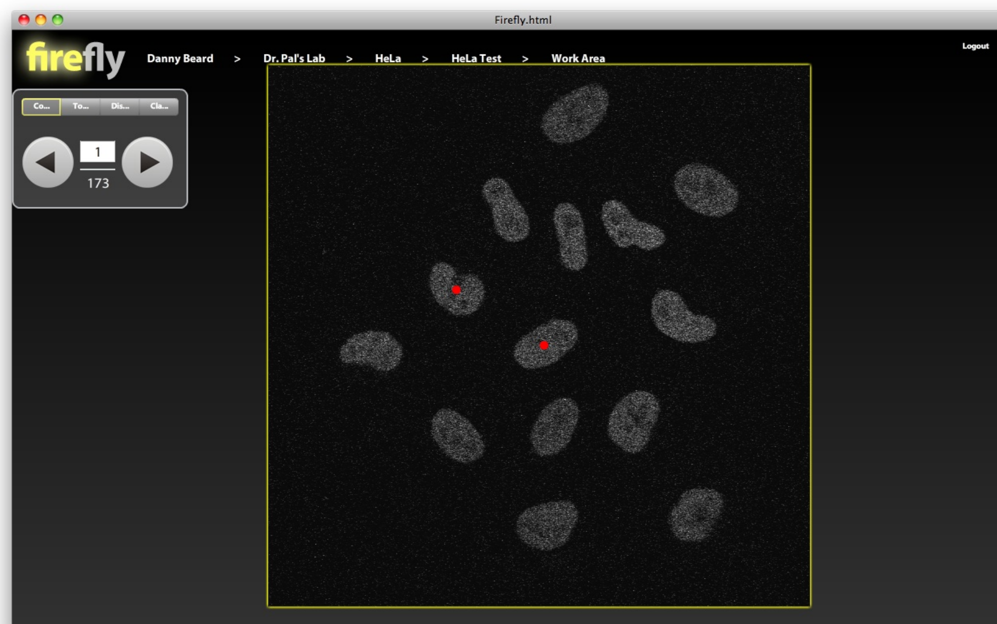

Figure 5.3: Sample Firefly Interface

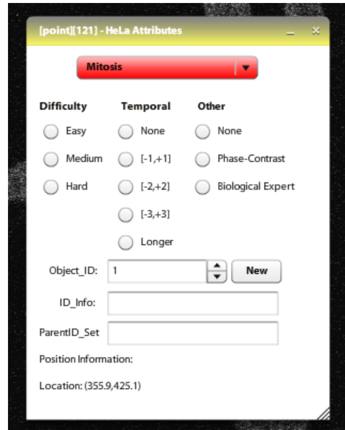

Figure 5.4: Attributes stored with a Marked Object

The classification portion of marked objects is now much more robust, allowing for different classes to be hidden, depending on the needs of the user. This works much in the same way as layers in traditional image processing applications work.

5.3 Implementation Hurdles

In the actual implementation of the system, multiples hurdles were encountered. One of the major requirements and needs of the system was to make it flexible enough so that we would be able to add new features and enhancements in the future. This, along with other unique features of the system led to distinctive problems that had to be resolved. The general state diagram of the system can be seen below:

5.3.1 Generalized Workspace

One of the bigger issues in developing the Firefly system was a way to create workspaces that would allow all possible types of future datasets. This problem is demonstrated in using the two main data types that were used in the initial prototype - image sequences and tiled pyramid files. The image sequence dataset type requires it's own set of unique tools and controls that other data types would not need. This includes to ability to navigate through the images throughout the sequence, the ability to copy marked objects between the images, and more. The tiled pyramid file merely needs

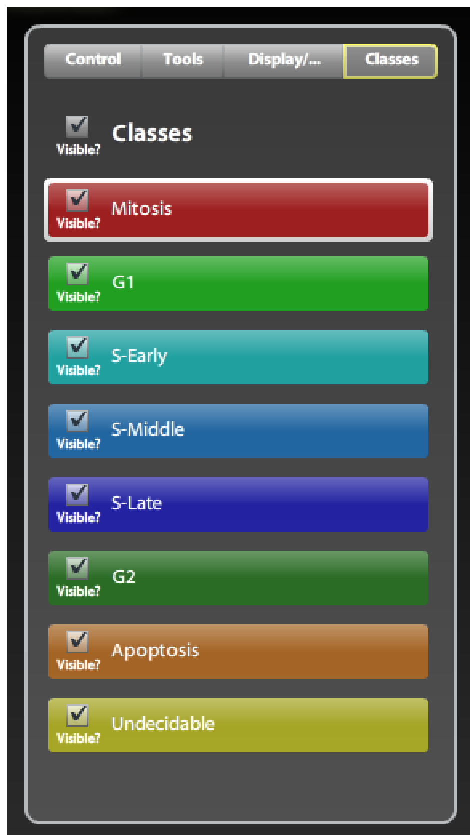

Figure 5.5: Classification Layers

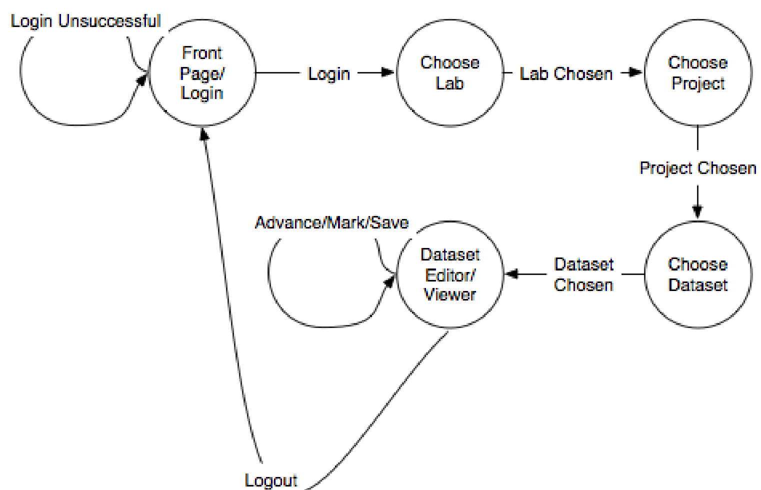

Figure 5.6: State Diagram of the Firefly System

a way to zoom in and out of the image to allow for easy placement of the marked objects.

While the different dataset types have obvious differences, there are also similarities as well. Most dataset types include a certain amount of tools that are used for creating the marked objects. Most of them also use a class system for classifying the marked objects. Using these similarities, a collection of base user interface controls were created that could be extended and used throughout many different dataset types. These controls include things such as a toolbox for selecting the current tool in use by the user, an informational panel on the dataset, a way of displaying and hiding the classes associated with the data, and a buffer control (This would be used with temporal navigation).

Using the markup language already built into Flex, a custom ActionScript class was made that can easily interpret and layout the components needed for each different type of workspace. In order to make a new workspace for use with a new datatype, or project, a simple XML based class can be made to merely layout the components. Typically, very little customization is needed, but it is possible to be extended.

A factory design pattern is implemented to actually create the objects within each dataset. Basically, each dataset type has a factory class that can tell Flex, what should be loaded to create the workspace, and what should be loaded to create the attribute window.

