UNIVERSITY OF CALIFORNIA,
IRVINE

Sensor Plot Kit: An iOS Framework for Real-time plotting of Wireless Sensors

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Electrical and Computer Engineering

by

Derrick Allen Lo

Thesis Committee:
Professor Pai H. Chou, Chair
Professor Rainer Doemer
Professor Brian Demsky

2015

DEDICATION

To my parents, Betsy and Simon.

TABLE OF CONTENTS

Page

LIST OF FIGURES **v**

ACKNOWLEDGMENTS **vi**

ABSTRACT OF THE THESIS **vii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Contributions . . . 2

2 Related Work **4**
2.1 Sensor Data Capture . . . 4
2.2 Sensor Data Visualization . . . 5

3 Problem Statement **8**
3.1 Requirements . . . 8
3.1.1 Functional Requirements . . . 8
3.1.2 Performance Requirements . . . 10
3.2 Objectives . . . 11
3.2.1 Performance . . . 11
3.2.2 Qualitative . . . 12

4 Technical Approach **13**
4.1 Problems with Model-View-Controller Design . . . 13
4.1.1 Core Plot Framework . . . 14
4.1.2 MVC Architecture . . . 15
4.2 MVVM Solution . . . 16
4.2.1 MVVM . . . 16
4.2.2 Advantages of MVVM . . . 18
4.3 Architecture . . . 19
4.3.1 SPKModel . . . 19
4.3.2 SPKDataSource . . . 20
4.3.3 SPKController . . . 21
4.3.4 SPKPlotView . . . 22
4.4 Design Space Exploration . . . 23

4.5 Development Process . . . 23
4.5.1 SPKPlotViewECG Class . . . 23
4.5.2 SPKFilter Class . . . 24
4.5.3 SPKDataStore Class . . . 25
4.5.4 SPKBuffer Class . . . 25
4.5.5 Demo Application Development . . . 26
4.5.6 Migration to MVVM Architecture . . . 27
4.5.7 Performance Improvements . . . 27

5 Experimental Results **29**
5.1 Metrics . . . 29
5.1.1 Latency . . . 30
5.1.2 Sensor Data Rate . . . 30
5.1.3 View Refresh Rate . . . 30
5.2 Experiment Setup . . . 31
5.2.1 Case Study 1: Single Sensor Data Stream . . . 31
5.2.2 Case Study 2: Wireless Multi-lead Sensor Data Stream . . . 31
5.2.3 Case Study 3: High Sample Rate Sensor Data Stream . . . 31
5.3 Performance Results . . . 33
5.3.1 Case Study 1: Single Sensor Data Stream . . . 33
5.3.2 Case Study 2: Wireless Multi-lead Sensor Data Stream . . . 35
5.3.3 Case Study 3: High Sample Rate Sensor Data Stream . . . 35
5.3.4 Evaluation . . . 38
5.4 Qualitative Results . . . 40
5.4.1 Reusability . . . 40
5.4.2 Scalability . . . 40
5.4.3 Minimal Development Time . . . 41

6 Conclusion **42**
6.1 Summary . . . 42
6.2 Future Work . . . 43

Bibliography **44**

LIST OF FIGURES

Page

2.1 Views of the Sensor Kinetics App 6
2.2 Screenshot of the Graphical Analysis application 7

4.1 MVC Architecture 14
4.2 MVVM Architecture 17
4.3 Sensor Plot Kit Architecture 19

5.1 Screenshot of 12-lead ECG sensor streaming on the iPad 3 demo application 32
5.2 Performance Results: Single Sensor Data Stream 34
5.3 Performance Results: Wireless Multi-lead Sensor Data 36
5.4 Performance Results: High Sample Rate Sensor Data Stream 37
5.5 Graphics Software Stack on iOS [7] 39

ACKNOWLEDGMENTS

I would like to thank my Advisor, Dr. Pai H. Chou, for all of his support, inspiration, and guidance throughout my research. His thoughtful perspectives and insight has allowed me to become a more effective researcher. This thesis would have not been the same without his enthusiasm and passion towards important research problems.

I am especially thankful of the loving support of my parents, Betsy and Simon. I would have not been able to succeed without the sacrifices they have made. I am also thankful for the support and kindness of my sisters, Carissa and Stefanie.

I would like to especially thank Mai for her continuing love and support throughout my research. I am especially grateful for her constant encouragement and comfort.

I am very grateful to my colleague, Ting-Chou (Brett) Chien, who helped me get acquainted with my research and has been instrumental in making it possible.

ABSTRACT OF THE THESIS

Sensor Plot Kit: An iOS Framework for Real-time plotting of Wireless Sensors

By

Derrick Allen Lo

Master of Science in Electrical and Computer Engineering

University of California, Irvine, 2015

Professor Pai H. Chou, Chair

This thesis presents a new open-source framework for plotting and managing streams of sensor data on iOS devices. The framework, called the Sensor Plot Kit (SPK), aims to help developers build apps for viewing sensor data that can be streamed in real-time or pre-recorded. It can handle multiple streams at relatively high data rates with short latencies, making it suitable for advanced medical applications such as electrocardiograms (ECG). This framework also supports filtering, post processing, and saving the data in non-volatile storage. A novel feature of our design is that we adopt the Model View View-Model (MVVM) design pattern that reduces complexity and maximizes code reusability compared to the standard Model View Controller (MVC) pattern in iOS while remaining compatible. Experimental results using a real-world ECG, on-board sensors, and pre-recorded results over actual and simulated iOS devices confirm that SPK enables developers to build feature-rich, robust apps that embed real-time, responsive plotting capability while significantly shortening development time. This work has the potential to make a crosscutting impact on science, engineering, medicine, the financial market, and many other fields that increasingly rely on smart phones and tablets as the primary device for viewing and interacting live and historical streams of data.

Chapter 1

Introduction

There has been a dramatic increase in development and demand of software applications for mobile computing devices. Advancements in technology has allowed these smart devices to include a wide array of embedded sensors such as an accelerometer, gyroscope, GPS, microphone and the ability to wirelessly connect to external sensors such as an electrocardiogram (ECG) or heart rate monitor (HRM). These sensing technologies have enabled the potential for new frameworks and applications to arise in all types of industries such as healthcare, automotive, logistics and manufacturing [13]. Apple alone has sold over a billion smart phones and tablets, allowing mobile software applications to broadly impact the masses through the accessibility and technology of these devices.

1.1 Motivation

As an increasing number of scientific, engineering, medical, and even financial applications require data plotting features in their mobile apps, it is surprising that the development of such a universal framework has received very little attention. A Dartmouth Survey on Mobile Phone Sensing

has stated that, “Although the potential of using mobile phones as a platform for sensing research has been discussed for a number of years now, in both industrial and research communities, there has been little or no advancement in the field until recently” [13]. The lack of a universal sensing framework has created an abundance of functionally similar sensor applications that are only compatible with one software platform.

Developers today are highly limited to either implementing their own customized application or paying for a private API to visualize and analyze sensor data on iOS devices. This has forced developers into duplicating effort across sensor applications and platforms. Without a standard framework, developers may fall into the same pitfalls of scalability, programability and performance limitations during development. Contrary to popular belief, programming sensor driven applications on iOS can be extremely complex and time consuming even when adhering to the commonly used Model View Controller (MVC) design pattern. Without the proper considerations, application modules can be overly customized, hard to unit test and susceptible to errors. A well-designed universal sensor framework is critical in reinforcing clear design patterns and enabling new applications with minimal development time.

1.2 Contributions

In this thesis, we reveal a progressive iOS framework for real-time sensor data visualization that is novel in design and far reaching in its impact. This highly extensible framework provides a universal approach to interfacing with diverse types of sensors and the ability to visualize multiple streams of sensor data simultaneously at relatively high data rates with short latencies. Our purpose was to create a well-structured and easy-to-use framework for developers through the novel use of the Model View View-Model (MVVM) design pattern. Our experimental results have established the viability of our MVVM based framework by demonstrating reusability, testability and extensibility of the core framework modules. Additionally, we provide developers with a test

bed to profile and tune the performance aspects of the framework. This test bed has allowed us to determine and characterize the hardware performance limitations among the iPad and iPhone devices. There has not been any other framework capable of enabling developers to create quality sensor plotting applications that are performance and design driven to significantly reduce development time. Our work will not only impact developers working on future mobile platforms for data visualization but will also empower authors of existing applications to adopt and benefit from our framework.

Chapter 2

Related Work

At a high level, sensor data platforms manage the processing of sensor data through three main steps: sensor data capture, data processing, and data visualization. Sensor data capture entails the acquisition, buffering, and transferring of the raw sensor data as it is received in real time or through a storage medium. The data processing step, which is often optional, includes the filtering and preprocessing of the raw sensor data. Finally, in the data visualization step, the raw or processed data is rendered onto the mobile device's screen as a graph or a basic numeric reading. In this chapter, we survey existing frameworks that are within the sensor data platform space.

