The Report Committee for John Jason O'Donnell
Certifies that this is the approved version of the following report:

StarMapper: An Android-based Application to Map Celestial Objects

APPROVED BY

SUPERVISING COMMITTEE:

Adnan Aziz, Supervisor

Nur Touba

StarMapper: An Android-based Application to Map Celestial Objects

by

John Jason O'Donnell, B.S.E.E.

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2013

Acknowledgments

I would like to thank my supervisor, Dr. Adnan Aziz, for his support and guidance on this project. I would also like to thank Dr. Nur Touba as the reader of this report. Finally, I would like to thank my parents and sister for their patience and support as I worked toward completion of my Master's degree, and a special thanks to my brother for helping me to regain my motivation when it was lost.

StarMapper: An Android-based Application to Map Celestial Objects

John Jason O'Donnell, M.S.E.
The University of Texas at Austin, 2013

Supervisor: Adnan Aziz

This report describes StarMapper, a mobile application designed for the Android platform that interactively maps the celestial sky and can provide Wikipedia information about celestial objects to the user. The stars, constellations, planets, sun, and moon are all rendered in real-time and the user can navigate the celestial map simply by pointing the device around the sky to find and identify the different celestial objects. However, if the user prefers, a manual touch—based map navigation feature is also available in StarMapper. While other Android applications currently exist for mapping the sky, such as Google's Sky Map, StarMapper aims to enhance the experience by also providing additional information about celestial objects to the user by means of a simple click on the screen. For obtaining more information about a particular constellation or other celestial object, the user only needs to click on the object's name in the map, and the device's web browser opens to the Wikipedia page of the clicked object. Through this simple mechanism, the user can learn much more about astronomy than just locations of celestial objects.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

StarMapper is an interactive, real-time, OpenGL-based application that maps celestial objects to the device screen, allowing the user to find and see constellations and stars by simply pointing the device in the sky. The user can optionally navigate around the map through a manual touch feature, and also retrieve information from Wikipedia by simply clicking on the names of the constellations and other celestial objects. The targeted users for StarMapper are those who enjoy astronomy and stargazing, such as in the user stories below.

1.1 User Story 1

Tim is a boy scout on a camping trip for the weekend. After his campfire dinner, his troop retreats to their respective tents for the night. However, Tim enjoys looking at the stars at night, and, with the night being clear, decides to try finding new constellations. Tim opens up StarMapper on his smartphone and pans it across the sky, easily finding new constellations because of the intuitive nature of the application. In addition, he is able to locate the planet Venus, and the star Vega, which he finds is part of the constellation Lyra.

Wanting more information about Vega, he simply clicks on Vega's name on the map and his phone's web browser opens up to the Wikipedia page for Vega, where he finds that Vega is the fifth-brightest star in the night sky.

1.2 User Story 2

Bob is an amateur astronomer who enjoys bringing his telescope outside the city at night to view different objects in the sky. One night, he is unsure of which object he would like to view, so he opens StarMapper on his phone and points it across the sky, discovering that Venus, Jupiter, and Mercury are currently in their 'Dance of the Planets' formation [22], a planetary conjunction that only occurs about once a decade. He quickly points his telescope in their direction to enjoy the spectacle.

1.3 Report Outline

The report begins with a general description of the Android platform, followed by a discussion of the Android application architecture before delving into the design and implementation of the StarMapper application itself. The report is concluded with a development overview, project challenges, key learnings, and some final thoughts.

Chapter 2

The Android Platform

Android is an open-source software platform developed by Google, along with the support of the Open Handset Alliance [13]. It is intended as a complete software stack that includes everything from the operating system, through middleware, and up to the applications level. This unified approach to mobile software development allows the application developers to run their applications on different mobile devices powered by Android [20].

This chapter presents an overview of the Android platform architecture, and the key principles underlying its design. The Android software stack consists of a Linux kernel and collection of C/C++ libraries exposed through an application framework that provides services and management of runtime and applications. A diagram of the architecture is shown in Figure 2.1. The five main components are: the Linux kernel, the core libraries, Android Runtime, the application framework, and the applications themselves [9]. The following sections will describe in more detail the purpose, principles, and interactions of each platform component.

