Expresso: Typesetting Handwritten Mathematical Expressions on the Post-PC Tablet Computer

[Project Final Report]

Josef Lange
University of Puget Sound
1130 NW 54th St #617S
Seattle, WA 98107
josef.d.lange@me.com

Daniel Guilak
University of Puget Sound
3396 Wheelock Student Center
Tacoma, WA 98416-3396
daniel.guilak@gmail.com

ABSTRACT

This paper sets forth to describe our process and products of fulfilling the capstone requirement of the Bachelor's Degree in Computer Science at the University of Puget Sound. Our project was to research, design, and implement a tablet-based software solution for the capture of handwritten mathematical expressions and compilation of its mathematical representation into an appropriate representative language (such as LaTeXor MathML).

Subjects approached in our project include image processing, artificial intelligence, application development for the Apple iOS platform (particularly in the tablet form factor), server development, as well as the adherence to a highly disciplined development cycle. We will attempt to most accurately and most briefly explain any uncommon concepts described hereafter.

1. INTRODUCTION

The utility of the touch-based computer interaction has only been recently fully realized, with the nascence of the "Post-PC" tablet in its migration away from the standard desktop usage paradigm commonly associated with computing. The modern tablet focuses on media consumption and so-called "basic" computing. In today's reality, most "Post-PC" tablets are incredibly powerful both in hardware and in software, supporting complex and challenging computations including the decoding of video, image processing, interpretation of multiple inputs, managing several network connections, and displaying high-resolution two- and three-dimensional graphics. Past their technical capabilities, these small computers have found their way into near-ubiquity of use.

The touch interface of a "Post-PC" tablet offers a unique tool to any consumer—the direct input of figures via touch—that has only previously been common to graphics and imaging professionals. Expresso sets out to take advantage of this new ubiquity in direct-interaction computing by positioning itself as an alternate input method for document production.

2. RELATED WORK

Handwriting recognition is a complex problem, only a segment of which my project wishes to solve. Existing solutions to handwriting recognition exist, and some perform at a useable level. Frequently, these systems require a period of "training" before any recognition can be achieved. Others use artificial intelligence to conclude what a sequence of handwriting is meant to be. For full-on text recognition, this is useful and almost required for any success. Commercial products exist, including but not limited to software built into Microsoft's Windows and Apple's Mac OS X Operating Systems. Additionally, third-party companies have produced software for a similar purpose, such as VisionObjects MyScript.[1] Academic projects, including those by Matsakis[3], Smithies et. al.[4], and Levin[2] exhibit valid solutions for the desktop model of usage, though are not implemented in a way that is ultimately accessible for the modern possibility of the "Post-PC" tablet as an input method, requiring the user to have an graphics tablet which is only useful for a small set of tasks.

Levin's project, CellWriter, was of particular interest to us, as it was also the work of an undergraduate computer science student. This made our goal slightly more realistic, at least to us. CellWriter, however, had some limitations which we had hoped to avoid. One, the software depended on the user writing each symbol in a predetermined cell, an option that simply is not realistic in our application. Further, this project expects one user to solely use the software so that it can learn the way *that* user specifically writes. Again, we wanted a solution more capable in terms of being useable by multiple users.

3. PROPOSED SOLUTION

This project's goal was to produce a feature-complete, usable piece of software for the Apple iOS platform, supported by a "cloud-based" application. This software would have a singular function: to capture handwritten mathematical expressions via the digitizing screen, and communicate with

[1] http://www.visionobjects.com/en/myscript/about-myscript/

with the cloud application and compile the interpreted characters into the appropriate mathematical representational language (namely LaTeXor MathML).

4. MOTIVATION & IMPORTANCE

Throughout their history, touch devices have attempted to attain status as a "universal tool" – something able to function seamlessly in their users' lives. Humans communicate naturally with handwriting, and devices from Apple Newtons, to Palm Pilots, to Pocket PCs, and now to the likes of the Apple iPad, Amazon Kindle Fire, and Microsoft Surface have tried to integrate handwriting recognition with little success since software keyboards are more familiar and efficient to most users. However, we believe that the size and shape of tablet devices are reminiscent of "slates" used in classrooms to scribble out mathematical equations.