5.3.2 Buffer

One of the more glaring issues with the original GTU system was the lack of a buffering system for the images. Typical image sets consisted of over 200 images, and trying to quickly navigate through the images was a hassle, since a new image had to be loaded every time the frame was changed. In Firefly, a generalized buffer solution was created to handle any type of data that could be buffered or saved locally. For framesets, the desired behavior is that the system should load all surrounding

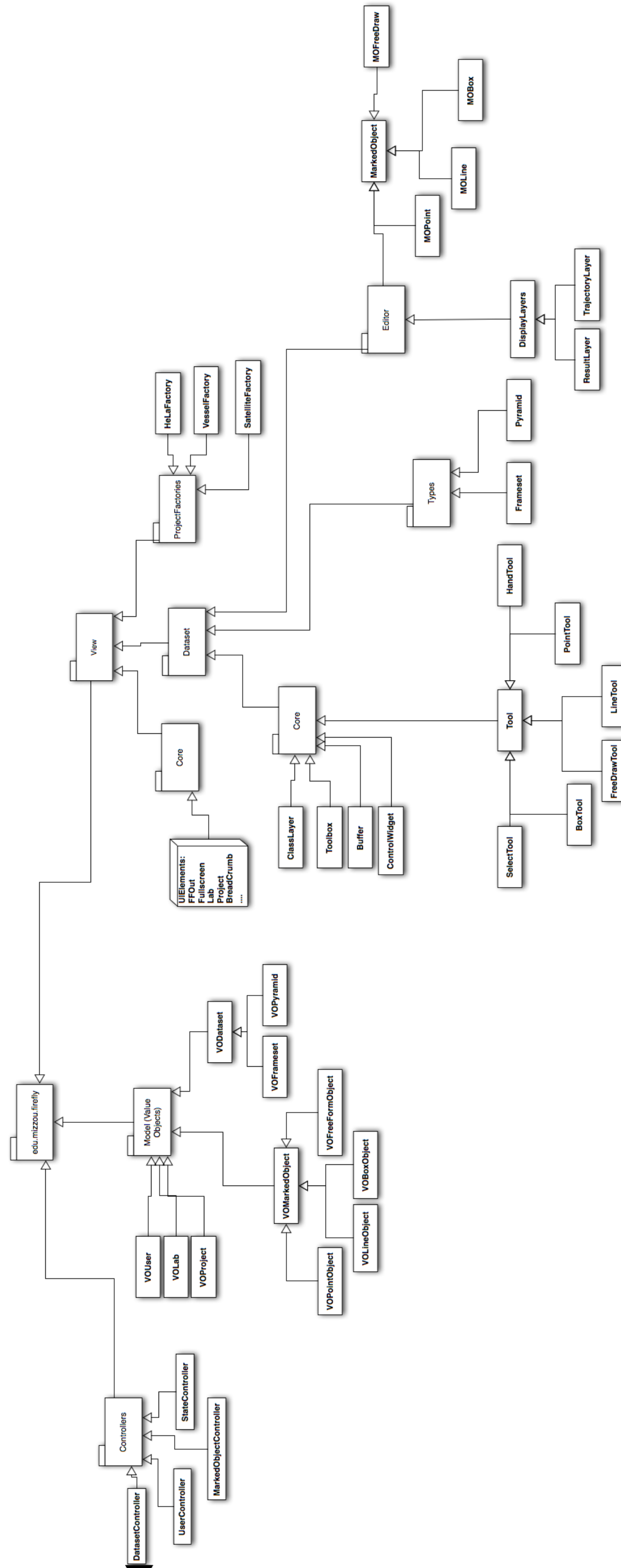

Figure 5.7: Firefly Class Diagram

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Canvas xmlns:mx="http://www.adobe.com/2006/mxml"
        xmlns:core="edu.mizzou.firefly.view.dataset.core.*"
        xmlns:layers="edu.mizzou.firefly.view.dataset.core.layers.*"
        xmlns:frameset="edu.mizzou.firefly.view.dataset.frameset.*"
        xmlns:tools="edu.mizzou.firefly.view.dataset.core.tools.*">

        <frameset:FrameBuffer/>

        <core:SlidingTabNavigator>
                <mx:Canvas label="Tools">
                        <tools:ToolBox>
                                <tools:dataProvider>
                                        <mx:Array>
                                                <tools:HandTool/>
                                                <tools:PointTool/>
                                                <tools:SelectTool/>
                                                <tools:LineTool/>
                                                <tools:BoxTool/>
                                                <tools:FreeOpenTool/>
                                                <tools:FreeFillTool/>
                                                <tools:SelectTool/>
                                                <tools:EraserTool/>
                                                <tools:PaintBucketTool/>
                                        </mx:Array>
                                </tools:dataProvider>
                        </tools:ToolBox>
                </mx:Canvas>
                <mx:Canvas label="Classes">
                        <layers:Layers/>
                </mx:Canvas>
                <mx:Canvas label="Info">
                        <frameset:FramesetInfo/>
                </mx:Canvas>
                <mx:Canvas label="Save Control">
                        <core:SaveControl buffer="{buffer}"/>
                </mx:Canvas>
        </core:SlidingTabNavigator>