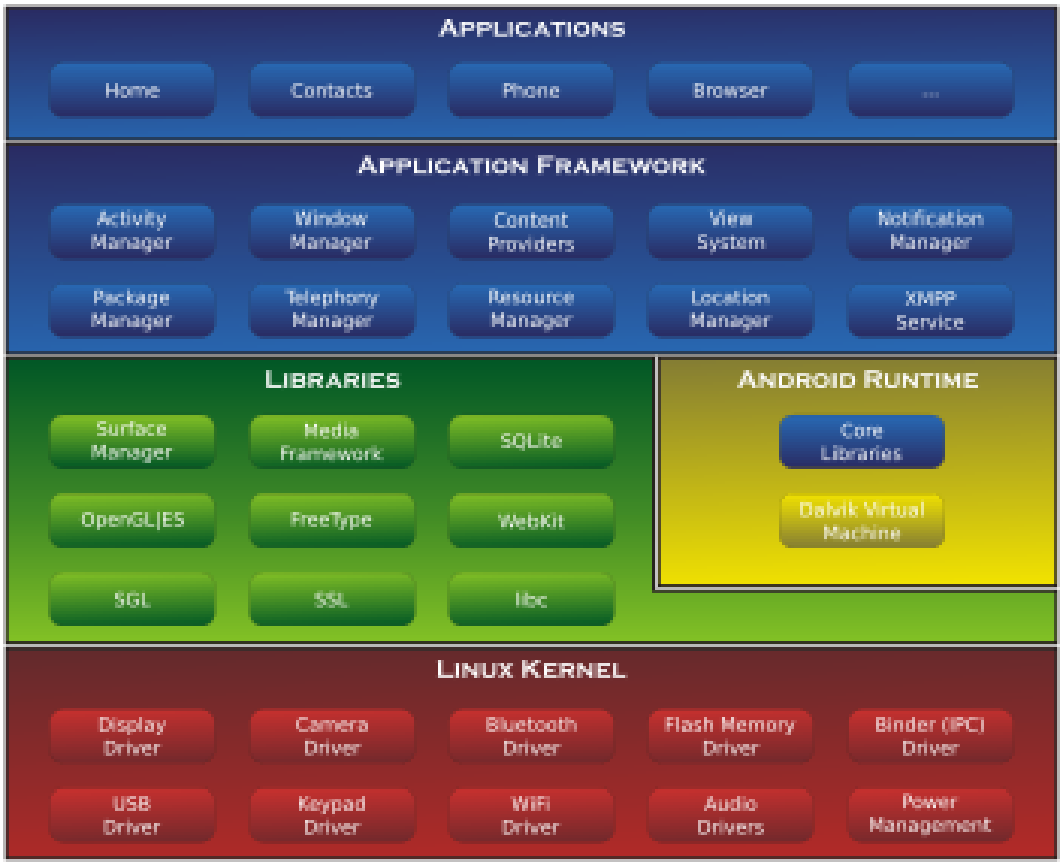

Figure 2.1: The Android Platform Architecture

2.1 The Linux Kernel

The latest version of Android, Android 4.4 KitKat, is based on the Linux kernel version 3.10, while all versions of Android older than 4.0 Ice Cream Sandwich, were based on Linux kernel version 2.6.x. Android's Linux kernel has further architecture changes by Google outside the typical Linux kernel development cycle, such as a power management feature called 'wake-locks' [24]. The Linux kernel acts as an abstraction layer between the device hardware and the upper layers of the Android software stack. It was chosen as the foundation of Android for several reasons, including its proven driver model and the ability to reuse some existing drivers from Linux in Android. It also contains memory and process management, a security model, networking, and other core operating system infrastructure [16].

2.2 Native Libraries

The layer directly above the Linux kernel consists of the native libraries used by the Android architecture. These libraries are written in C/C++, and consist of components such as the surface manager, which is responsible for displaying the correct draws from the appropriate applications at the correct time, and to the appropriate pixels on the screen. The libraries also contain two sets of graphics libraries: the OpenGL ES library and the SGL graphics library. The OpenGL ES library is capable of rendering in 3D, while the SGL library is the main 2D graphics library used by Android. The implementation of the OpenGL ES library can be accelerated through hardware if the device

running the library is capable of it. Android is capable of utilizing both libraries within the same application, so both 2D and 3D renderings are possible simultaneously. The StarMapper application described in this report utilizes the OpenGL ES library for its 3D perspective capabilities.

The media framework is supplied by the Open Handset Alliance, and includes all the codecs that make up the media experience, including mpeg4, h.264, mp3, aac, and other audio and video codecs. The FreeType library contains the functions necessary for bitmap and vector font rendering done by Android. An implementation of SQLite is used for creating databases in the system's memory. These databases are useful for information storage and also sharing of data between applications. An open-source browser engine called WebKit is also included in the C/C++ libraries. It powers the web browser included in Android, and is also the same engine that powers Apple's Safari browser. Google has enhanced the engine to render on small screens, such as for mobile phones. Also included is the SSL (Secure Socket Layer) library, which is responsible for network security.

Typically, the native C/C++ libraries are not directly accessed by the Android application developer; they are accessed solely through the Java-based Android core library APIs which will be described in the next section. In the event that direct access to these libraries is needed, it can be achieved by using the Android Native Development Kit (NDK), the purpose of which is to call native methods of non-Java programming languages like C/C++ from within Java code using the Java Native Interface (JNI) [19].

2.3 Android Runtime

The Android Runtime was designed by Google specifically to meet the needs of running Android in an embedded environment [16]. An embedded environment typically contains limited resources in regards to memory, cpu power, and battery life. In order to run Android in this type of environment, Google created the Dalvik Virtual Machine, a special type of a Java Virtual Machine specifically designed and optimized for Android. The Dalvik Virtual Machine, along with the Android core libraries, make up the Android Runtime.

2.3.1 The Dalvik Virtual Machine

The Linux kernel provides a multitasking execution environment that allows multiple processes to execute concurrently. Thus, it is possible that every active Android application could simply run as a process directly on the Linux kernel. However, in reality, each Android application runs on its own instance of the Dalvik Virtual Machine. Running applications in virtual machines provides a number of advantages. Firstly, applications are essentially sandboxed, in that they cannot detrimentally interfere (either intentionally or accidently) with the operating system, other applications or directly access the device hardware. Secondly, this enforced level of abstraction makes applications platform neutral in that they are not tied to any specific hardware [19].

The Dalvik Virtual Machine was developed by Google and relies on the underlying Linux kernel for low-level functionality. It is more efficient than the standard Java Virtual Machine in terms of memory usage, and is

specifically designed to allow multiple instances to run efficiently within the resource constraints of a mobile device [11]. In order to execute within a Dalvik Virtual Machine, application code must be transformed from standard Java class and JAR files at build time to the Dalvik executable format (.dex), which on average has a 50% smaller footprint than standard Java bytecode [19]. Dex formatted bytecode data structures are also designed to be shared across processes. This helps to allow multiple Dalvik Virtual Machine instances to run simultaneously, one for each application that is running. Standard Java files can usually be converted into Dex format using the dx tool included with the Android SDK.