2.1 Sensor Data Capture

SensingKit is exceptional example of a sensor data capture framework that is capable capturing sensor data continuously from all the embedded sensors on a mobile phone. The framework is an open-source multi-platform library that enables developers to easily extend the framework to include other sensors or functionality [11]. SensingKit provides a broad API to interface with embedded sensors; however, they omitted the option to filter, preprocess, and most importantly,

visualize the sensor data on the mobile device. SensingKit intended users to rely on external tools to analyze and visualize the raw sensor data. While the framework has great potential to interface with a large set of sensors, it ultimately lacked any functionality after sensor data capture.

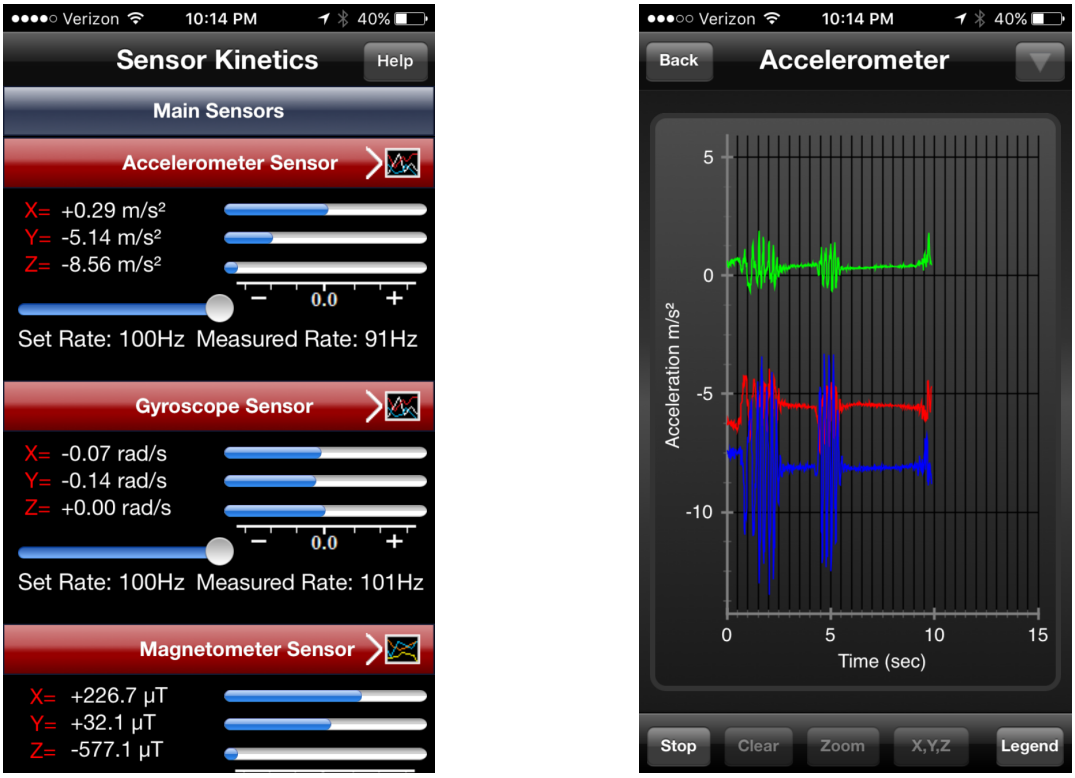
Our proposed Sensor Plot Kit framework was motivated by this unmet need for functionality to visualize captured sensor data within the mobile device platform. Similar to SensingKit, we intend to provide a universal interface to different types of sensors but beyond the scope of just sensors internal to the mobile device.

2.2 Sensor Data Visualization

Sensor Kinetics and Graphical Analysis are both iOS applications that allow users to capture and visualize sensor data.

The Sensor Kinetics app, as seen in Figure 2.1a, visualizes in real time the embedded sensors within the iPhone or iPad [10]. Users can interactively adjust the sampling rate and click the individual sensor for a graphical plot of the data as shown in Figure 2.1b. In addition, in the paid version of the application, users can filter, record, and share data from these sensors. Sensing Kinetics overall is a very precise and information rich application.

Graphical Analysis is developed by Vernier and is marketed towards students in STEM education as a tool to wirelessly collect, analyze, and share sensor data in the classroom [17]. They exclusively interface with only Vernier branded sensors (e.g., thermometers, heart rate, pH, motion, and force sensors) and some of the iPhone's embedded sensors such as the accelerometer. The app can plot up to 3 graphs simultaneously as well as record and export the sensor data. One of the unique benefits features of the application is that users can interactively analyze and make annotations on the graphs as shown in Figure 2.2.



(a) Main Sensor Data view

(b) Plotting of the Accelerometer

Figure 2.1: Views of the Sensor Kinetics App

There are, however, several common pitfalls among the Sensor Kinetics and Graphical Analysis applications. Both these applications are closed-sourced projects that provide no documentation or source code, which makes these platforms unextensible for anyone else but the authors. Another limitation is that users only have an option of viewing up to one to three sensor data streams at a time. These pitfalls not only make it impossible to customize or expand the functionality of the applications, but it also prevents developers from leveraging or reusing the sensor interfaces and features that these applications all share. Our proposed Sensor Plot Kit is designed to provide developers with the freedom of extensibility and customization across all parts of the sensor data platform.

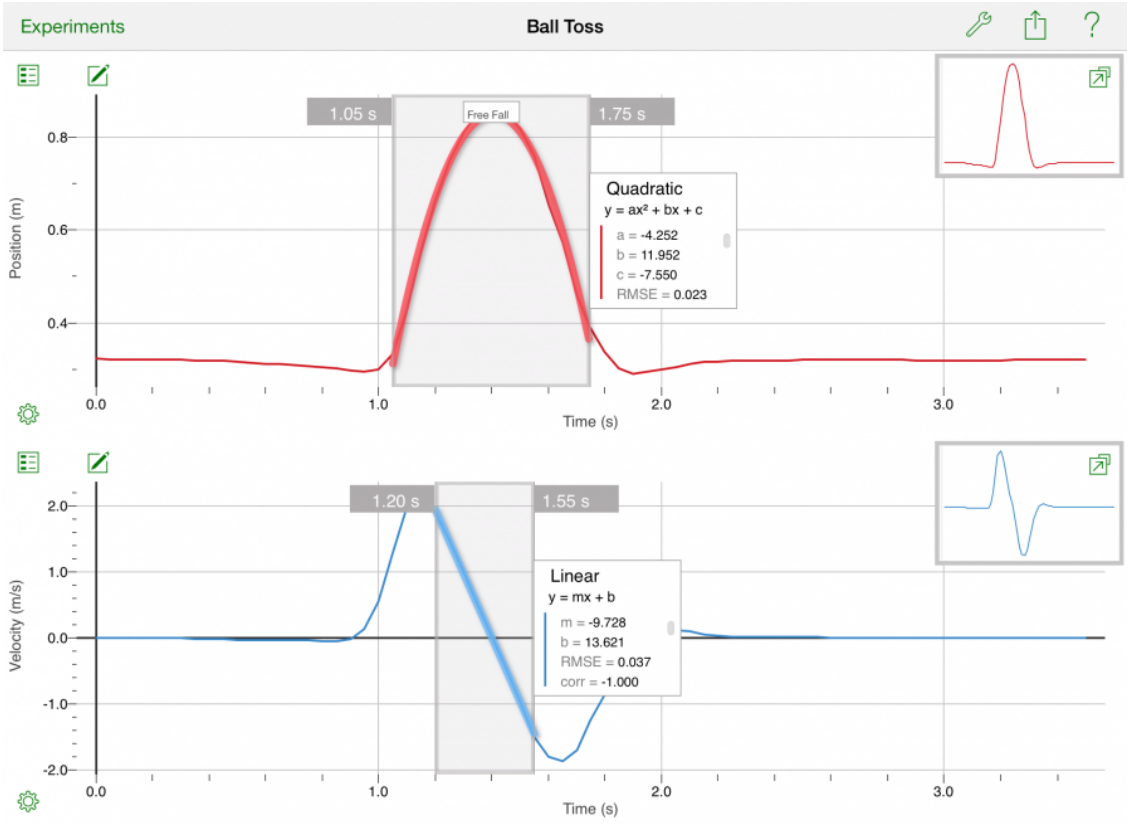


Figure 2.2: Screenshot of the Graphical Analysis application

Chapter 3

Problem Statement

The purpose of this work is to create a plotting kit for sensor data that can be live-streamed or pre-recorded, at a given rate and with minimal latency. This chapter states the requirements in more formal terms and the objectives.