        <frameset:FrameAdvance frameBuffer="{buffer}">
        </frameset:FrameAdvance>

</mx:Canvas>
```

Figure 5.8: Sample workspace code in Firefly

frames from the currently active frame, so that the user is able to quickly navigate through the frames. Memory becomes an issue in dealing with the buffer. One of the limitations of the way Flash handles images is that when a compressed image format is loaded into the player, it is stored uncompressed in memory. At first, it was thought that since the space taken by the images on disk was less than 200 megabytes, we could load all images in the buffer since most modern computers would not have a problem allocating 200 megabytes for flash player. However, once these images were uncompressed, the amount of memory resources needed became huge. With this in mind, the new buffering solution needed to only keep a smaller subset of the images in memory at a time. The first attempt of making a buffer involved making a first in, first out queue, where the images would be added into an array of fixed length, which would be cycled through, to store the newly loaded images. The first image loaded in the buffer, would be the first image to be overwritten when the buffer is full. However this is not the best solution for the frame buffer problem. For example, if we are currently on frame n, then the buffer should begin filling itself by load frames following this order: n+1,n-1,n+2,n-1,....,n+l/2,n-l/2, where l signifies the size of the buffer.

If we move to frame n+1, in reality, it should only have to load one more frame to the buffer (n + 1 + l/2), eliminating the frame located at n-l/2. However, if the queue was used, then the newly loaded frame would actually eliminate the frame at location n, since it was the first loaded frame in the buffer.

In order to get the desired behavior, you need to reorder the items in the array to match. Every time the current viewable frame is changed for example, it will again attempt to fill the buffer moving out in the usual way, but first checking if items exist in the buffer. If they do exist, they are moved to the front of the array. So, when a frame is reached that has not be loaded, the last item is popped of the array, and replaced with the newest frame. This keeps the always keeps the needed frames in

the buffer.

5.3.3 Pyramid Files

One of the requirements of the new system was to be able to handle several different datatypes besides just the frame sequences like the HeLa dataset. In it's current implementation, we have allowed for the display and ability to annotate large resolution images within Flash player. These images are typically stored in some sort of a pyramid format. Essentially what this means, is that the source image is scaled to different resolutions, creating layers, and then each layer is split into tiles of equal sizes. This creates essentially a pyramid of tiles, since the highest resolution will have the most tiles, compromising the base of the "pyramid", and the lowest resolution layer, will typically only have a couple of tiles, compromising the top of the "pyramid".

This way, depending on the current zoom of the image, we get the layer, that most closely matches it, and then retrieve the corresponding tiles that are needed in the current display region.

Creating a system that can display these images, is a fairly difficult task, but several open source tools have been created to speed up the process. For this project, the OpenZoom library [15] made for Flash player was used. This library was originally made to read deep-zoom images - a pyramid file type created by Microsoft. A custom library was made to make the Kolam datatype readable by the OpenZoom engine. This engine is being looked at for expansion to displaying temporal data as well.

5.3.4 Raster Drawing

Raster drawing was a challenge for the tiled pyramid files due to their size. Flash player is limited to displaying rasterized bitmap content smaller than 4096 by 4096 pixels. Many of the tiled pyramid files can greatly exceed this resolution. This means that we will be unable to use only one bitmap object within flash to represent the hand drawn information. Along with this, for performance reasons, we would not