Both authors at one point or another have been frustrated with a homework assignments requiring mathematical typesetting in LaTeX. We believe it would be great for a user to jot down an equation on an iPad or other tablet device and have the LaTeX code appear on the document they are editing on their laptop or desktop computer.

This software could easily improve the classroom environment in all levels of schooling. Getting mathematical equations into type is a frustrating task for many teachers in the K-12 levels, and even frustrating for the higher-education professor and student. With this software, student and educators alike can easily convert their glyphic mathematical thoughts into usable typesetting language. This will hopefully open doors to clearer and more specific teaching and learning throughout all levels of education.

5. METHOD & IMPLEMENTATION

5.1 Overview

A large-scale application such as Expresso requires the use of a myriad of different technologies which must all work together in harmony – the entirety of which is colloquially called "The Stack." As the authors are financially unstable university students, it was necessary to assemble a stack with little to no monetary obligation. Such a task is commonly called "bootstrapping" – a reference to the idiom "to pull one up by one's bootstraps," which we will often cite as reasons we opted to use open-source libraries and free services despite the fact that their premium equivalents generally are better documented and occasionally less buggy. The authors are also happy to release Expresso and its related applications under an open-source license as a serendipitous side-effect of the bootstrapping process.

The software components of this project stand upon a foundation of three central parts, as generally depicted in Figure 2. The primary and most visible component is the client application, Expresso, with which the user directly interacts. Expresso communicates with Barista, the "front-end" server hosted on Heroku, via a RESTful API which communicates with "Bean" and its related applications, the server which performs all of the computationally intensive work – image manipulation and symbol recognition – hosted on Amazon EC2.

5.2 "Expresso" (iOS Client)

The iOS client for Expresso is the face of its distributed computation system. The user directly interacts with Expresso on his or her iPad, iPhone, or iPod touch, and is able to interact with each step of the recognition and identification process. The application itself was developed using Apple's Xcode IDE, freely available to any who wish to develop applications.

The programming language most commonly used when developing iOS applications is Objective-C, a superset of ANSI C that gives a solid syntax for object-oriented programming. This syntax is a little bizarre at first glance, but is really powerful for rapid development of easily-readable code.

The availability of first-party frameworks from Apple for developer use is astonishing; if you need to do something you think is more than trivial, chances are there is a class and/or method for that already written. Apple's documentation, too, turned out to be an invaluable tool throughout the development process.

Even more exciting, and perhaps a bit confusing at first, was the graphical development of a GUI for these apps. While one can programattically lay out an interface for iOS, Xcode includes an Interface Builder, which on-the-fly actually instantiates UI objects and then archives them away when compiling your application, only later to be unarchived and ready-to-go on launch.

iOS development is build upon the foundation of the Model-View-Controller software architecture pattern. Each class should fall into one of these three categories, and there are guidelines as to how members of each category interact. A model class encapsulates data utilized by applications. A view, rather intuitively, is a means by which we see data, whether it is an image view, a table view, a text view, or something like a button. What remains, then, is the controller, the real hard worker of the three categories. A controller is responsible for, to some degree, "running the show". Controllers interact with model objects to see and manipulate data. Controllers, too, interact with view objects in order to either change what the view is showing or doing, or to receive information pertaining to user interaction.

The general user interaction of the most basic use case is outlined in Figure 1. The application makes use of a Navigation Controller, a special type of View Controller that can be best explained as a slide projector. It takes interaction like, "show the next UI view!", or "Go back to the start!". Naturally, one can dynamically change what the "next slide" will be, but the metaphor still stands. As the user completes a portion of the pipelined tasks of connecting, drawing, correcting, and enjoying, the Controller progresses from the current view to the appropriate next view. Behind the scenes, iOS and Apple's already-built-in magic is doing the really hard work of animating everything, as well as managing memory.