3.1 Requirements

Requirements are the features that must be implemented or properties that a correct implementation must have. They can be divided into functional requirements and performance requirements.

3.1.1 Functional Requirements

The functional requirement can be characterized by the input-output characteristics of the plotting framework.

Input

The framework receives incoming streams of raw sensor data either from the sensor directly or through reading the data from a file. There can be a number of streams (n) of data within a single sensor. Every stream of data must be sequentially ordered relative to the time (t) when they were produced by the sensor. The first bits of sensor data produced will be the first bits fed into the framework. The framework expects that the data will be pushed in at the real rate, sensor data sampling rate (s), at which they are produced. In the case that data has been pre-recorded, it is important to note that the data storage medium must be able to provide data to the framework at the same rate at which it was captured or faster. Let N_{input} denote the aggregate number of samples of data the framework will receive over time. It can be defined by the equation

$$N_{\text{input}} = n \times s \times t \tag{3.1}$$

Output

The output of the framework is a time-history rendering of the input streams of sensor data onto the screen of an iOS device. In addition to rendering the data points, our framework is also required to support the application programming interface (API) for the appropriate axes and labels on the screen to provide a meaningful reference to the sensor data.

Moreover, as this work targets specifically the iOS platform, the CorePlot framework in iOS is utilized to help properly frame and further abstract the plot details to be drawn. The output of our Sensor Plot Kit is considered up to the point to where the CorePlot framework in iOS is called to request the rendering of data.

There is possibility for expansion of the framework to allow the output to be sent to other functions

or devices. The framework is required to allow the data to be output to a data structure such as an array to allow further processing in the framework or usage by the developer.

Interactivity

The SPK should support developers to build interactive apps, in addition to providing real-time or static renderings of sensor data. In the real time scenario, the framework will present the user a window into the sensor data stream at the real rate that it is being sampled. Users will be able to pause the live data and resume the stream from where it left off in part to the buffering of the sensor data. In contrast, the static rendering provides a window into a subset of prerecorded data. Data can be auto streamed or played back to simulate a real-time stream of sensor data. In both scenarios the size of the window can be adjusted to allow users to view more data within the screen.

3.1.2 Performance Requirements

Performance requirements include both the latency and the view refresh rate. Latency is the delay from the input sample to the corresponding output. View refresh rate is the rate at which the output can be rendered on the screen.

Latency

The framework must uphold real-time streaming of sensor data by maintaining a bounded latency l from when sensor data is received to when it is requested to be rendered. There is a practical limit at which streaming is considered real time, and any observed latency beyond that limit will be considered unusable. We determined from early experiments that any latency beyond 1 second (1000 ms) is considered unacceptable by the user depending on the domain of the application. There are domains that can tolerate a higher latency such as lower frequency temperature based

applications; however, we are not concerned with those domains and are targeting applications that consider 1 second a general upper bound.

View Refresh Rate

The view refresh rate is the number of times per second that a display updates the contents of its buffer. There is no reason to render at a rate higher than the maximum view refresh rate of a device's display as the user will never see the extra frame buffer updates. iOS devices such as the iPad and iPhone displays have been reported to have a maximum refresh rate of 60 Hz. The framework must be able to maintain a stable view refresh rate to prevent any lag during rendering.

3.2 Objectives

Objectives are the cost functions to be maximized or minimized by the design. They are ways to measure how good a design is.

3.2.1 Performance

Performance is the main quantitative objective. That is, the framework should aim to achieve the highest refresh rate supported by the device's display and the lowest latency, although it may be impossible to achieve both. Further, achieving a low latency is paramount over a higher refresh rate. Without a low latency, the system as a whole will lack sufficient performance and attainment of a higher refresh rate would be fruitless. However, our performance objective is more general because we also want to support a wide range of hardware. This means we will need to characterize the performance trade-offs of rendering sensor data at different sampling and view rates on different types of iOS hardware. Developers are typically not fully aware of these performance trade-offs

until their application code is complete and tested. We intend for developers to leverage these cost functions in their application designs so that they can create quality applications that will endure minimal latencies and maximum sensor sampling rates. Furthermore, we hope to provide insight on the limitations of iOS hardware to aid in determining hardware requirements.

3.2.2 Qualitative

Qualitatively, we aim to maintain three characteristics of programability: reusability, scalability, minimal development time. Reusability will allow modules of the framework to be used in multiple diverse applications. Reusability not only allows new applications to be deployed faster but also enables existing applications to adopt this framework as well. Scalability is key as it provides developers with flexibility in visualizing multiple sensors and extensibility to build powerful solutions utilizing the processed data from the framework. Lastly, minimal development time is key as the intention of the framework is to aid the developer in bring up of complex sensor data visualization rather than impeding or distracting developers from application design.

Chapter 4

Technical Approach

For iOS app programming, Apple recommends that developers adopt the model-view-controller (MVC) pattern of organizing their program code. To be compatible with the development of most apps, our proposed SPK should also support a similar style. However, a straightforward MVC model that builds on top of the Core Plot framework in iOS results in an overly complex view controller that is difficult to reuse. We solve the complexity problem by adopting a Model View View-Model (MVVM) style. This chapter first provides a background on the problems with the MVC pattern.

4.1 Problems with Model-View-Controller Design

Model-View-Controller (MVC) is the most commonly used design pattern within iOS as the majority of Apple's frameworks follow this paradigm. Apple defines MVC as "a design pattern that assigns objects in an application one of three roles: model, view, or controller" [4]. It requires that every class in an application maintain one of these roles. This is to promote code separation, reusability, extensibility and a well-defined interface. In addition, each role is subject to how they

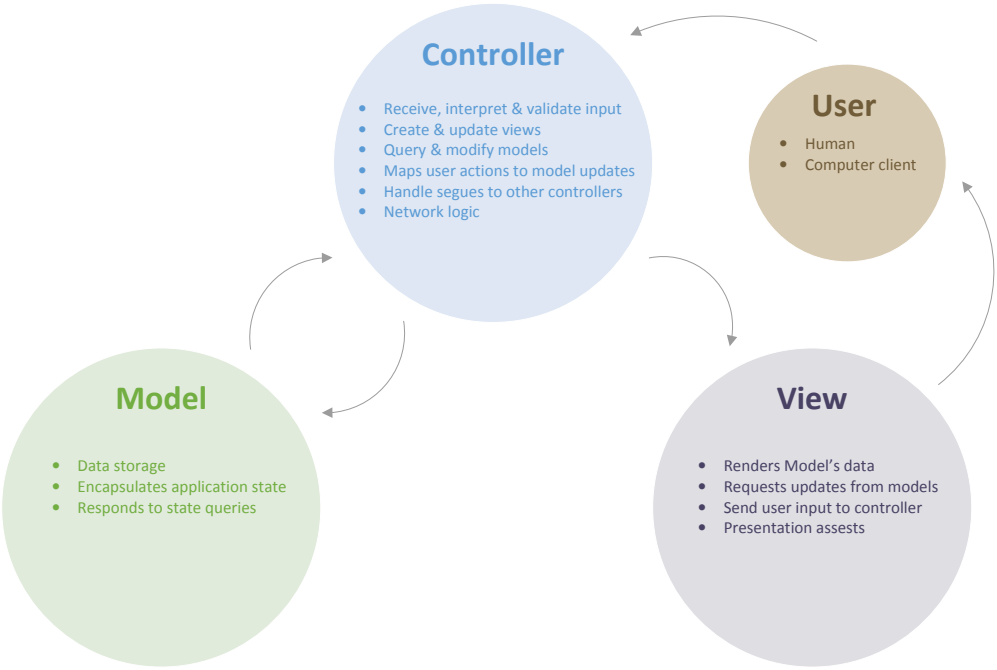


Figure 4.1: MVC Architecture

can communicate with the other MVC objects. Figure 4.1 is an overview of the MVC Architecture and highlights the typical responsibilities of each role.

Our main design objective was to help manage the lengthy user interface (UI) setup for developers as well as provide a clear interface for sensor data capture. By decoupling the front-end UI from the backend sensor data capture, the sensor plotting process can be vastly simplified.

4.1.1 Core Plot Framework

Core Plot is the de-facto open source library on iOS for building graph based applications [15]. It is a powerful and customizable framework that is capable of plotting a wide variety of charts

and graphs. It is can used for creating static 2D bar and pie charts, scatter plots and graphing mathematical functions. Drawing within the framework is executed through Apple's Quartz 2D drawing engine. Although Core Plot provides the functionality to create real time animated plots, we have observed that the animations do not scale up well with higher data rates.

We could build our own plotting library from the ground up that would give us more control over the design. However, it probably would not be as efficient, robust, or as evolvable over future versions of the OS. In contrast, Core Plot has been proven to be extremely flexible and powerful for plotting scatter plots and other types of graphs.