want to load all of the bitmap data from the database for such a large image if not needed.

The solution was to tile the bitmap information as well. Each bitmap tile is a set size, and then tiled according to the size of the image. Using this method, we can once again use the buffer class created previously, and limit the amount of data that needs to be drawn on the screen. This information is saved back to the database as a compressed bytearray. Early tests of this method have shown it to work well for very large images.

Chapter 6

Conclusion

6.1 Future Work

The Firefly system was designed knowing that future enhancements would be added later. The structure of the code and the system was made to be as modular as possible. These enhancements will not only add greater value to the tool as a way of gathering ground truth, but make it an even more generic utility for the researchers to better visualize and understand the various data sets they analyze.

While the Flash player virtual machine is a powerful tool - it is not able to process data at the speed and ease that other tools such as Matlab is able to. Because of this, we saw the need for somehow interfacing between Firefly, and other applications. Currently data is able to be extracted easily, but sending processed data back to the system is unsupported. An example where this might be useful would be using an existing Matlab program to segment a specified region of the image currently being analyzed. While a native Actionscript module could potentially be made to do this, it would be somewhat difficult, and be redundant to redo code that has already been written and tested in other languages. Interfacing with other programs would either have to be done through web service calls that are then able to interface directly with these applications, or run Firefly locally, and then allowing it to directly access the other tools.

Using web services would allow for Firefly to still run through the browser and give

all users of the system access to these new tools. It does require a significant amount of overhead in setting it up, since all of these services would need to be compiled to run on the server that is housing the web services. The system also would have to deal with bandwidth issues back to server, especially since the amount of data sent could be relatively large.

Running it locally allows for faster data transfer, but limits the users who could use this functionality. Since Firefly was created using Adobe Flex, making a local version would be relatively simple using Adobe AIR technology. Essentially we would be able to compile the same application to be installed locally on machines, thereby removing the security sandbox imposed by Flash player, and allowing the system to access the local filesystem. Again, this would need to be configured per machine, effectively removing the collaboration aspect of the system.

Along with interfacing with other tools and applications, would be the ability to interface with alternative data storage systems. Currently we are only using the MySQL backend as a way to store all of the data captured. We would still like to leave the option for some labs to still use a flat file, possibly XML based storage system, where it may be difficult to actually setup the database. Also, there are many accepted database standards for different dataset types. For example, for biological microscopic images, OMERO has become a standard tool to use within the system. It utilizes many standards for the storage of this data, including it's own database backend. It would become important in the future to interface with this backend for easier use with other labs who might not necessarily be using the Firefly tool.

Another important feature that is being worked on is the ability to store free-form drawing in a vector format. Currently, the data is stored in a rasterized bitmap, but this is sometimes not the ideal way of storing the data, especially when it needs to be compared with other vector information for the dataset. Flash player has limitations in the amount of points it can display currently, so we are looking at other ways for