One of the more challenging elements of the iOS client app was the drawing section. One might assume that direct drawing on-screen might be available from Apple for developers to use, but no such framework is available. There is,

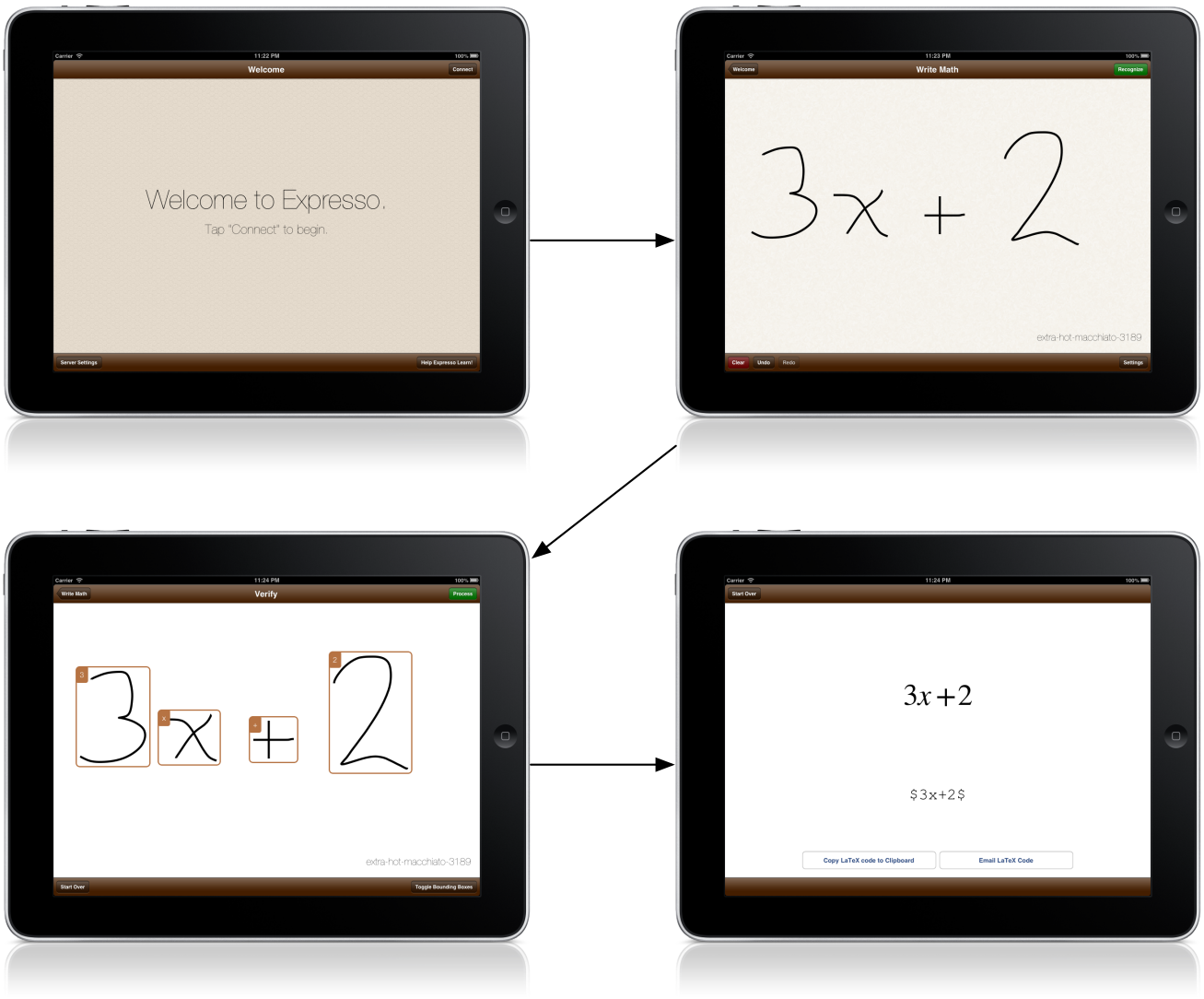

Figure 1: General User Interaction with Expresso. The user first connects to the web application. Then, they draw their math and send it off. The server returns to the client the recognized symbols. At this point, the user has the opportunity to correct any mistakes Barista and Roaster may have made. Finally, the user requests the crunched-out math and LaTeXcode.