However, despite its popularity and exposure, the library requires significant setup in code and a deep understanding in order to use the library effectively. Not only is utilizing the library inherently difficult and time consuming but building a strong architecture around Core Plot has also proved to be challenging. This is further exacerbated by Apple's MVC design paradigm.

4.1.2 MVC Architecture

In the MVC design specification, Apple has a very broad definition of the Controller's purpose. Apple's documentation specifies that "View Controllers are traditional controller objects in the MVC design pattern. Their job is to provide many behaviors to iOS apps" [4]. This loose definition causes developers to put the majority of their application logic in the View Controller class.

The View Controller should focus on only acting as a link between model and view and not carry any "business logic" [9]. The problem only gets worse when applications involve complex external frameworks such as Core Plot and frameworks that handle network connectivity such that the View Controller is usually the most fitting class as these frameworks are strictly out of the scope of what Apple defines for Model and View patterns. Furthermore, the View Controller ends up becoming the data source and storing data rather than the actual model object [5]. Adhering to the MVC

paradigm can cause adverse effects as it ends up creating massive view controllers that are complex and not re-usable within the application, resulting in duplication of code and the code to become more error prone.

4.2 MVVM Solution

To solve the complexity of implementing Core Plot within the MVC paradigm, we adopted the Model View View-Model (MVVM) design pattern. MVVM separates the concerns within the View Controller into separate classes to reduce complexity and maximize their reusability [9].

4.2.1 MVVM

Because View Controllers are typically linked to distinct View classes, MVVM pairs the View and View Controller components together. Now the logic responsible for preparing data for the View that is normally found in the View Controller is being moved to the View Model class. The View Model sits between the Model and the View/ViewController classes [5]. The diagram in Figure 4.2 describes the MVVM architecture and how the View Model fits into the new design.

View Controller Model

The View Controller is now solely responsible for the event handling logic of the UI and defining the connections between the View, View Model, and Model.

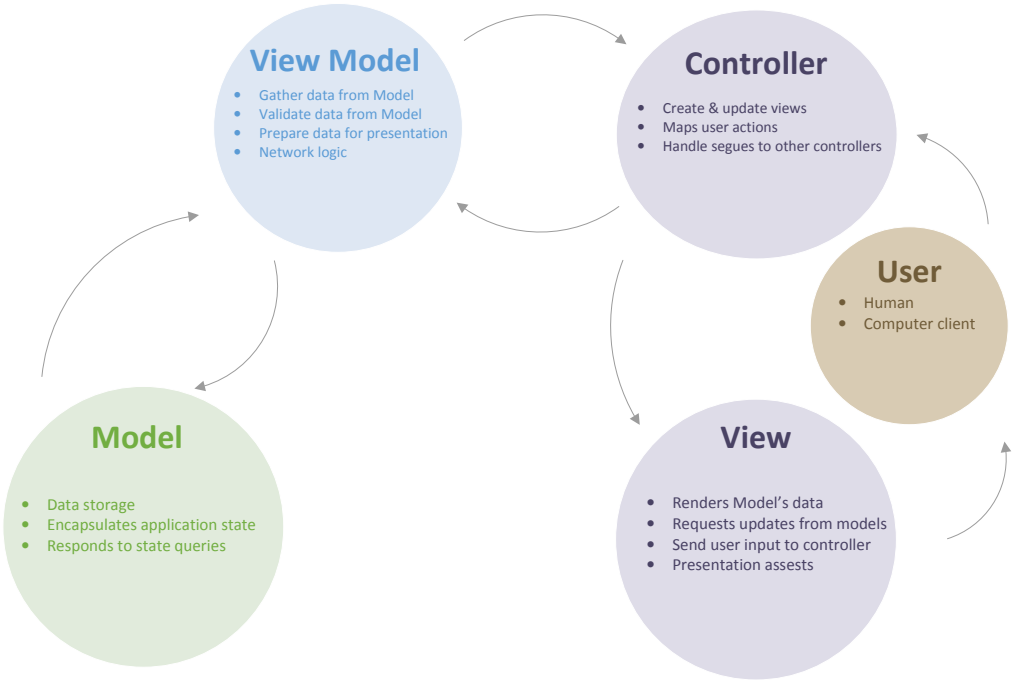


Figure 4.2: MVVM Architecture

View Model

The View Model class will handle the presentation logic for the View Controller. The class basically ensures that all the data coming from the model is properly type cast and formatted so that it can be used directly by the View. For example, the Model may have its values formatted as an NSDate, but the View class expects the data to be an NSString variable [8] . In addition, network logic can be placed within the View Model reducing the clutter in the Controller.

Model

Lastly, the Model class maintains control of the storage and persistence of the data used throughout the application [8].

4.2.2 Advantages of MVVM

The advantage of MVVM over MVC is that it improves testability, reusability and compatibility. Testability is improved as it allows developers to easily test functions in the View Model without affecting the core View Controller logic. Unit testing within MVC designs is difficult because the View Controller is responsible for so many aspects of the application. Data validation and UI logic in Controller can intertwined within the same functions making it sometimes impossible to test without changing the code.

Reusability of the Controller and View classes is facilitated by MVVM as different data sources can now be defined in the View Model and Model classes without the need to make any significant changes to the View and Controller.

Lastly, MVVM is compatible with existing MVC applications as it is just an expanded abstraction of the MVC design.

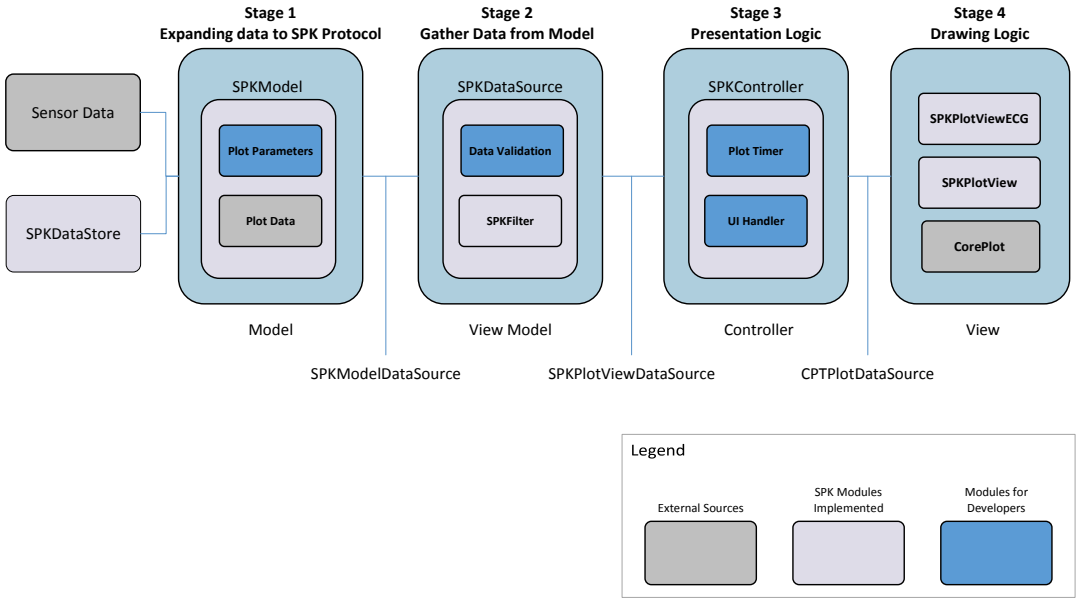


Figure 4.3: Sensor Plot Kit Architecture

4.3 Architecture

The Sensor Plot Kit (SPK) framework architecture is made up of four main classes: `SPKModel`, `SPKDataSouce`, `SPKController`, and `SPKPlotView`[14]. They correspond directly to the Model, View Model, Controller, and View design pattern of MVVM. Figure 4.3 illustrates the high-level architecture and describes the data flow and relationships between the classes.

4.3.1 SPKModel

The `SPKModel` class is responsible for capturing and buffering the incoming raw sensor data from the sensor. There are numerous ways an application developer can connect wireless sensors and

collect sensor data streams to iOS devices, so this class was intended to be the main interface to the external sensors.

After buffering the sensor data, the data is sent to the SPKDataSource using the data source design pattern [1]. To provide `SPKDataSource` with data using this approach, the `SPKModel` class is designated as the datasource for `SPKDataSource` within `SPKController`. The interface between `SPKModel` and `SPKDataSource` is defined under the SPKModelDataSource protocol. This protocol is declared within `SPKDataSource` and contains two method prototypes, `dataFromSensor()` and `plotParamsFromModel()`. The `SPKModel` class is responsible for implementing these method prototypes in order to adhere to the SPKModelDataSource protocol.