storing complex contours efficiently within the system. One way of doing this would be to store the contours as chain codes within the system.

It has been found that many times, it would be a great help to be able to see not only the ground truth of a dataset, but also superimpose the results of the algorithms being tested as well. Besides giving researchers a unique view of their dataset, it also could potentially speed up the time needed to mark these datasets with ground truth. It would be much quicker to merely correct the automated results when there are errors instead of having to completely recreate the data that mostly exists already through the algorithm.

Other enhancements add to the types of datasets that are supported. There are framesets that also contain multiple layers for each frame - essentially giving the images depth. This would require the base buffer created to be expanded to support buffering in two possible directions.

One of the more technically challenging enhancements include expanding the Kolam support now implemented to support temporal scrubbing as well. This is useful to help track moving objects within the high resolution satellite images for example.

6.2 Summary

Firefly, while still being improved and enhanced, has been created using the ideas and lessons learned from the proven GTU system. This system was created using Adobe Flex, which gives us a highly interactive and robust experience for dealing with many different dataset types that could be encountered when obtaining ground truth.

Using the ground truth obtained through the system, researchers are able to verify the results of their image analysis algorithms. By using a central server and a database for concurrent access to the data, researchers are also able to work in a collaborative environment, also giving them tighter control on access to these datasets. Future work will only expand the power and possibilities of Firefly as an even more robust

and generic tool.

Appendix A

Computational Provenance

A.1 AVA - Advanced Video Archive

One of the original motivations for making a web based system was the need to be able to better organize the projects and datasets that were being processed. The Multimedia Communications and Visualization Laboratory at the University of Missouri was having issues keeping track of where original data came from, and where all of the processed data was located. Essentially, the data was put through a pipeline of processes that produced several different useful results. Typically, different members of the lab were involved with different aspects of the processing, resulting in the data being scattered throughout different computers and locations. It became difficult to track this information. This is the beginning of the need for a more centralized system.

To help in originally solving this problem, the Advanced Video Archive was created (AVA). AVA allowed for a certain amount of fixed attributes to be assigned to each dataset, and then tracked each process/output as it went through the data processing chain. The attributes tracked can be seen in the table below:

This was all purely manual data entry, and very little was done to automatically create this data. The processing chain was displayed in a tree format that helped the users better visualize how the original data had been used, and what processes are necessary for specific output.

Filename	Name of the physical file in the system
Geographic Location	The latitude and longitude of where the data was captured
Date/Time	When the data was originally captured
Video Camera Type	The type of camera used (and it's parameters)
Keyword	Used for easily searching for and retrieving data in the future
Source	Person who acquired the data, or network location where it was acquired
Publication	Any publications that utilized this data
Authorization	If the data was classified, or restricted to a certain subset of users
Acknowledgment	List of acknowledgments needed for use of the data
Comments	Any quirks or comments that should be known about the particular dataset

Table A.1: Attributes stored in AVA system

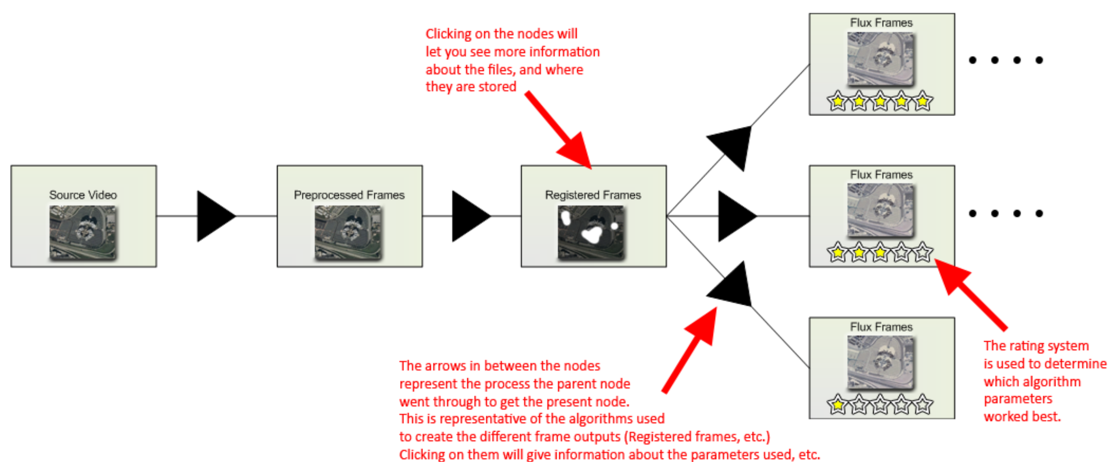

Figure A.1: The tree structure showing the AVA processing chain