2.3.2 Android Core Libraries

The Android core libraries, also known as the Dalvik libraries, contain the collection classes, I/O, and other tools and utilities that are typically used both directly and indirectly by Android developers. These libraries form the foundation of the Android framework. The core libraries fall into three main categories: Dalvik Virtual Machine Specific Libraries, Java Interoperability Libraries, and Android Libraries.

2.3.2.1 Dalvik Virtual Machine Specific Libraries

The Dalvik Virtual Machine libraries enable requesting or modifying Virtual Machine specific information. Code that uses these classes is only portable across Dalvik-based systems, but are generally not directly used by

most Android application developers [14]. Table 2.1 shows the different Dalvik specific libraries, along with a description of each library [8].

Package	**Description**
dalvik.annotation	Defines annotations used within the Android system
dalvik.bytecode	Provides classes surrounding the Dalvik bytecode
dalvik.system	Provides utility and system information classes

Table 2.1: Dalvik VM Specific Libraries

2.3.2.2 Java Interoperability Libraries

Android applications are developed primarily with the Java programming language. The standard Java development environment includes a vast array of classes that are contained in the core Java runtime libraries. These libraries provide support for tasks such as string handling, networking, file manipulation, and many others that are both familiar and widely used by Java developers regardless of platform. The Java interoperability libraries are an open source implementation, based on the Apache Harmony project [6], of a subset of the standard Java core libraries that have been adapted and transformed for use by applications running with a Dalvik Virtual Machine [19]. Table 2.2 lists some of the most commonly used Java interoperability libraries.

Library	Description
java.io	Provides I/O facilities
java.lang	Provides core classes of Android, including Object
java.nio	Provides buffer classes to handle data
java.security	Provides classes of Java security framework
java.sql	Provides interface for accessing SQL databases
java.util	Provides an extensive set of utility classes

Table 2.2: Java Interoperability Libraries

2.3.2.3 Android Libraries

The Android libraries encompass the Java-based libraries that are specific to Android development. The libraries composing the Application Framework fall into this category, in addition to those libraries that facilitate user interface building, graphics drawing, and database access. A summary of some key core Android libraries available to developers is given in Table 2.3.

Library	Description
android.app	Provides access to application model
android.content	Facilitates content access between apps
android.hardware	Provides API to access device hardware
android.opengl	Java interface to Open GL ES 3D rendering API
android.media	Provides classes to enable playback of audio/video
android.util	Utility classes such as XML handling
android.view	Building blocks of application UI
android.widget	Collection of prebuilt UI components

Table 2.3: Android Libraries

2.4 The Application Framework

The Application Framework is a set of services that collectively form the environment in which Android applications run and are managed. All Android applications use the Application Framework as their base toolkit. This framework implements the concept that Android applications are constructed from reusable, interchangeable, and replaceable components. Because of this framework, applications also have the ability to publish their capabilities along with any corresponding data that can be found and reused by other applications [19]. Table 2.4 shows the managers, services, and providers that make up the Android Application Framework.

Component	**Description**
Activity Manager	Controls application lifecycle
Window Manager	Java abstraction layer for SurfaceManager
Content Providers	Allows apps to shared data with other apps
Resource Manager	Provides access to resources such as UI layout
Notifications Manager	Allows apps to display notifications to user
View System	Extensible set of views to create application UI
Package Manager	Allows apps to find other apps installed on device
Telephony Manager	Provides info about telephony services on device
Location Manager	Provides access to location services
XMPP Services	Provides API for XMPP comms protocol

Table 2.4: The Application Framework

Many of the components in the Application Framework are obvious in their purpose, but the Content Provider component is a piece of the Application Framework that deserves special consideration. It is the component

that allows applications to share their data with other applications, and is the major driver behind reusability and interchangeability in the Android system.

2.5 Applications

The topmost layer in the Android software stack consists of applications. These comprise both the native applications provided with a particular Android implementation (such as the web browser, phone, contacts, email, etc.), and third-party applications downloaded, installed, or developed by the user of the Android device, such as StarMapper. All of the applications on this layer are built and use the same framework provided by the layers below it. The next chapter will discuss in more detail the main building blocks that can comprise a typical Android application design.

2.6 Summary

Understanding the overall architecture of the Android software stack helps to create a strong foundation for Android application development. The key goals of the Android architecture are performance and efficiency in both application execution and the implementation of reuse in application design.

Chapter 3

StarMapper — An Android Application

This chapter will describe the design of StarMapper, an Android application that interactively maps celestial objects and provides users with Wikipedia-based information on those objects. Before reviewing the StarMapper application itself, however, we will first go over the general architecture of an Android application.

3.1 The Android Application Architecture

Android applications are written in the Java programming language. The Android SDK compiles the application code, along with any application data or resource files, into an Android *package file*, with the suffix *.apk*. An individual *.apk* file is considered one application. This is the file used by Android devices to install the application [17].

Each Android application is comprised of one to four main component types: Activities, Services, Content Providers, and Broadcast Receivers. An application can have multiple instances of each of these components. Activities, Services, and Broadcast Receivers can be activated between applications through an asynchronous message system called an *Intent*. The Content

Provider is activated through a *ContentResolver*. In addition to these components, each application must also have a special file called a *manifest file*, which declares the components used in the application, along with application requirements. Application-specific resources such as image and audio files make up the final piece of an Android application.