The two methods, independently push the raw sensor data and plot parameters to the `SPKDataSource`. The raw sensor data is pushed one datum at a time as it is received from the sensor. The plot parameters are only pushed once, as the plot does not dynamically change once it has been created by Core Plot. The parameters are used to setup the graph's sampling rate, plot name, duration of view, plot scale, line color, and plot ranges.

The `SPKModel` class was designed to be reusable for different sensors as it will define where the initial raw data is stored and how the plot will be configured for a particular sensor type. This module is one of the application developer's main responsibility for building out in order to rapidly get data loaded into the SPK framework.

4.3.2 SPKDataSource

The `SPKDataSource` class prepares the data from the `SPKModel` to be rendered on screen by the `SPKPlotView` class. Data will flow in from the model where the application developer can validate, filter, and preprocess the raw data before sending the data through to the `SPKPlotView` class. Plot parameters are validated separately from the sensor data stream as they are separate domains

of information and concerns. The benefit of separating the functionality of the SPKModel and SPKDataSource classes is that it allows the data to not only be validated on the fly during run-time but also allows the developer to create and run unit tests within the SPKDataSource class.

A FIFO display buffer is instantiated in SPKDataSource to act as a temporarily data store for data that has been committed to be viewed on the screen. The display buffer size is defined by the sensor sampling rate and plot duration. The buffer size directly dictates the fixed amount of data points that will be rendered in the view at one instance.

After data has been validated and potentially filtered or preprocessed, it is flushed and committed to the display buffer. SPKDataSource is responsible for reporting when new data is ready to be plotted. Once the sensor data and plot parameters are committed, the data is pulled through the SPKPlotViewDataSource protocol to the SPKPlotView. The SPKDataSource and SPKPlotView are connected using the same data source design pattern used to connect the SPKModel and SPKDataSource except that they adhere to different protocols.

4.3.3 SPKController

The SPKController class holds ownership of all the other classes in the SPK framework. The controller is the first class that is loaded by the application and therefore responsible for instantiating the SPKModel, SPKDataSource, and SPKPlotView classes. As the class name implies, the Controller defines the delegates and datasources for the SPKModelDataSource and SPKPlotViewDataSource protocols. By defining the framework's module relationships solely within the Controller, the application developer has the ability to dynamically change a module's data source or delegate through a single line of code.

Another key aspect of the controller is the plot timer, which sets the rate at which data is refreshed in the SPKPlotView class. This timer is critical for rendering sensor data streams onto the device's

screen. The timer is set to invoke a function call within a specified time interval. The time interval is defined as the View Refresh Rate (v). Within the function call, if the `SPKDataSource` class indicates new data is ready to be plotted, the Controller will flush the sensor data from the `SPKModel` to the `SPKDataSource` and subsequently call a function within `SPKPlotView` to render the new data in the FIFO display buffer.

The last responsibility of the Controller is to handle the logic of the User Interface (UI). For example, if there is a button rendered on the touch screen to pause the sensor data stream. When the `SPKPlotView` class detects the press of the button, the Controller will be notified and handle the event with the proper action and logic to pause the sensor data stream. This is the case for not only buttons but also sliders, zooming, and transitions or segues to other View Controllers.

4.3.4 SPKPlotView

The `SPKPlotView` class contains all the logic necessary to setup and interact directly with the Core Plot framework. The View class utilizes the plot parameters passed in from the `SPKPlotViewDataSource` and applies them within the data source method implementations for Core Plot. The sensor stream data is passed directly to the data source methods from the protocol so the View class does not own or buffer the data.

The decision was made to define `SPKPlotView` as a generic class for plotting generic data over time with no defined axes. This allows other classes to subclass `SPKPlotView` and create sensor specific plots that correlate to their data types. Most sensors stream data using established units, making it unnecessary to modify this class once a subclass has been created to support a specific type of sensor. This is especially beneficial when using medical sensors with this framework as they typically follow an established standard of plotting, which can be complex and very technical. This framework greatly reduces development time for medical applications that utilize standardized medical data types as developers only need to define the `SPKPlotView` once or can use preexisting

implementations.

4.4 Design Space Exploration

In order to understand the design of plotting applications, we analyzed the available source code of the ECGWavesPlayer which was a simple application demonstrating the visualization of three ECG leads on an iOS device [18]. We attempted to reconstruct parts of the source code to plot data from physical sensors rather than a fixed array of values. The effort proved to be futile as the code was too convoluted to be reused for alternative applications. There was a lack of general interface and ability to customize the plot characteristics. Thus the ECGWavesPlayer was not a suitable basis for a framework.

Focus shifted to investigating the design of BabyECG [6], an iOS application developed by a lab mate, which utilized the well-known iOS plotting framework Core Plot. The application utilized Core Plot to plot a simple ECG lead in real time but did not incorporate any axes to give the user reference of the precision or units of the data.

4.5 Development Process

The design and development of Sensor Plot Kit was taken step-by-step, where each class was developed according to priority and necessity.

4.5.1 SPKPlotViewECG Class

We first focused on the development of the `SPKPlotViewECG` class to aid in creating a graph view that adhered to the medical specifications of a standard Electrocardiogram plot. This was a top

priority because we wanted to prove that the Core Plot framework would be viable for plotting ECG graphs. Although it was trivial to setup up the axes to plot according to the standard, there were some nuances of the iOS devices that produce an unintended result. Because the aspect ratio on iPad and iPhone devices differ depending on the generation of the device, the grid would often appear as rectangles rather uniform squares. Typically, Apple promotes developers to use AutoLayout to provide a uniform View experience across devices, but because the grid is setup programmatically through the Core Plot framework, it prevented the direct use of Autolayout. The issue was worked around by adjusting the bounds of which the grids were created, which would force a uniform grid. In addition, there were troubles plotting grid lines with higher precision of units, from say mV to uV. Because of the larger range of computation by a magnitude of 1000, the device struggled to render the grid lines in the uV scale. A workaround was to force a conversion of the input data from the sensor to mV to accommodate for the lack of support plotting in uV units. After overcoming these hurdles, we were able to demonstrate that Core Plot was capable of creating precise and detailed ECG plots.

4.5.2 SPKFilter Class

We focused next on creating the `SPKFilter` class to help filter incoming raw sensor data streams. The class design was inspired by Apple's Image Filter API, which applies image filters on raw images. There are two public methods:

- `init(filterWithName: String, filter: FilterType, filterValue: Any)`
- `applyFilter(inputBuffer : [Double])`.

Using `init`, the developer can create a custom filter based on our provided `FilterType` and specify the parameters of that filter. For example, if the developer wanted to set a Lower Limit filter to remove data lower than a limit of 10, they would instantiate the `SPKFilter` class with the following

parameters `SPKFilter("lowerLimitFilter", SPKFilter.FilterType.LowerLimit, 10)`. Once the filter is created, the developer can apply the filter to an array using the `applyFilter` method. This method utilizes Swift's built-in filter method to filter out items from an array. Once the items are filtered, the method will return the new array.

4.5.3 SPKDataStore Class

Driven by the necessity to replay captured ECG data, we focused on developing the `SPKDataStore` class to load and store captured data from the iOS device's internal flash storage. `SPKDataStore` has three main methods: `update()`, `saveToDisk()`, and `readFileFromDisk(filename)`. The `update` method is an interface to `SPKDataStore`'s internal data buffer to store the Sensor data when it is captured. The developer must add a call to this update function in order for the data to be saved. The `SaveToDisk` function will create a new file in the application's documents folder and save the captured data from the DataBuffer into an array. We have implemented storing the plot parameters into a plist file, while the remaining features are left for future work. This would allow for seamless playback of the data array as it was captured. The `readFileFromDisk` function takes in the filename that the user wants to read and outputs the array within the file. Both `SaveToDisk()` and `ReadFileFromDisk()` methods utilize Apple's `NSFileManager` API to access iOS's file system. In the demo application, we built out a View Controller to act as a file picker from the documents folder and a player to view back the recorded sensor data stream. Overall, this class proved to be not only useful for development of the framework but also a key feature for application developers to allow users to offload data to their iOS devices.

4.5.4 SPKBuffer Class

Next we focused on the `SPKBuffer` class, which is a FIFO buffer that can be modified to be a circular buffer. The motivation behind creating the `SPKBuffer` was to be able to dynamically transfer

data between the SPK modules. The BabyECG application implemented a simple FIFO buffer, which allowed the developer to enqueue or dequeue unit-length data, but with Sensor Plot Kit, we needed the ability to enqueue or dequeue data of varying lengths. The `SPKBuffer` design was inspired by the TPCircularBuffer [16], which allowed multiple methods to produce and consume data into the buffer at varying counts. When producing data, the data is appended to the tail of the array where consuming data is dequeued and removed from the head.