A.2 Computational Provenance

AVA is a solution for a more general area of study known as provenance management. The dictionary defines provenance as the source or origin of an object; its history and pedigree; a record of the ultimate derivation and passage of an item through its various owners. [16] We have already seen how this is important with the AVA system. We are interested in information like where did this data come from? How was this data created? Traditional methods of keeping track of this information has typically led to confusion and loss of data, as shown with the motivations for AVA.

With this in mind, computational provenance, or modeling the provenance of a dataset in a computational model, has become very important for researchers. Researchers want to be able to easily track their results, the processing chains that were created to get there, and what were the inputs and outputs at each step along the way.

For computational provenance there are two main types: prospective and retrospective. Prospective merely allows a researcher to create a chain of processes that would be necessary to create certain types of output. It is not necessarily concerned with the details needed to produce this information. Retrospective provenance is recorded after execution or at every process during the chain. It records detailed information about the parameters needed for the different algorithms to derive a particular output of data. [16]

A.3 Provenance Tools

There are already many tools that exist to facilitate computational provenance. One of the future goals of the Firefly system would be to utilize some of the concepts of these systems to provide provenance tools to researchers as well. In this case, the ground truth tools thus far created would merely be subset of the more general tool

used throughout the entire process of analyzing the data.

A.3.1 Vistrails

Vistrails [17] is provenance engine that does both prospective and retrospective provenance analysis. It's main focus is on exploring how different parameters affect the output of different datasets. It provides a robust way of comparing and visualizing these different results. The prospective provenance information is stored in XML files, and the retrospective detailed information is stored in database on the backend.

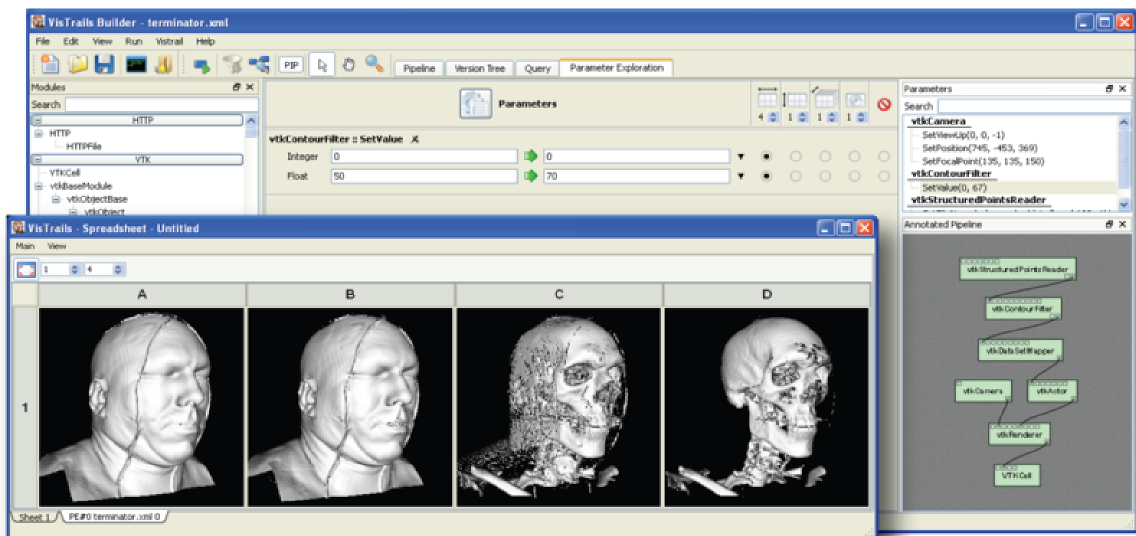

Figure A.2: Comparing parameters in Vistrails [17]

A.3.2 Taverna

Taverna [18] also records both prospective and retrospective information for a data-flow project, but does so, utilizing web services. Essentially all processes done on the data are done utilizing web services that have been setup to receive the data with certain inputs, and output a specified set of data back to the system. It has successfully been used in a wide variety of application domains.

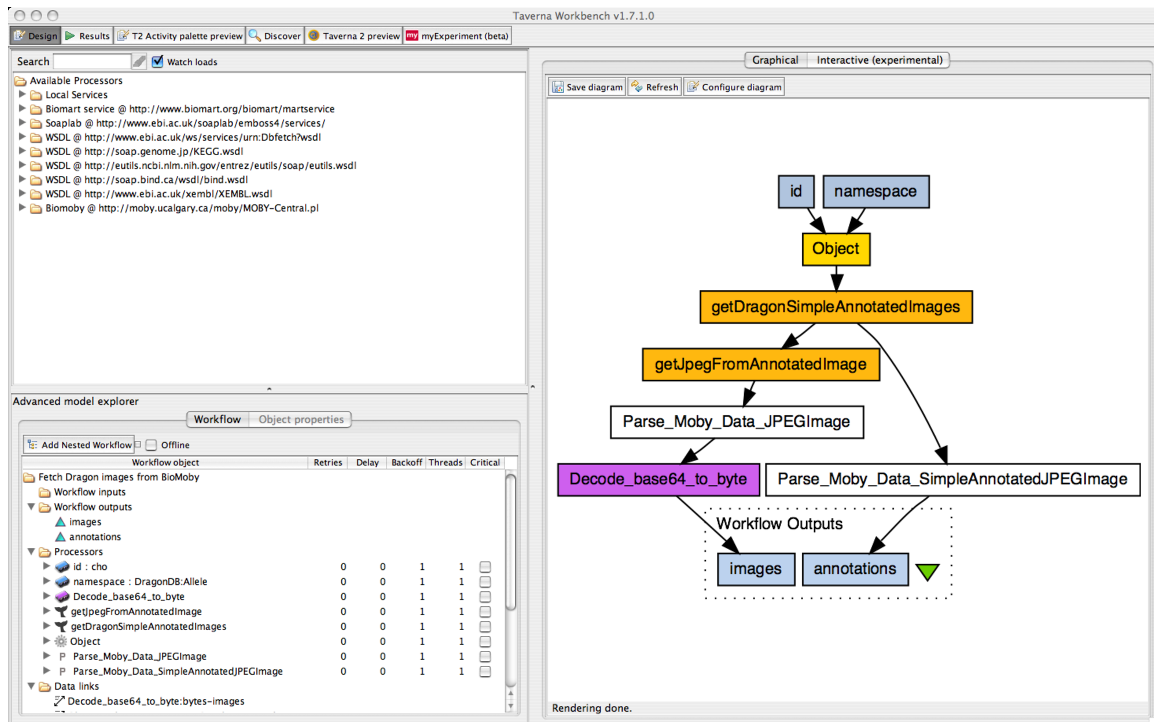

Figure A.3: Data-flow Visualization in Taverna [18]

Appendix B

Entity Relationship Diagram

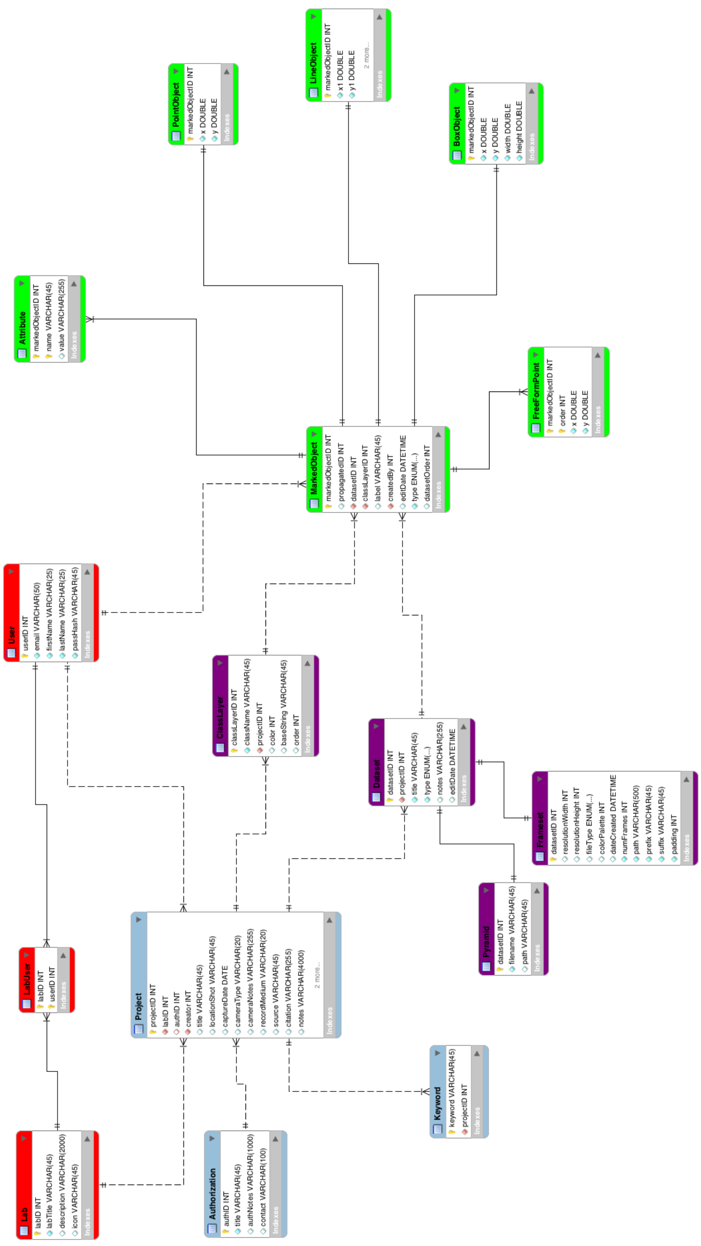

Figure B.1: Firefly Database ERD

Appendix C

Firefly User Manual

Access the Firefly system following the URL given to you. You should be prompted with a login page if you are at the right location.

C.1 Navigation

Firefly follows a fairly standard navigational interface. Up at the top, you should see a breadcrumb navigational toolbar that includes who the current user logged in is, what the current lab and project you are working in is, and the current dataset you are editing.

Log in by entering the credentials given to you:

If you currently belong to more than one lab or project, you will be prompted to select the lab that you would like to work in for the current session:

If you only belong to one lab and one project, you will automatically be taken to a list of all of the datasets that you currently have access to. You can click on a dataset to see some detailed information about it on the left hand side of the screen. You can launch the dataset by either double clicking on the row in the list, or clicking the "Launch Editor" button one the left-hand side.

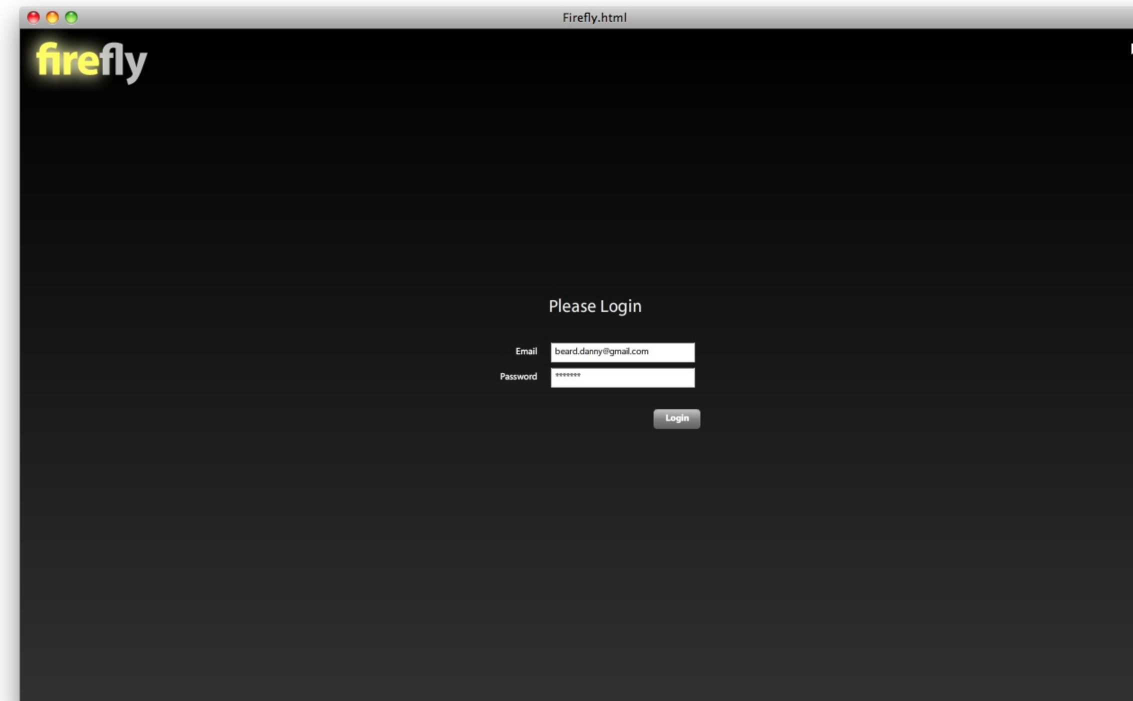

Figure C.1: Login Page

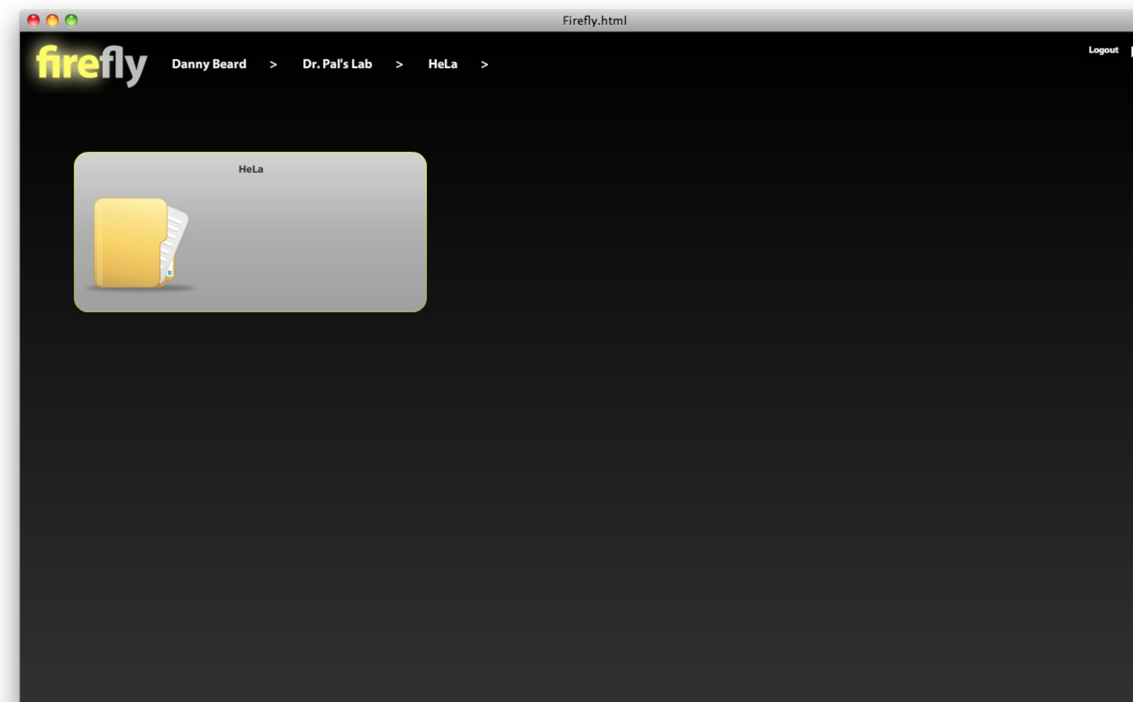