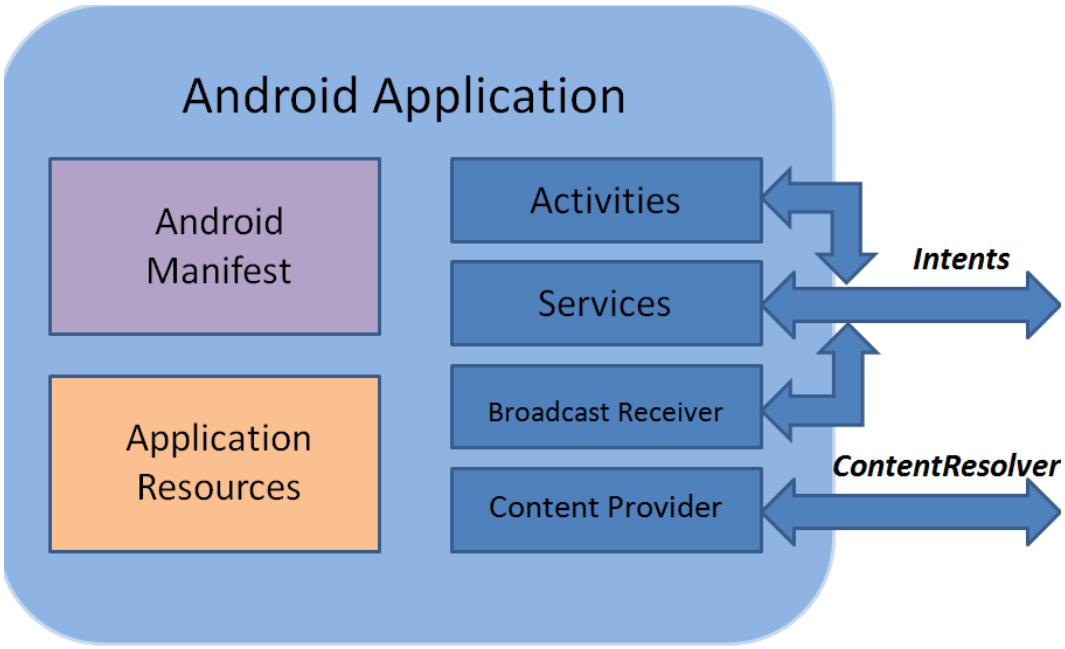

Figure 3.1: The Android Application Architecture

3.1.1 Android Application Components

3.1.1.1 Activities

An Activity is a single, standalone module of application functionality that usually correlates directly to a single user interface screen and its corresponding functionality [18]. An email application, for example, might have

three activities that comprise it: one for listing all emails, one for opening and reading a particular email, and one for composing/replying to an email.

Activities are intended to be reusable and interchangeable components that can be shared between applications. If another application being developed has a need to send out email messages, it can simply reuse the activity that composes emails from the example previously mentioned, rather than develop the same activity specifically for the new application.

Activities must also be implemented completely independent of other activities in the application. In other words, an activity cannot rely on being called at a known point in a program flow, since other applications may call the activity outside of the current application.

3.1.1.2 Services

Services are tasks that do not have a user interface, have a long lifecycle, and run in the background. They can be both started and managed from Activities, Broadcast Receivers, or other Services. A Service is intended for an application that needs to continue to perform a task, but does not need a user interface visible to the user. Even without a user interface, they can still notify the user of events through the use of notifications and toasts, which are small messages that appear on the screen without interrupting the current Activity visible to the user.

Services are given a higher priority than other processes by the Android runtime and are only terminated as a last resort by the system to free up

resources [18]. If the runtime does need to kill a Service for its resources, that Service will be automatically restarted once adequate resources are available.

One example of a Service is an audio streaming application. A user may start a song from an Activity (UI screen) within the audio streaming application, but once the music begins, the Activity can be closed, and the user can use other Activities or applications while the music continues to play.

3.1.1.3 Broadcast Receivers

A Broadcast Receiver is a component that does not run until it is triggered by an external event, such as the phone ringing, or the battery becoming low. This external event is sent to the Broadcast Receiver through a mechanism called a Broadcast Intent, which is described in more detail in the next section. The Broadcast Receiver must be further configured with an Intent Filter to indicate the types of Broadcast Intents in which it is interested. When a matching Broadcast Intent is received, the Broadcast Receiver is invoked by the Android runtime regardless of whether the application that registered the Receiver is currently running in the foreground. Broadcast Receivers then have a limit of five seconds to complete any tasks that are required of it [18]. This is to ensure major processing work is not done inside the Receiver.

Broadcast Receivers operate in the background and, like Services, do not have a user interface.

3.1.1.4 Content Providers

A Content Provider is the mechanism that allows data to be shared between applications within Android. The Contacts application built into most Android implementations is a good example. It contains a Content Provider that other applications can use to access the user's stored database of contacts on the Android device.

Content Providers can also be useful for reading and writing data that is private to the application, and not shared with other applications. For example, a NotePad application can use a content provider to save notes input by the user.

3.1.2 Intents and Content Resolver

3.1.2.1 Intents

Activities, Services, and Broadcast Receivers are activated by an asynchronous message called an Intent [17]. Intents bind components to each other at runtime. They act as messengers for a component that requests an action from other components, whether the other components belong to the current application or another application. An Intent can specify whether to activate a specific component, or just a specific *type* of component. In other words, the Intent can be either explicit or implicit.

Activities and Services use Intents to determine what action to perform, and the Intent may also specify the URI (Uniform Resource Identifier) of the data to act upon. In cases where an Intent is used to request an Activity to

start, an Intent can also be used for the requested Activity to return a result. One example is allowing a user to select a contact from the user's contact database, and returning the contact information in a URI contained within an Intent.