4.5.5 Demo Application Development

We next focused on determining the performance boundaries of our framework through building a simple test bed to allow us to interactively modify the sensor data and view refresh rate within a demo applications UI. We observed that the modifying the view refresh rate would not have much of an affect on the performance, whereas changing the sensor data rate would cause a longer latency in displaying the data as the framework was working harder to keep up with the plotting.

In an attempt to push the performance boundaries further, we decided to scale up the demo application by linking the QT Medical ECG device and plotting the full 12 leads in real time. On the initial run of the full 12-lead ECG view, the application struggled to plot the data as the latency had delays of over 12 seconds.

We then investigated possible explanations and solutions to the sluggish latencies. After further research on the Core Plot forum, we realized that we did not account for the difference in performance that would occur on iPads with and without Retina screens. Retina screens are high-resolution displays with up to 4x the pixel density of a non-Retina screen. So comparing an iPad 3 with a Retina screen to an iPad 2 without, the iPad 3 would have to do up to four times more work. After disabling the number of simultaneously leads, we were able to get 4 leads to plot at comparable speeds to that of a single lead display. Furthermore, we investigated moving the rendering of the plot data onto the main thread of the CPU, but later realized through experiments that

rendering is best kept off the main thread as it may starve other important processes. Lastly, we investigated other options to improve performance and found that avoiding the use of transparent colors and colors with an alpha less than one will increase the amount of work when rendering the plot.

4.5.6 Migration to MVVM Architecture

With the test bed, network logic, and SPK framework integrated into the demo application, it became apparent that the framework was overloaded and cluttered, because the majority of the code existed within the View Controller. After further research, we determined that it was suitable to adapt `SensorPlotKit` to use an MVVM based architecture versus the loosely defined MVC design pattern. We then adapted a delegate and protocol based structure for communication between modules. This allowed for a structured and clear interpretation of the frameworks API as the majority of iOS frameworks utilize the same delegate and datasource protocol.

4.5.7 Performance Improvements

Another minor improvement was changing the method at which plot parameters were set. The BabyECG application created an interface within the Storyboard development environment using `IBDesignable` variables, which are settable within each View. While creating the 12-lead ECG demo application, we found that using the storyboard to set the plot parameters was extremely tedious and often unstable as Xcode would constantly crash while setting the variables. To make the process more efficient, we created a structure to hold on the plot parameters and utilized datasource methods within our protocols to pass the structure from the model to the view. Because the 12-lead ECG views had identical plot parameters for every lead, we were able to set the parameters once and have it applied to all 12 leads. Using the `IBDesignable` method, it would have required setting 8 or more variables on each of the 12 leads.

Lastly, to further improve plotting performance, we realized that it would be more efficient to draw over the prior set of data points rather than pushing and popping points one at a time out of the display buffer. Even to the untrained eye there is noticeably less jitter in plotting over points naturally from left to right on the x axis versus using the original method of pushing points from the furthest right point on the x axis so that they would shift and animate over to the origin of the x axis. This method required less rendering as it only had to redraw $\log(n)$ points as more points were draw on the screen vs. a static n redraw. We reattempted to use this method in the demo application to plot all 12 leads and we were able to achieve our plotting objective with a reasonable latency.

Chapter 5

Experimental Results

After the implementation of the Sensor Plot Framework, we challenged our functional and performance claims by building out several applications that would be practical in the real world. Three applications were built to validate plotting of a single lead sensor, an ECG sensor with twelve [1] output leads and a prerecorded data stream to simulate a high sample rate sensor. These applications were tested, measured and analyzed for both performance and qualitative measures.

5.1 Metrics

Several metrics are used to evaluate the framework's performance. They include latency, sensor data rate, and view refresh rate.

[1] Standard electrocardiograms have a twelve output leads however when plotting their data, there is typically an additional lead present on the plot called Rhythm II which is interpreted from the 12 leads

5.1.1 Latency

The most critical metric is latency, especially when evaluating a real-time application. If the data cannot be plotted shortly after it is received, it shows poorly on its precision and usability. Latency is affected by many different factors, hence demonstrating low latency will in turn improve performance. A major factor that will affect latency is the number of simultaneous streams of sensor data being handled.

5.1.2 Sensor Data Rate

The sensor data rate or sampling rate is the rate at which the sensor data is received by framework from the sensors or read in through a file. If there are multiple sensor data streams the framework must work harder to maintain the highest data rate possible. If the sensor data rate is higher than the rate at which the framework can process and plot data, the framework will have buffer the data, putting additional load onto the framework.

5.1.3 View Refresh Rate

View refresh rate is another metric that measures how fast the device can render images on the host device's screen relative to time. It can be controlled within the framework to help tweak performance of the plotting.

Several trade-offs can be inferred from the metrics mentioned above such as latency vs. sampling rate vs. view refresh rate when evaluating a single stream of sensor data and latency vs. number of sensor streams.

5.2 Experiment Setup

5.2.1 Case Study 1: Single Sensor Data Stream

In this application, our purpose was to observe the effects of varying the sensor's sampling rate and the framework's view refresh rate. We chose the iPad and iPhone's onboard accelerometer as our data source to the framework, because Apple's Core Motion framework allows developers to easily capture real-time data at a specified sample rate in just a few lines of code. The accelerometer's data was also very stable and predictable, making it an obvious choice since it did not need any filtering, and the range of the data was easily controlled.

5.2.2 Case Study 2: Wireless Multi-lead Sensor Data Stream

To further scale and stress our framework, we created an application to plot a 12-lead ECG sensor in real time as shown in Figure 5.1. In contrast to the first case study, the ECG sensor has a fixed sampling rate of 50 Hz and data is transmitted wirelessly over Bluetooth Low Energy (BLE) Technology. The purpose of this case study was to demonstrate the framework's ability to manage buffering and rendering of a multi-lead sensor data stream in real time. In our tests, we modified both the view refresh rate and number of streams (leads) rendered to understand its implications on overall latency.

5.2.3 Case Study 3: High Sample Rate Sensor Data Stream

In this high performance driven application, we utilized pre-recorded ECG data stored in non-volatile memory (flash) to act as the data source for a single-lead view output. Because we were no longer limited by the on-device sensor or BLE ECG sensor to stimulate the framework, we

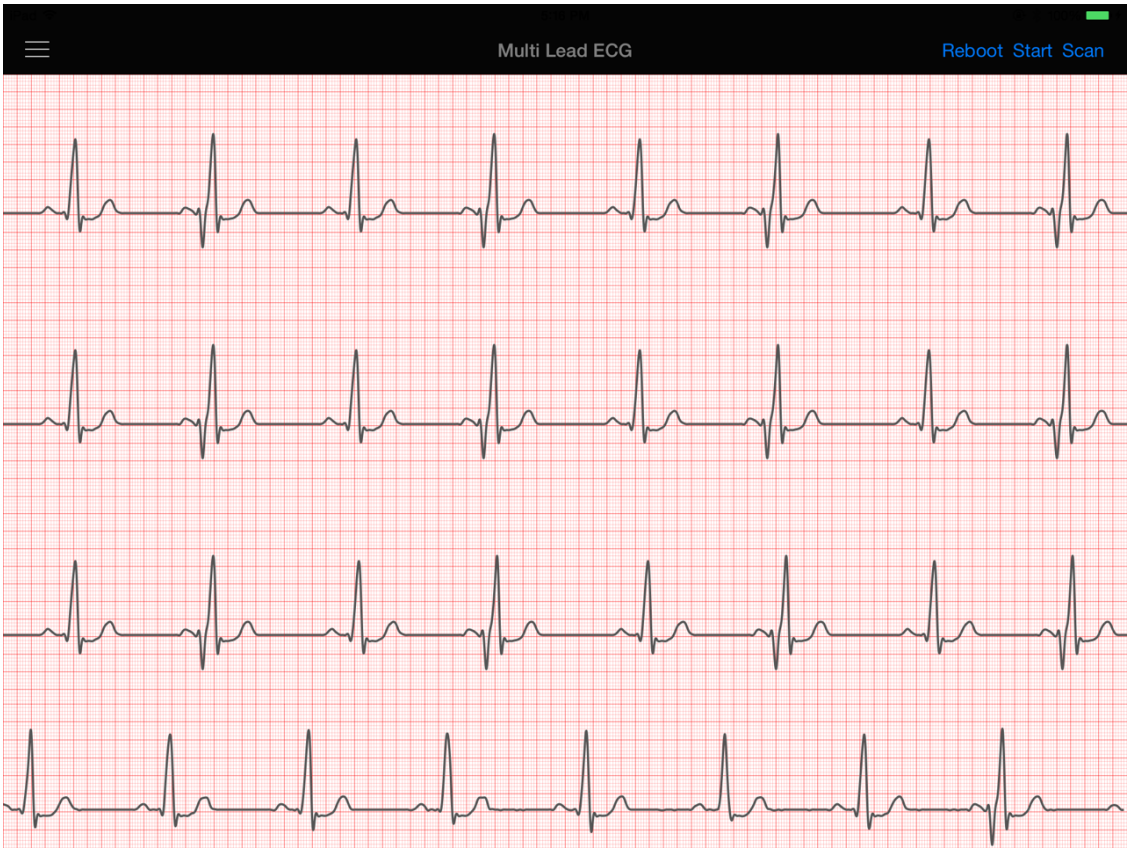


Figure 5.1: Screenshot of 12-lead ECG sensor streaming on the iPad 3 demo application