however, the ability to detect and track a continuing touch and drag across the screen. Since the device cannot simply record every point touched down to an infinitessimal scale, iOS instead fires off an event every 1/60th of a second.

To simply draw a line segment "connecting the dots" was trashy and not very good-looking. By responding to this event appropriately, we were able to implement a curve-approximating algorithm based on a tutorial found online at "MobileTuts+"[2]. The algorithm remembers each point in the touch/drag in sequence and in doing so approximates a bezier curve given the point's values. It also re-places each point between each series of four to a smooth average of the tail and head of the previous and next curve respectively. Each curve is its own object, a `UIBezierPath`. As more curves are drawn on-screen, the performance can be significantly affected as the system needs to render and re-draw the paths on each refresh. The solution to this is to cache a rendered bitmap of all the previous curves, rendered, each step along the way.

This algorithm had some drawbacks which we had to work around. First, the user would not be easily able to draw a single point. Instead, they would have to draw a small circle on-screen, which is not ideal. A special case was added to the code to accomodate a curve within a certain threshold of size to be drawn as a filled circle of that size on-screen, instead of whatever input the user actually put in. Further, the method of maintaining only one cached image and not retaining the path objects of previously-drawn paths more or less ruled out the possibility of maintaining an undo/redo stack. To get around this, previously-drawn paths were not thrown out but instead retained in a Model object, `EXDrawing`, and used to reconstruct a rendered version of a previous or future state of the drawing.

In addition to the code adapted from the tutorial on "MobileTuts+", the iOS client utilized two open-source projects: `ASIHTTPRequest`[3] (BSD License) and `MBProgressHUD`[4] (MIT License). `ASIHTTPRequest` is a collection of classes making `HTTP` requests rather straightforward and convenient to implement in code, a necessity for this project. `MBProgressHUD` was a nice compilation of views which made displaying modal progress displays (whether determinate or indeterminate) trivial in code.

The Xcode project for Expresso comes in at just under five thousand lines of code, separated into seventeen different classes. It compiles and runs on iOS devices running iOS 5 or above, though iPads running iOS 6 are the optimal target.

5.3 "Barista"

A barista, in the coffee world, is the person who takes one's coffee order and completes and delivers one's drink. Likewise, Expresso's Barista takes requests for information and delivers said information. The most fitting method of data transmission for our use was that of a RESTful API. REST, or Representational State Transfer, "relies on a stateless, client-server, cacheable communications protocol"[1]. In essence, REST relies on the fact that no two entities are ever automatically in-sync with one another; an entity can request data from another and manipulate it, but it must send the updated data back to its source to keep changes consistent across a system. REST has, as of late, become a *de facto* standard amongst software developers looking for a server-client architecture.

REST is but an idea; there is no official standard to adhere to, nor a strict set of guidelines upon which to build one's API. Fortunately, there exist already-implemented frameworks for the rapid development of REST-minded API server. One such tool is Flask-RESTful[5], a BSD-licensed Python library that creates a framework for constructing a RESTful API on top of a web server operated by Flask[6]. Flask itself is an extremely lightweight web server, also used under the BSD license.