Figure C.2: Choosing a Project

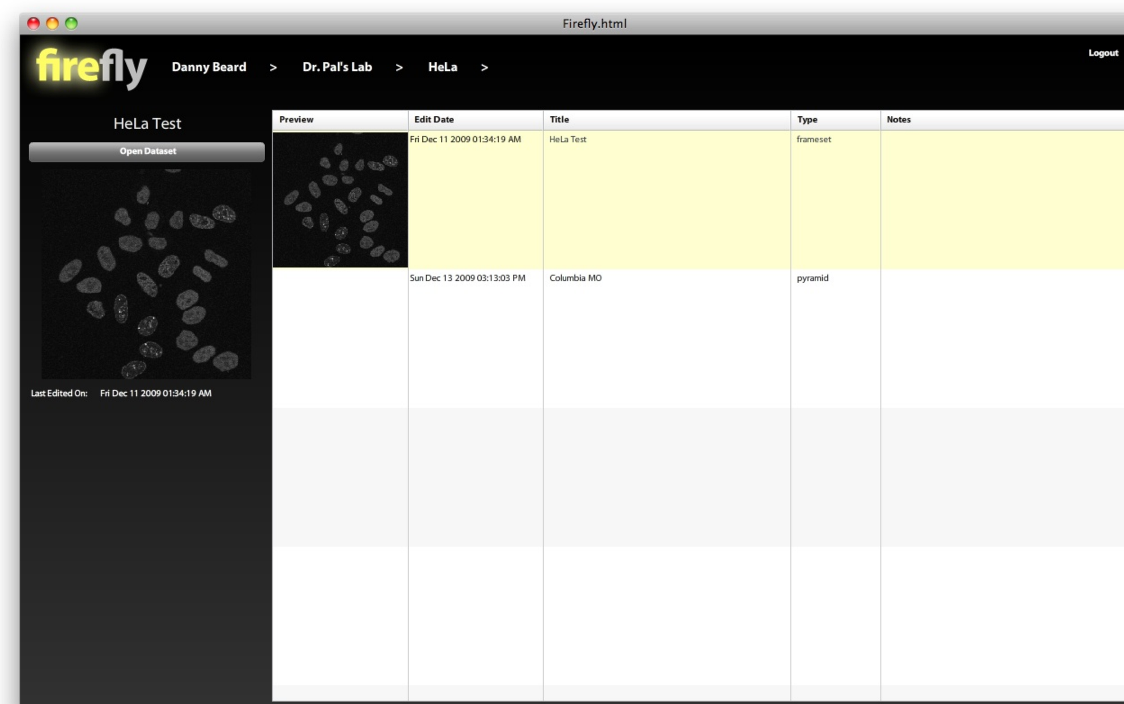

Figure C.3: List of possible datasets

C.2 Workspace

This takes you to the workspace area of Firefly. This is where all viewing of data and annotation is possible. On the left hand side of the screen is the "Control Widget". The control widget contains much of the functionality needed to work with a dataset. For example, in the HeLa dataset, we have four sections in the control widget that are important to us:

- Frame Control
- Tools
- Display/Saving
- Classes

Depending on the type of dataset being annotated, these sections are dynamic for the user. The Frame Control widget allows the user to easily change what frame is

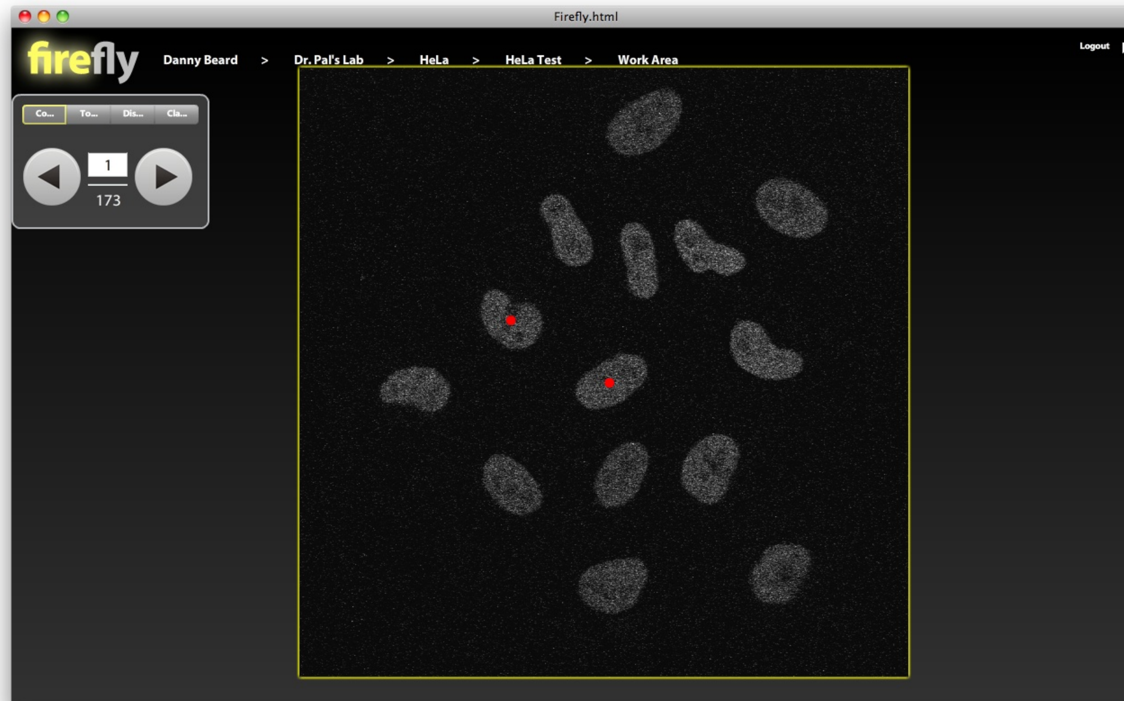

Figure C.4: Workspace view of Firefly

currently being displayed. The current frame is displayed along with the total number of frames that exist in the system. There are also previous/next controls to allow the user to easily change what frame is currently being displayed for the user.

Figure C.5: Frame control widget

The Tools section in the control widget contains all of the necessary tools for the current dataset being annotated. Again, this is dynamic based on the type of dataset the user is working with. Certain tools might not be appropriate in certain situations. The user selects the tool simply by clicking on it among the set of tools available.

How the tools are used is explained more in depth below.

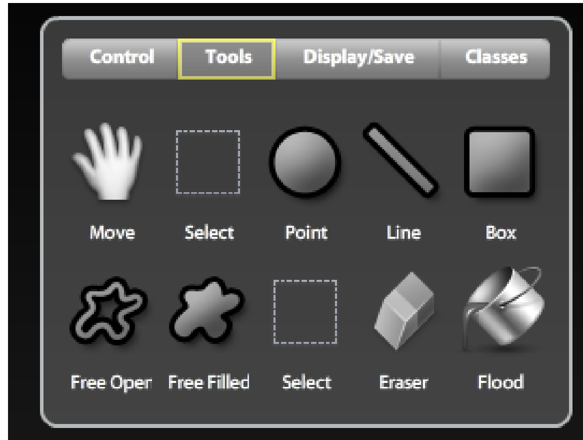

Figure C.6: Available Tools

The Display/Saving section contains a lot of miscellaneous functionality within the system including the ability to save the annotations to the dataset. The "Save Current Image" button allows the user to easily save an image to their local computer of the current frame being displayed with all of it's annotations drawn on as well. Otherwise, all currently changed data is saved whenever the user advances a frame. There is also functionality which allows the user to copy all marked object from a previous or following frame. This is important, since many times we are wanting to track objects as they progress through the frames. The changes to the objects are typically fairly small, so by copying them from a previous frame, we only have to change the attributes that have actually changed. The Display tab also contains functionality that allows the user to draw a trajectory track of all of the objects in the system. This helps the user to see where a particular object in the dataset has come from, and also where it is going. The trajectory is drawn by putting a starting frame number, and ending frame number. All movement of objects within these two frames are then drawn on the screen with a gradient based on time.

The Classes section is used to display the amount of classes that exist for the current dataset. Here, by clicking on a class, the user is selecting it as the current

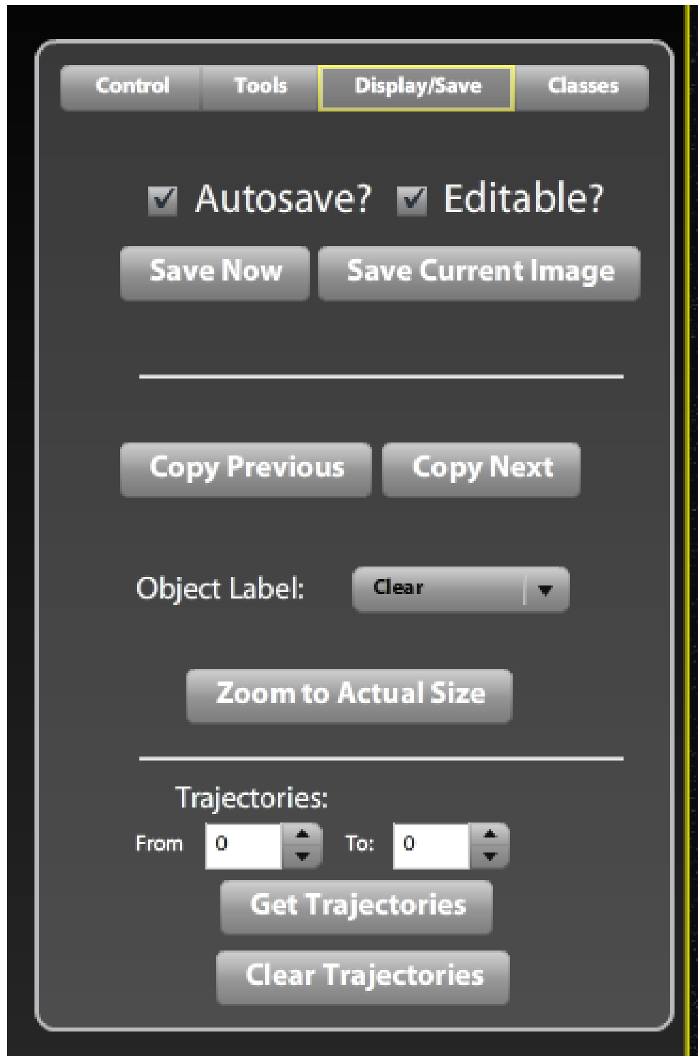

Figure C.7: Save Control

class that is being used for annotating. What this means, is that any annotating done, will by default be part of this class. There are also toggles to hide certain layers within the system. To hide or show layers, simply use the checkbox located next to the class label. There is also a master checkbox that will hide or show all layers within the dataset.

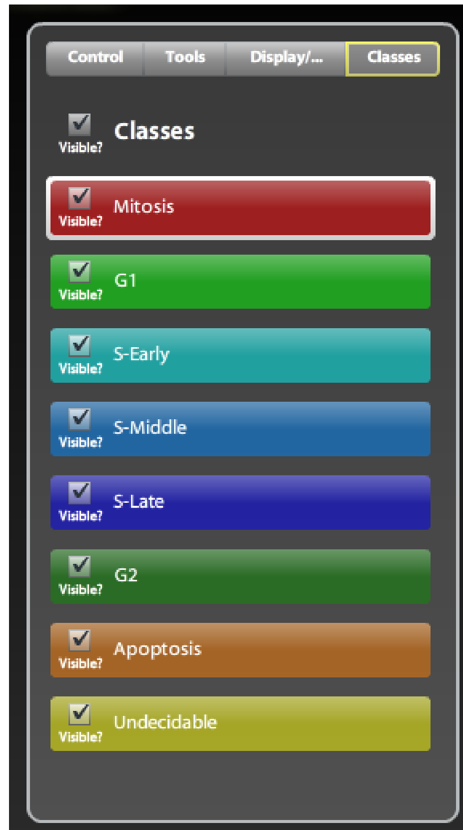

Figure C.8: Firefly Classes

Editing is fairly straightforward and intuitive for anyone who has used an image processing program before. Once a tool is selected, simply click on the dataset being displayed to add the annotation. The point tool simply drops a point where the user clicked their mouse. The line and box tools work via interactive creation where the user must click and drag to finish the creation of the object. The free-draw tool works much like a paint tool in an image editing program.

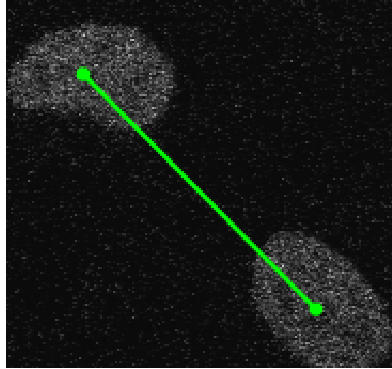

Figure C.9: Line Drawing

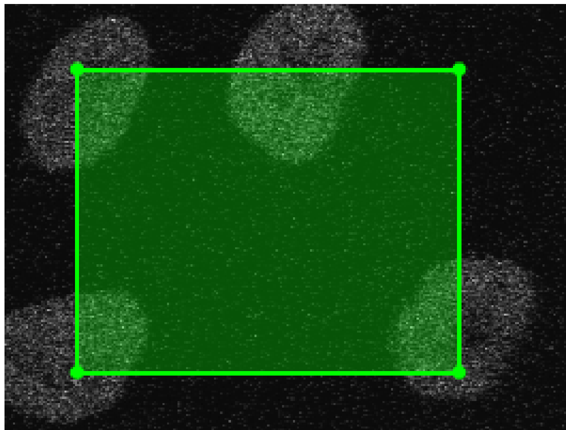

Figure C.10: Box Drawing

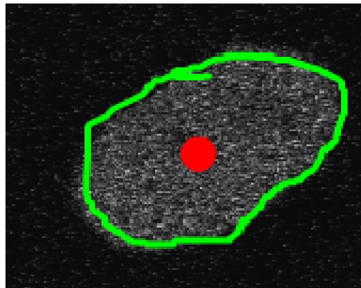

Figure C.11: Free-Drawing