Broadcast Intents are system-wide Intents sent out to all applications with a registered Broadcast Receiver. Broadcast Receivers whose registration matches the Broadcast Intent are then invoked by the Android system. Unlike Intents for Activities and Services, which define the action to perform, Broadcast Intents simply define the announcement being broadcast. A Broadcast Intent can be an asynchronous Intent that is broadcast to all Broadcast Receivers at the same time, or it can ordered in such a way that it is sent to one Receiver at a time, processed, then aborted or allowed to proceed on to the next Broadcast Receiver.

3.1.2.2 Content Resolver

Content Providers are activated when targeted by a request from an object called a Content Resolver. Content Resolvers handle all direct transactions with Content Providers so that the component requesting the transactions with the Provider only needs to call methods on the Content Resolver object. This leaves a layer of abstraction and security between the requesting component and the Content Provider.

3.1.3 The Android Manifest

The Android Manifest file pulls together the various elements that comprise an Android application. It is an XML file that declares all the Activities, Services, Broadcast Receivers, and Content Providers that make up the application. Figure 3.2 shows the declaration of the main Activity component in the StarMapper application. In addition to component declarations, the Manifest file also declares required user permissions needed by the application, the minimum SDK version needed, required hardware and software features, and and API libraries the application needs (other than the Android framework APIs) [17].

```
<activity
    android:name="com.starmapper.android.program.StarMapper"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.NoTitleBar.Fullscreen">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Figure 3.2: Activity Component Declaration in StarMapper Manifest File

3.1.4 Application Resources

A typical Android application will a contain a collection of resource files that hold strings, images, fonts, and other types of data that appear in the user interface in conjunction with the XML representation of the UI layout [17]. The resource files are typically stored in the application's */res* directory. The

subdirectories within the */res* are allowed to have *qualifiers* in their names to provide alternative resources for different device configurations, such as when the device's screen orientation is switched between portrait and landscape, and also for different screen resolutions.

3.1.5 Summary