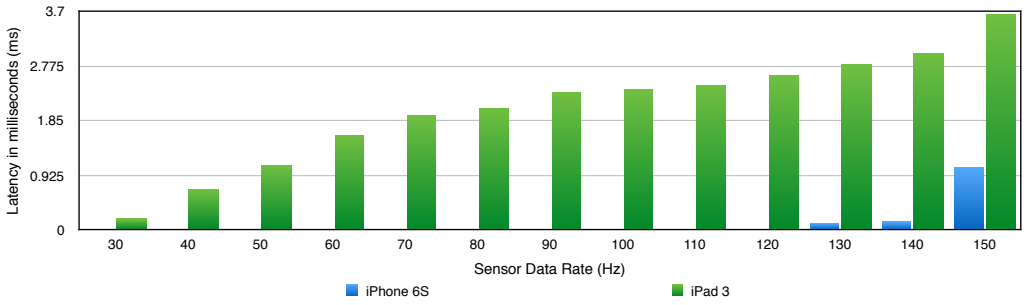
were able to utilize the iOS Simulator to simulate the application on a wider array of hardware. We tested our application on the iPhone 6S, iPad 2, and iPad 3 simulators along with the physical iPhone 6S and iPad 3 devices to cross-reference the latencies observed. The data is read from the flash into DRAM, which allows the data to be available instantly so the sensor data sampling rate was dictated by the view refresh rate. We dramatically increased the view refresh rate to push the rendering boundaries in an attempt to find the highest rate that would incur no latency.

5.3 Performance Results

For all the applications, we calculated latency through the same method. We first measured the duration of time from when a single point of data was received from the sensor in the SPK framework to when it was submitted to be rendered by Core Plot. Since we explicitly set a timer to dictate how often the View should render new points from the sensor, we took the delta between the set view refresh interval to the observed duration to determine the latency of the framework plotting. It was our initial intention for the latency calculation to include the duration of time it took the internal Core Graphics framework to plot the physical pixels on the screen after receiving the request from Core Plot, but we were unable to find a callback API within Core Graphics to notify the completion of rendering.

5.3.1 Case Study 1: Single Sensor Data Stream

In our experimental run when plotting the accelerometer data, we noticed minimal latency when the view refresh rate and data-sampling rate were set to 30 Hz. We attempted to adjust the sampling rate to be faster than the view refresh rate but observed that data will be bottlenecked by the view refresh rate. In addition, setting the view refresh rate greater than the data-sampling rate would be inefficient and impractical as the process would attempt to refresh the view data despite no new



	Sensor Data Rate (Hz)												
Device	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**	**110**	**120**	**130**	**140**	**150**
iPhone 6S	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	4ms	13ms	17ms	114ms	152ms	1058ms
iPad 3	182ms	687ms	1090ms	1598ms	1948ms	2055ms	2315ms	2370ms	2449ms	2611ms	2794ms	3000ms	3650ms

Figure 5.2: Performance Results: Single Sensor Data Stream

available sensor data.

We experimented further by increasing the refresh and data rates to determine the threshold at which the devices could maintain minimal latency. The results are seen in Figure 5.2.

The iPhone 6S had a dramatically better performance over the iPad 3 as it had a smaller LCD screen and more powerful processor. There was a negligible latency at refresh and data rates up to 100 Hz. Once we pushed the rate to 150 Hz did we start noticing latency of around 1 sec. In addition we observed significant signs of interpolation of the accelerometer data when the sample rate was at 150 Hz.[2]

[2]The accelerometer within the iPhone has an observed minimum and maximum sampling rate of 10 Hz and 100 Hz to conserve battery power. [12]

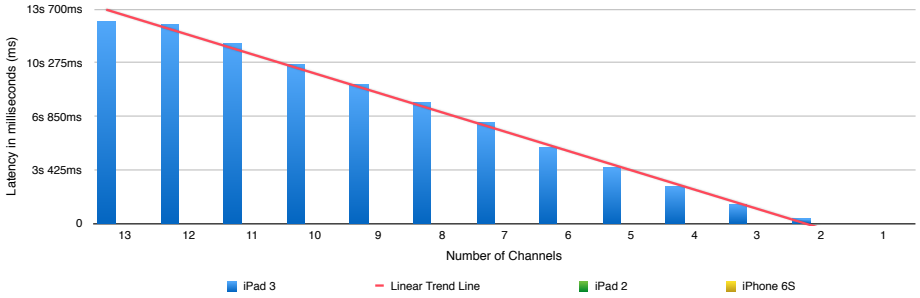
The iPad 3 was slower overall, with mediocre latencies when plotting under 50 Hz. Above 50 Hz, the latency was not feasible for use.

5.3.2 Case Study 2: Wireless Multi-lead Sensor Data Stream

In this case study, we experimented with plotting multiple leads simultaneously on a single screen. We kept the view refresh rate fixed at 30 Hz and the sensor data rate at 50 Hz. The purpose of this experiment was to determine the performance characteristics when plotting multiple leads by adjusting the number of lead stream. We first tested running the full 13 streams at 30 Hz connected to the QT Medical ECG device and observed negligible latency on the iPhone 6S at around 2 ms. In stark contract, the iPad 3 had a latency of over 12 seconds, which rendered the application useless for real-time use as shown in Figure 5.5

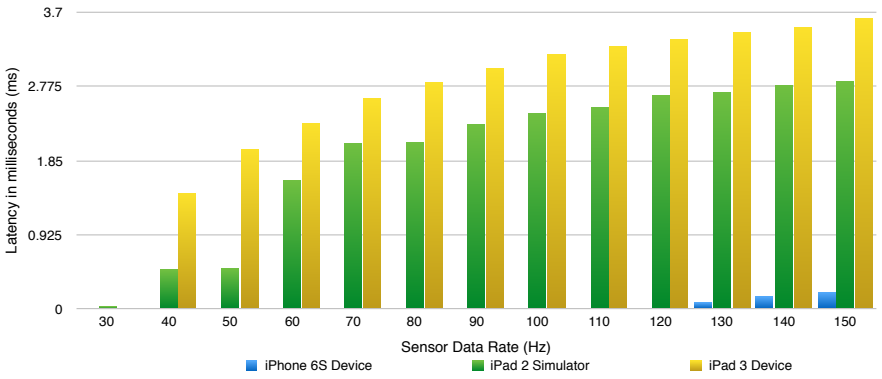
5.3.3 Case Study 3: High Sample Rate Sensor Data Stream

Here we wanted to expand our testing out to a wider range of iOS hardware. We wanted to get more data specifically on the iPad 2 as it has the same size screen as the iPad 3 but with a lower pixel density. As the results show in Figure 5.4 the iPad 2 held a low latency below 1 sec at 50 samples per sec, whereas the iPad 3's latency was nearly quadrupled. This aligns with our expectations as forums on Core Plot have suggested that retina displays like that on the iPad 3 require four times more work than a non-retina display.