Barista leverages these technologies to serve and receive the data Expresso needs. Of course, every web application software requires web server hardware. For Barista, we selected Heroku[7], a mind-shatteringly simple deployment platform for web application software, particularly those programmed in Python or Ruby (we used Python). Launching an application on Heroku is literally as simple as signing up for an account and using Git source control[8] to push your code to your account on their servers. Heroku remains free to developers so long as they only run one process at a time. This, of course, presented issues when we wanted to implement recognition functionality modularily, particularly when it looked as though the recognition would need to be running a process of its own. The discussion in Subsection 5.4.1 explains our workaround to such an issue.

Barista, since it is a web server, communicates using the `HTTP` Protocol. In particular, Barista utilizes the `GET` and `POST` methods of the `HTTP` Protocol specification. Their function is rather straighforward; `GET` requests and expects to receive data, while `POST` sends and expects the remote updating of data.

Having an API defined convenient, but what about the data itself? Shouldn't that be represented in some conveniently-standardized manner? That's where `JSON`[9] comes in. `JSON` is a fantastic standardized data format. It supports singular generically-typed elements, and the basic collections of arrays and dictionaries. Utilizing a common vocabulary of keys for requesting values, several different platforms can utilized `JSON` data without requiring any transformation of the data. Where, then, is the data stored? We ended up utilizing the Redis[10] NoSQL database system (again BSD-licensed), essentially a very large map of key-value pairs. This allowed us to define our own relationships in code and define and manipulate structures of data on-the-fly. To access Redis, we utilized the Python interface to Redis, redis-

[2]http://mobile.tutsplus.com/tutorials/iphone/ios-sdk_freehand-drawing/
[3]http://allseeing-i.com/ASIHTTPRequest/
[4]https://github.com/jdg/MBProgressHUD
[5]https://github.com/twilio/flask-restful
[6]http://flask.pocoo.org
[7]http://heroku.com
[8]http://git-scm.com
[9]http://www.json.org
[10]http://redis.io/

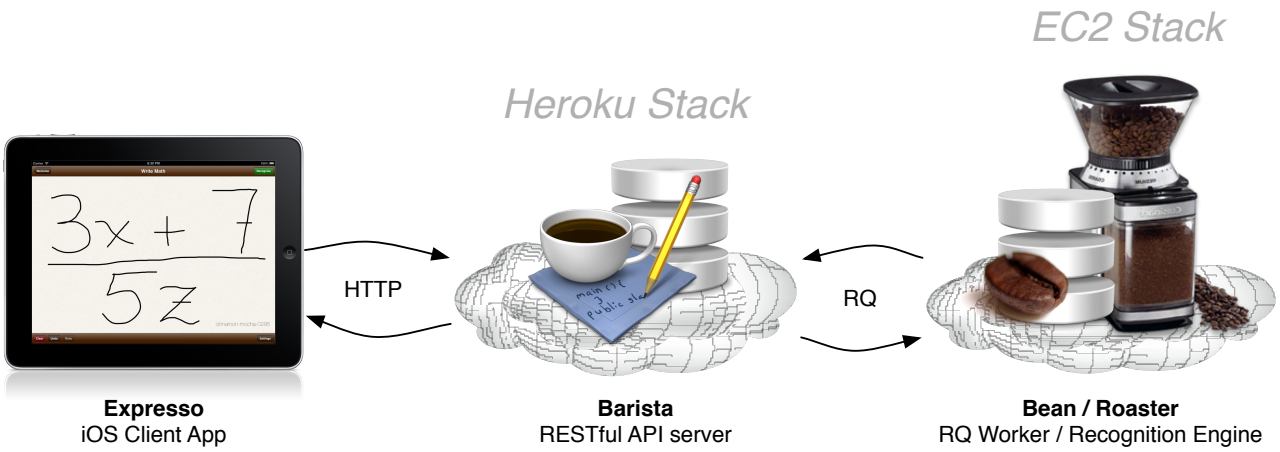

Figure 2: Simplified Organization of Software Components

py[11] (under MIT license).