Activities, Services, Broadcast Receivers, and Content Providers are combined together with an Android Manifest File and application resources to create an Android application. Intents and Content Resolvers are the primary means by which the four main components allow reuse, interoperability, and data sharing between applications, and even between different components of the same application. The Android Manifest file contains the declaration of all the components for a particular application, in addition to other requirements needed by the application. Application resources, along with XML layout files, are used to form the user interface of the application.

Now that we have an understanding of the typical Android application architecture, let us look into StarMapper, an Android application that follows the Android application structure discussed here.

3.2 StarMapper Activity Class

The top-level component of the application is the StarMapper Activity class. This Activity runs on application startup and instantiates many of the central classes required for the application to function, such as the GLSurface-

View object, and the main rendering class, called StarMapperRenderer, that runs the OpenGL graphics engine that supplies the main user interface. It also handles the hardware sensor setup, user preference settings, processes touch inputs from the user, retrieves Content Providers, and instantiates the User model class that represents data associated with the user of the application. Figures 3.3 and 3.4 show UML class diagrams of the StarMapper Activity and renderer, respectively. Some of the fields and methods in the classes have been removed for brevity and clarity in the diagram.

3.2.1 Device Location

One of the first tasks the StarMapper Activity performs upon startup is determine the device's current location. The Android System provides a *Location Service* that applications can access to achieve this. When accessed, it returns a LocationManager object. An additional object, called a *Criteria* object, can be used as a filter to help select the best content provider for location based on the settings of the Criteria object. For a location provider, the best provider would be able to return a location with the minimum accuracy required by the application. Two example location providers would be a wireless network location provider, which gives coarse coordinates, and GPS, which provides much more accurate results.

For the StarMapper application, it is necessary to achieve the highest accuracy available since the quality of the application is highly dependent on precise measurements and calculations of both the device and the celestial

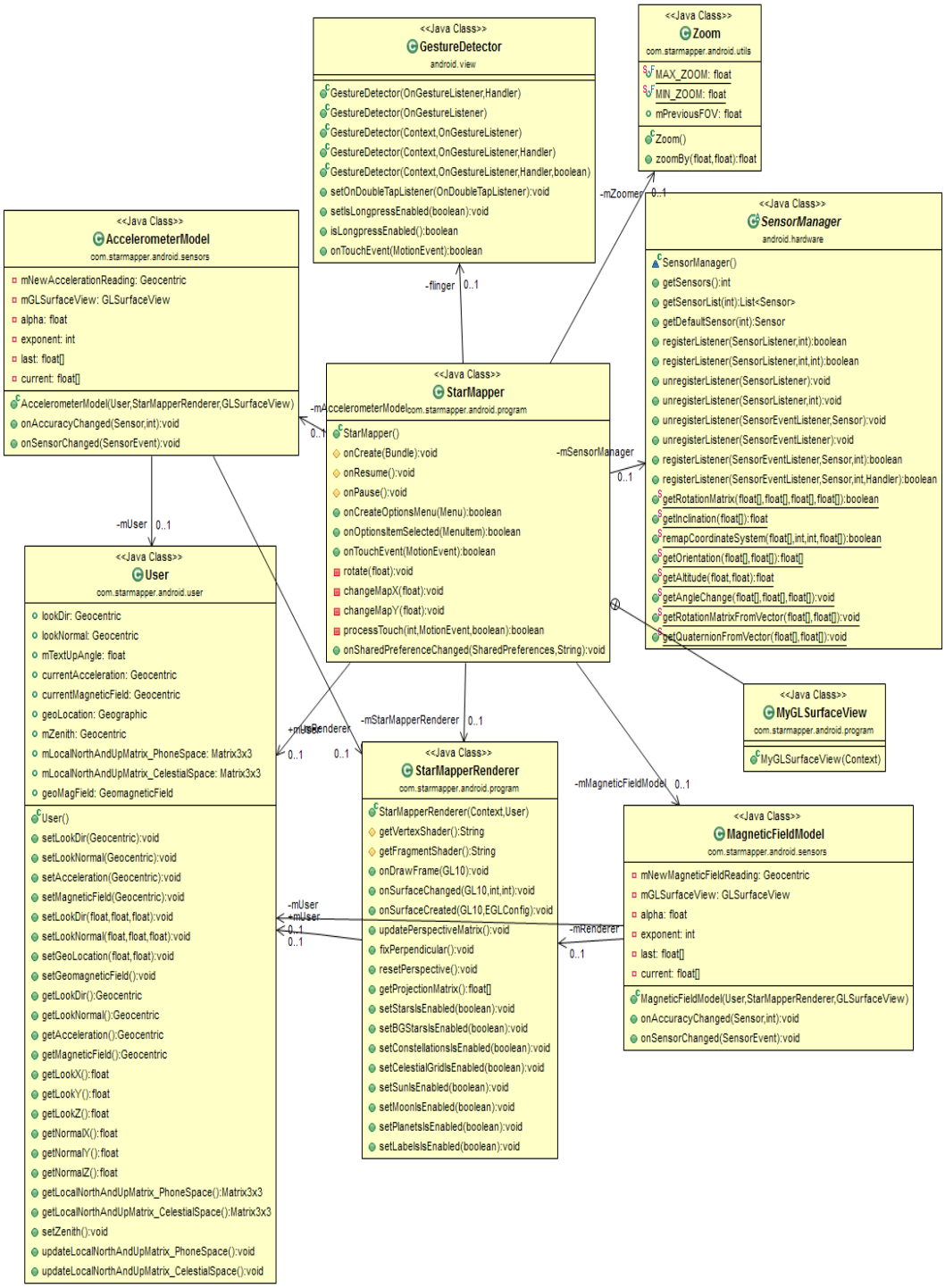

Figure 3.3: StarMapper Activity with Sensor, User, and Renderer Classes

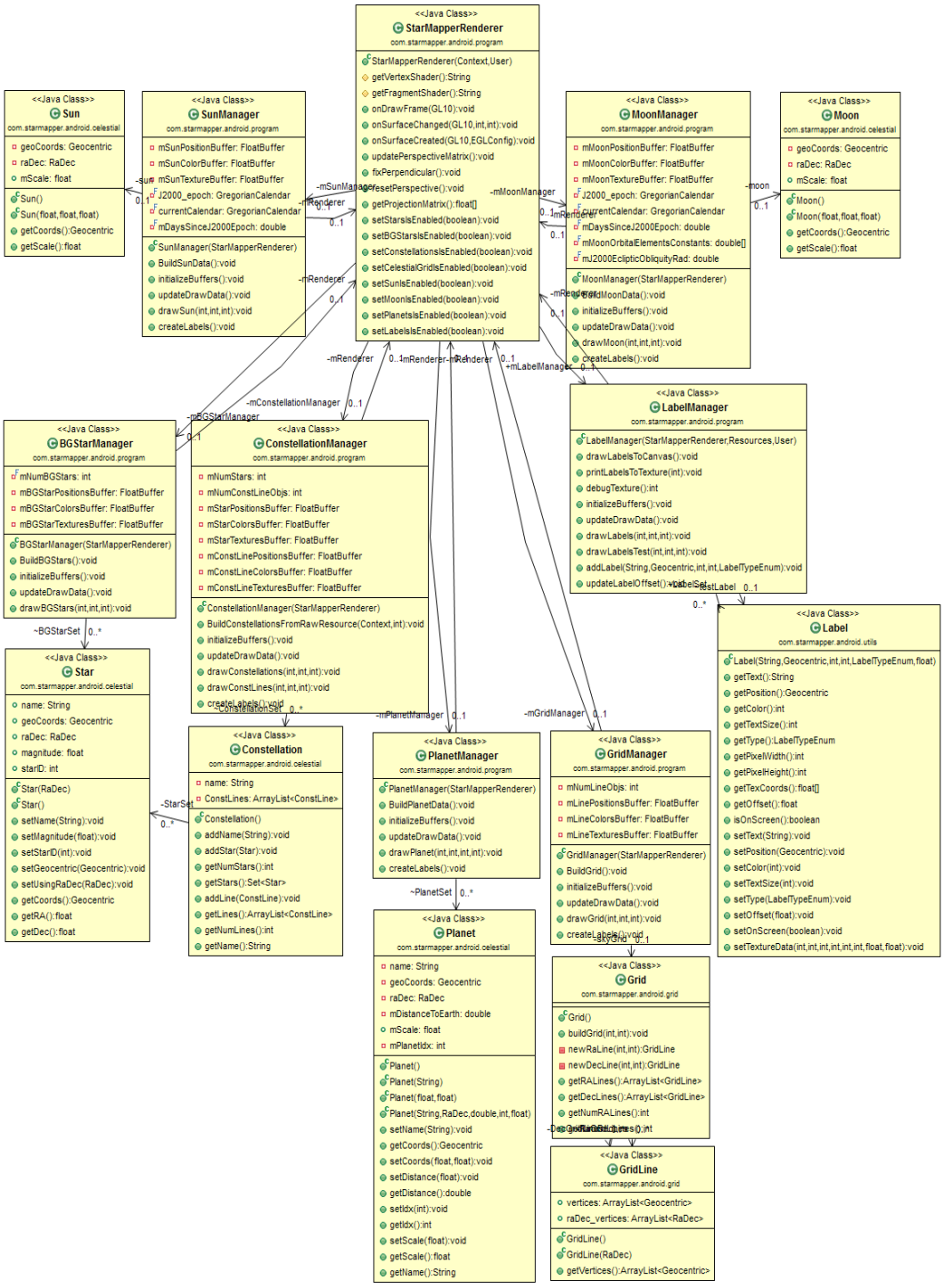

Figure 3.4: StarMapper Renderer with Celestial Object Classes and Managers