The selection tool is used to select multiple objects and either move them all at once, or also delete them. All of the placed objects are also able to be interactively edited after being created this includes changing their location and size. This is done by using the control points on each object. Simply click on a point and drag to move the point around the editor.

Attributes for a particular annotation are brought up by double-clicking on the object. When the object is double-clicked a window pop ups with all attributes that can be stored for that particular object. This is specific to a particular type of dataset, and the type of object (e.g. point, line or box) being annotated. All information is immediately saved to the local cache when changes are made, so there is no need to click a save button. The information is saved to the database once the user either advances the frame, or explicitly saves the object to the database, by clicking the "Save Now" button.

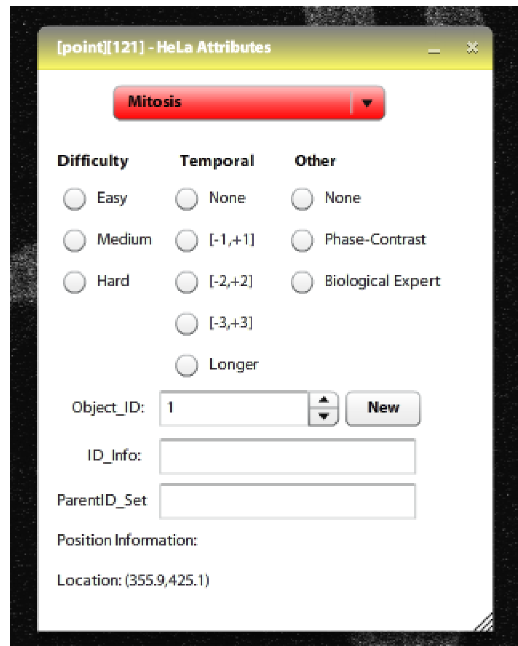

Figure C.12: Attribute Window

There are also a number of keyboard shortcuts which can be used to enhance the productivity of the user experience:

Alt + Click	Drag the editor canvas, regardless of the tool selected
Del	Delete the currently selected items in the canvas
F1	Show debug output console
F2	Put editor in minimal mode (Take away GUI elements)
Left/Right	Move through the frames in the current dataset
Shift + Number	Automatically move to a specified panel in the control widget

Table C.1: Firefly keyboard shortcuts

Once the user is done, simply close the window, or click the logout button to allow another user to use the Firefly system.

C.3 Accessing GTU and Firefly

Currently, GTU can be accessed at the following URL:

http://meru.cs.missouri.edu/GTU/?set=HeLa.TS5 - The TS5 can be changed to any of the live datasets (TS3, etc)

Firefly can be accessed here:

http://firefly.cs.missouri.edu/

Bibliography

[1] Ian Joseph Roth. Real-time visualization of massive imagery and volumetric datasets. Master's thesis, University of Missouri, 2006.

[2] Viper: The video performance evaluation resource, December 2009.

[3] E. Meijering, M. Jacob, J.-C.F. Sarria, P. Steiner, H. Hirling, and M. Unser. Design and validation of a tool for neurite tracing and analysis in fluorescence microscopy images. *Cytometry Part A*, 58A(2):167–176, 2004.

[4] Dcelliq (dynamic cell image quantitator), December 2009.

[5] Tom Noda and Shawn Helwig. Rich internet applications. *Best Practice Reports*, 2005.

[6] Jeremy Allaire. Macromedia flash mx—a next-generation rich client. Technical report, March 2002.

[7] J. C. Preciado, M. Linaje, F. Sanchez, and S. Comai. Necessity of methodologies to model rich internet applications. In *WSE '05: Proceedings of the Seventh IEEE International Symposium on Web Site Evolution*, pages 7–13, Washington, DC, USA, 2005. IEEE Computer Society.

[8] J. C. Preciado, M. Linaje, S. Comai, and F. Sanchez-Figueroa. Designing rich internet applications with web engineering methodologies. In *WSE '07: Proceedings of the 2007 9th IEEE International Workshop on Web Site Evolution*, pages 23–30, Washington, DC, USA, 2007. IEEE Computer Society.

[9] Mozilla code licensing, 2009.

[10] Ted Patrick. Flash player mental model - the elastic racetrack. July 2005.

[11] James Polanco and Aaron Pedersen. Understanding the flex 3 component and framework lifecycle. Technical report, 2009.

[12] Avraham Leff and James T. Rayfield. Web-application development using the model/view/controller design pattern. In *EDOC '01: Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing*, page 118, Washington, DC, USA, 2001. IEEE Computer Society.

[13] David Tucker. Introduction to cairngorm. November 2007.

[14] Jeremy Wischusen. Choosing a flex framework. Technical report, 2009.

[15] Open zoom - open source library, November 2009.

[16] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T. Silva. Provenance for computational tasks: A survey. *Computing in Science and Engg.*, 10(3):11–21, 2008.

[17] Vistrails: Scientific workflow and provenance management system, December 2009.

[18] Duncan Hull, Katherine Wolstencroft, Robert Stevens, Carole Goble, Matthew Pocock, Peter Li, and Thomas Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(Web Server issue):729–732, July 2006.