So, with all this in mind, we designed and developed a two-piece system for Barista. The front-side was essentially a façade design pattern. It hooked into Flask-RESTful to implement the methods called when certain `HTTP` requests were made to the server at certain URL paths. Each of these methods utilized the Barista's back-side, a series of classes to Pythonically encapsulate the data stored in the Redis database for our use. Each class has the ability to detect whether or not it is "dirty", that is, whether or not the data the object contains is at the same state as its original data in the Redis database. This allows us to instantiate an object from the Barista module, load data into it, manipulate it, and then put the data back. It's almost a RESTful methodology in and of itself.

Barista definitely serves up a mean cup of Expresso.

5.4 "Bean", "Roaster", & "Grinder"

These three components of Expresso – Bean, Roaster, and Grinder – are grouped together because they comprise the so-called "powerhouse" of the application as they perform all of the computationally-intensive work: image manipulation and symbol recognition. We decided to host these components separately on an Amazon EC2[12] virtual server T1-micro instance rather than as additional processes on the Heroku instance which proved to be financially prohibitive, while Amazon EC2 is free for the first year.

5.4.1 "Bean"

As there is a lot of work for the back-end to complete, it was in our best interest to implement a system that would allow us to take advantage of all of the virtualized server's resources. Towards that end, we decided on using RQ[13], a Python library for implementing a priority task queue in a Redis data-structure server instance residing on our server. Barista (on Heroku) submits various different jobs to the RQ task queue on the EC2 instance, and Bean is the worker (technically a modified instance of 'rqworker') that monitors its status and pops tasks off to delegate to Roaster (image manipulation) and Grinder (symbol recognition).

There are many advantages to implementing this sort of architecture. The ability to vary priority levels for different tasks allows us to make sure that important and time-sensitive jobs (e.g., those that directly impact the user like symbol recognition) will receive computational attention before ones that are not necessarily imperative (e.g., adding additional data to the symbol recognition data set) for regular function. In addition, Bean allows for scalability as well. Multiple Bean instances can be running on the same machine (or even different machines!) to be able to pull and delegate jobs – all that is required is that they be able to connect to the Redis instance, whether locally or over the Internet.

6. EVALUATION

7. DISCUSSION & ANALYSIS

Throughout the process of development, we ran into several challenges, some of which we successfully worked through or around, others we did not. Among these challenges were character recognition, image processing, poorly-documented resources, and a proclivity for getting "hung up" on a particular issue.

Naïvely, we assumed that the smaller set of symbols to be recognized (digits and operators as opposed to letters) would make the recognition process easier. While it is true in some sense, the ease it gains is in increased accuracy of a working recognition engine. We didn't have one of those. We shuffled through several recognition engines and implementations of various forms of Artificial Intelligence, none of which seemed capable of successfully identifying symbols. This, obviously, caused a large issue that cascaded into other segments of the project. Because we yearned so desperately to have recognition work, we found ourselve essentially wasting time trying to get recognition to work instead of moving on to other

[11]https://github.com/andymccurdy/redis-py
[12]http://aws.amazon.com/ec2/
[13]http://python-rq.org/

components of the project.

The image processing involved in preparing an image for recognition is suspect to some of our problems. Neither of us were particularly familiar with the technique behind image encoding and still know very little. Our use of `OpenCV`[14] and `PIL` (Python Imaging Library)[15] was guided by poor documentation and, most of the time, wild guesses.

8. CONCLUSION

9. REFERENCES

[1] M. Elkstein. Learn REST: A Tutorial: 1. What is REST?, 2008.

[2] M. Levin. *CellWriter: Grid-Entry Handwriting Recognition*. PhD thesis, University of Minnesota, Minneapolis, MN, USA, 2007.

[3] N. E. Matsakis. *Recognition of Handwritten Mathematical Expressions*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1999.

[4] S. Smithies, K. Novins, and J. Arvo. A handwriting-based equation editor. Technical report, Department of Computer Science, University of Otago, Dunedin, New Zealand, 1999.

[14] http://opencv.org

[15] http://www.pythonware.com/products/pil/