```
// Retrieving location
mLocationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
Criteria locCriteria = new Criteria();
// gps
locCriteria.setAccuracy(Criteria.ACCURACY_FINE);
locCriteria.setBearingRequired(false);
locCriteria.setAltitudeRequired(false);
locCriteria.setCostAllowed(true);
locCriteria.setSpeedRequired(false);
locCriteria.setPowerRequirement(Criteria.POWER_LOW);

String locProvider = mLocationManager.getBestProvider(locCriteria, true);
Location loc = mLocationManager.getLastKnownLocation(locProvider);

mUser.setGeoLocation((float) loc.getLatitude(), (float) loc.getLongitude());
```

Figure 3.5: StarMapper GPS Location Provider

objects. Because of this dependency, GPS is used since it provides the most accurate measurements. The current latitude and longitude of the device are then retrieved from the content provider, and stored for future use. Figure 3.5 shows the section of the StarMapper Activity that provides the GPS coordinates of the device.

3.2.2 Menu

The menu is where the user can adjust application settings such as which celestial objects to display on the map interface, and also where the user can adjust between auto and manual modes of the application. The auto mode utilizes the hardware sensors to determine what section of the sky is displayed on the device screen, and manual mode relies on touch input. Figure 3.6 shows the main options menu of StarMapper. Note that the Mode option is not currently used by the application, and so does nothing when clicked.

Figure 3.6: StarMapper Main Menu

3.2.3 User Settings

The menu of the StarMapper application is implemented through the use of a *Shared Preference* and a *Preference Fragment*. These objects provided by the Android system allow an easy interface between the hardware menu button on a device and the menu design the developer would like the application to use. Fig 3.7 shows StarMapper's settings page, which overlays the main map when open.

3.2.3.1 Shared Preference

A Shared Preference object is an application-internal content provider by which different components within an application keep the user's current settings in sync. One Shared Preference is instantiated for the entire application and is accessed by each component as needed. Upon instantiation, an XML file containing the application-specific settings is consumed by Android's PreferenceManager to set the user's default values. This allows for easy modification of the settings menu after creation. Figure 3.8 shows StarMapper's preference XML. A Listener for the Shared Preferences object is then registered with Android to detect and update any settings changes made by the

Figure 3.7: StarMapper Settings

user.

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android" >
    <PreferenceCategory
        android:title="@string/settings_title">
        <CheckBoxPreference
            android:key="settings_provider_stars"
            android:defaultValue="true"
            android:title="@string/settings_stars" />
        <CheckBoxPreference
            android:key="settings_provider_background_stars"
            android:defaultValue="true"
            android:title="@string/settings_background_stars" />
        <CheckBoxPreference
            android:key="settings_provider_constellations"
            android:defaultValue="true"
            android:title="@string/settings_constellations" />
        <CheckBoxPreference
            android:key="settings_provider_sun"
            android:defaultValue="true"
            android:title="@string/settings_sun" />
        <CheckBoxPreference
            android:key="settings_provider_moon"
            android:defaultValue="true"
            android:title="@string/settings_moon" />
        <CheckBoxPreference
            android:key="settings_provider_planets"
            android:defaultValue="true"
            android:title="@string/settings_planets" />
        <CheckBoxPreference
            android:key="settings_provider_celestial_grid"
            android:defaultValue="true"
            android:title="@string/settings_celestial_grid" />
        <CheckBoxPreference
            android:key="settings_provider_labels"
            android:defaultValue="true"
            android:title="@string/settings_labels" />
    </PreferenceCategory>
</PreferenceScreen>
```

Figure 3.8: StarMapper XML Preferences

3.2.3.2 Preference Fragment

The Preference Fragment controls the actual user settings in StarMapper. It operates as a modular section of the Activity in which it runs. It also consumes the XML file shown in Figure 3.8 to determine the list and hierarchy of preferences for the StarMapper application.

3.3 The StarMapper Renderer

The StarMapper Renderer class is an implementation of the *GLSurfaceView.Renderer* interface, and the main user interface to the StarMapper application. It utilizes the OpenGL engine as the graphics renderer, and handles all the associated setup and matrices involved with OpenGL, such as loading the texture data and initializing the vertex and fragment shader programs. It also contains the instantiations of each celestial object manager, and calls the methods of each manager to update the draw-related data of each object.

The three main methods that need to be implemented in the GLSurfaceView.Renderer interface are: *onSurfaceCreated*, *onSurfaceChanged*, and *onDrawFrame*. The onSurfaceCreated method is called when the renderer is created and is used as an initializer for OpenGL application-level settings, such as loading the texture data. The onSurfaceChanged method is called whenever the screen orientation changes, and can be used to control different view layouts for an application to use as the user changes how he is holding the device. The current width and height of the screen are passed into the method as parameters and saved by the StarMapper application as field variables to be used during OpenGL perspective matrix calculations. The OnDrawFrame method is called once per frame by the application. It is this method that calls all the draw methods of each object manager, which updates all the draw data and performs the actual draws to screen through the use of OpenGL API calls.