	Number of sensor streams												
Device	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**
iPad 3	12s 888ms	12s 739ms	11s 484ms	10s 141ms	8s 857ms	7s 710ms	6s 401ms	4s 788ms	3s 559ms	2s 343ms	1s 139ms	256ms	< 1 ms
iPad 2	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms
iPhone 6S	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms

Figure 5.3: Performance Results: Wireless Multi-lead Sensor Data



	Sensor Data Rate (Hz)												
Device	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**	**110**	**120**	**130**	**140**	**150**
iPhone 6S Device	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	9ms	9ms	10ms	83ms	153ms	211ms
iPad 2 Simulator	29ms	497ms	514ms	1600ms	2066ms	2079ms	2289ms	2431ms	2498ms	2669ms	2698ms	2779ms	2845ms
iPad 3 Device	6ms	1436ms	2000ms	2306ms	2630ms	2822ms	3003ms	3174ms	3265ms	3348ms	3438ms	3504ms	3620ms

Figure 5.4: Performance Results: High Sample Rate Sensor Data Stream

5.3.4 Evaluation

When considering the fluid rendering performance of high definition video streams on the iPad 3, the SPK framework lacks the efficiency and throughput performance relative to video rendering on iOS. Apple has specified that the iPad 3 can support the rendering and decoding of 1080p H.264 encoded video streams at a rate of least 30 frames per second [3]. The SPK framework experiences additional latency primarily due to the software stack. Core Plot is built on top of Core Graphics or Quartz 2D framework which has been known to not be as efficient as interfacing directly with the OpenGL (ES) library. Apple has specifically noted that Core Graphics (also known as Quartz) is the native drawing engine for iOS apps and provides support for custom 2D vector- and image-based rendering. Although not as fast as OpenGL ES rendering, this framework is well suited for situations where you want to render custom 2D shapes and images dynamically" [2]. The diagram below shows an overview of the graphics software stack [7]. Furthermore, Core Plot does not efficiently utilize the Graphics Processing Unit (GPU) in comparison to the Core Video framework. Core Video efficiently pipelines and caches video frames, allowing the CPU to continuously push new frames to the GPU as well as reduce the load on the GPU through caching when presenting frames with similar textures. Core Plot has some caching features but we have not fully evaluated its effectiveness. Further, we believe there could be some performance improvements to SPK to reduce the CPU load in order to increase throughput to the GPU.

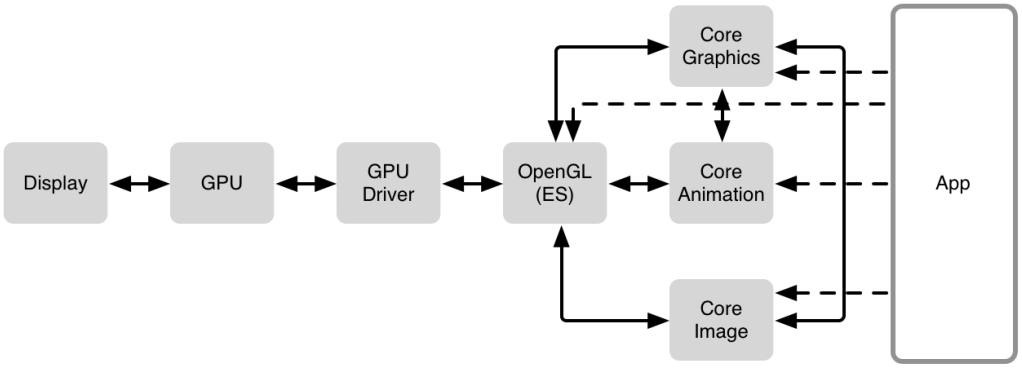


Figure 5.5: Graphics Software Stack on iOS [7]

5.4 Qualitative Results

This section discusses evaluation results based on qualitative metrics.

5.4.1 Reusability

We have achieved reusability of the SPK framework through our MVVM based design, a structured interface through explicit protocols and the creation of demo applications with minimal modification. The MVVM design allowed our demo applications to reuse the `SPKDataSource` and `SPKPlotView` modules as they were turnkey ready and did not need any further modification to be incorporated into the applications from case study 1 and 3. If our framework did not properly modularize or separate out the concerns, the applications would have required modifications in all areas of the framework. The explicit protocol definitions, `SPKModelDataSource` and `SPKPlotViewDataSource`, made it a frictionless process to modify the source of the sensor data in both case studies. We were able to modify the sensor data source by changing only 6 lines of code with the `SPKController`. Through the clarity of design and enforcement of protocols, SPK has been able to demonstrate reusability.

5.4.2 Scalability

Scalability of the SPK framework has been exhibited through the extensibility of increasing the number of sensor streams. In case study 2, we implemented an application that managed multiple sensor streams through the simplicity of instantiating additional `SPKPlotViewECG` modules for each sensor stream. Further, we established that there existed a linear increase in latency in relation to the number of sensor streams outputted. We determined on the iPad 3 that each additional sensor stream would inflict at least an additional 1000 ms of latency. The low latency results of iPhone 6S and iPad 2 clearly demonstrates SPK's ability to scale up the number of sensor streams.

5.4.3 Minimal Development Time

When SPK was originally developed using a MVC design, it took a few days to complete the implementation. Subsequently, when we attempted to modify the sensor data source on top of the cluttered architecture, it took upwards of two hours to update and understand the implications of changing the data source. After shifting the design to MVVM, we were able to drastically reduce our development time of the demo applications. Once the initial demo application was built, we were able to bring up the application in case study 3, through modification of the sensor data source, in under ten minutes. The multi sensor stream application in case study 2 took around two hours to implement. SPK's strengths in structured MVVM design and reusability has allowed the framework to significantly reduce development time of sensor plotting applications.

Chapter 6

Conclusion

6.1 Summary

This thesis presents the Sensor Plot Kit (SPK) for helping the development of iOS apps that must perform viewing of real-time streaming sensor data or archived data. Our use of the Model View View-Model (MVVM) design pattern addresses a pitfall with the standard Model View Controller (MVC) pattern in iOS, where the view controller can quickly become too complex. MVVM reduces complexity and maximizes code reusability while remaining compatible with MVC. With short latencies and high data rates, our proposed SPK is suitable for advanced medical applications such as electrocardiograms (ECG). Our results on real-world applications show that developers find it much quicker to build low-latency, high-throughput, real-time plotting apps that are robust and maintainable. The fact that SPK is open-source makes it easy to customize and embed into a wide range of apps in many application domains in science, engineering, medicine, the financial market, and many other fields.

6.2 Future Work

The SPK described in this thesis represents the first step in this direction. In terms of performance, there is room for improvement in the reduction of software latencies especially for iPad devices that plot simultaneous streams of sensor data. Characterization of the latencies would help identify areas of optimization. I believe the manner at which multiple views are refreshed could be optimized to work in parallel rather than in a sequential manner.

Many new features remain to be implemented for future work, especially for medical applications. The medical industry would highly benefit from the support of additional medical sensor types as well as the ability to load and store data in medical standard formats such as DICOM. Furthermore, SPK has the potential to expand its interface by supporting the loading and storing of sensor data from cloud based servers or nearby mobile devices. This would broaden the framework's impact and increase the accessibility of its data. To take advantage of the touch screen and robust User Interface, SPK could provide the ability for users to annotate and take in notes of the sensor data as it streams in real-time.

Bibliography

[1] Apple. Delegates and Data Sources. https://developer.apple.com/library/ios/documentation/General/Conceptual/CocoaEncyclopedia/DelegatesandDataSources/DelegatesandDataSources.html.

[2] Apple. iOS Technology Overview. https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/MediaLayer/MediaLayer.html.

[3] Apple. iPad (3rd generation) - Technical Specifications. https://support.apple.com/kb/SP647?locale=en_US.

[4] Apple. Model-View-Controller. https://developer.apple.com/library/ios/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html.

[5] A. Chernish. Lightweight iOS view controllers. https://yalantis.com/blog/lightweight-ios-view-controllers-separate-data-sources-guided-mvc/.

[6] T.-C. B. Chien. BabyECG. http://git.epl.tw/qtmedical/ecg_iosapp/tree/master/BabyECG.

[7] D. Eggert. Getting Pixels onto the Screen. https://www.objc.io/issues/3-views/moving-pixels-onto-the-screen/.

[8] A. Furrow. Introduction to MVVM. https://www.objc.io/issues/13-architecture/mvvm/.

[9] E. Goloboyar. Graceful approach to working with Core Plot. https://yalantis.com/blog/work-core-plot-library-alternative-approach/.

[10] INNOVENTIONS, Inc. Sensor Kinetics. https://itunes.apple.com/us/app/sensor-kinetics/id579040333?mt=8.

[11] K. Katevas, H. Haddadi, and L. Tokarchuk. Poster: Sensingkit: A multi-platform mobile sensing framework for large-scale experiments. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, 2014.

[12] C. Labs. setAccelerometerInterval. https://docs.coronalabs.com/api/library/system/setAccelerometerInterval.html.

[13] N. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. Campbell. A survey of mobile phone sensing. *Communications Magazine, IEEE*, 48(9):140–150, Sept 2010.

[14] D. Lo. SensorPlotKit. https://github.com/derrickallenlo/SensorPlotKit.

[15] E. Skroch. Core Plot. https://github.com/core-plot/core-plot.

[16] M. Tyson. A simple, fast circular buffer implementation for audio processing. http://atastypixel.com/blog/a-simple-fast-circular-buffer-implementation-for-audio-processing/.

[17] Vernier Software & Technology. Vernier Graphical Analysis. https://itunes.apple.com/us/app/vernier-graphical-analysis/id522996341?mt=8.

[18] W. Yang. ECG Waves Player. https://github.com/hezone/ECGWavesPlayer.