3.3.1 OpenGL for Embedded Systems

Android utilizes OpenGL for Embedded Systems (OpenGL ES) as a component within its core libraries. OpenGL ES is a subset of the OpenGL rendering API, optimized for use within embedded environments, such as the introduction of fixed-point data types, since embedded processors may not have a floating-point unit.

StarMapper uses OpenGL ES version 2.0, which removes the fixed-function pipeline dependency inherent in OpenGL ES version 1.x, and replaces it with a programmable pipeline, as shown in Figure 3.9. This allows

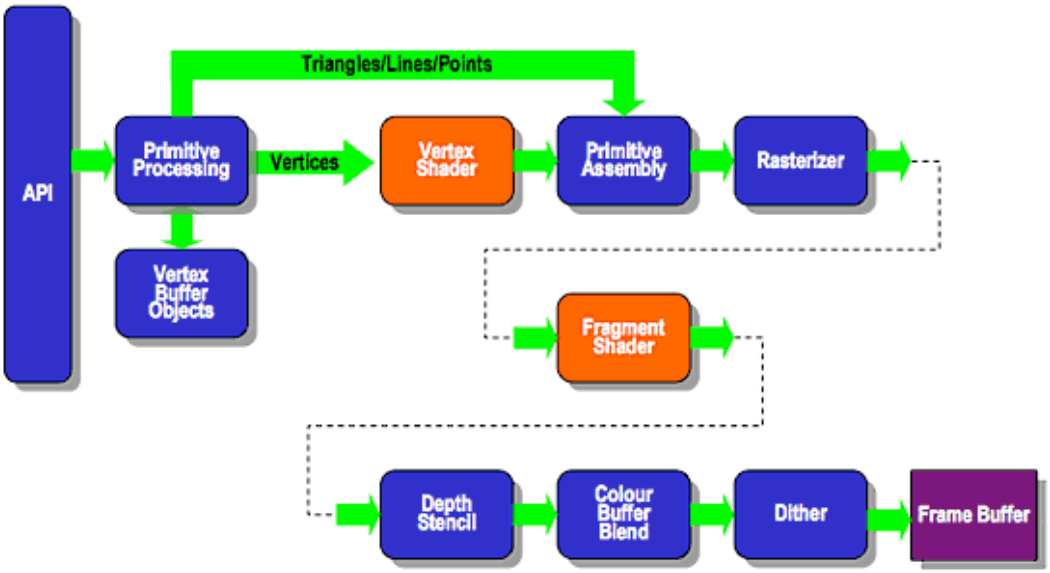

Figure 3.9: OpenGL ES 2.0 Programmable Pipeline

more freedom to the graphics programmers, but also results in more code since the pipeline must be implemented as custom shader programs [23]. The

orange boxes in Figure 3.9 show the placement of the vertex and fragment shaders written by the programmer within the graphics pipeline. Because of the shader program requirement, the OpenGL ES 2.0 version required by StarMapper must be declared in the Android manifest file. Furthermore, support for OpenGL ES 2.0 did not become available in Android until API level 8, so a required Android SDK version must also be specified in the manifest. Figure 3.10 shows the StarMapper specifications.

```
<uses-sdk
    android:minSdkVersion="17"
    android:targetSdkVersion="17" />
<uses-feature
    android:glEsVersion="0x00020000"
    android:required="true" />
```

Figure 3.10: StarMapper Required SDK/OpenGL Versions

3.3.1.1 Shader Programs

Shader programs are small programs written in GLSL (GL Shader Language) and compiled to run on the device's GPU [3]. StarMapper's shaders are set up inside the *onSurfaceCreated* method within the renderer. Setup includes passing the shader programs as strings into the OpenGL API for compilation, linking them to each other and the application, and binding the I/O of the shader programs to Java handles in the renderer.

StarMapper's vertex shader is responsible for translating the input vertices of each celestial object from object coordinates into normalized device

coordinates, where the X and Y axes ranges are (-1,1) [5]. To achieve this, an MVP (ModelViewProjection) matrix is passed into the vertex shader which acts as the vertex transformation matrix. The vertex shader is called once per vertex.

The fragment shader in StarMapper provides the color to objects that will be rendered. Unlike the vertex shader, the fragment shader is called once per fragment (pixel), with the possibility of doing color interpolation of fragments between vertices. Because the celestial object textures used in StarMapper already have accurate color, the fragment shader simply maps each color fragment to its corresponding color from the texture. No additional lighting or coloring effects are needed.

3.3.1.2 OpenGL Matrices

OpenGL matrices translate all objects to be rendered from their own space into the normalized device coordinates. The three main matrices used are the *model* matrix, *view* matrix, and *projection* matrix.

The model in the case of StarMapper is the representation of the entire sky. The sky can be thought of as a spherical shell that completely surrounds the user, and this shell acts as the model object for which the model matrix is defined. The model matrix shifts objects from object space, where the object is the center of the universe, into world space, where different objects can be placed relative to each other. Because the user is essentially always at the center of this sky shell, the model matrix needs only be set to the identity

matrix, and can be kept constant, as in Equation 3.1.

$$M_{model} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

The view matrix for StarMapper can be thought of as the 'camera' matrix, since it defines the eye location (where the camera is), the look direction (where the camera is pointing), and the normal to the look direction (to determine whether the camera is upside down). Unlike the model matrix, which is kept constant, the look and normal directions are constantly changing as the user looks around the sky. The view matrix is a primary determiner of what section of sky is currently displayed to the screen at any given time. Multiplying the vertices of any object by the combination of the model matrix and view matrix will convert the vertices into modelview space. In other words, it will translate the location of the vertices into their locations within the sky shell. The view matrix is given in Equation 3.2.

$$M_{view} = \begin{bmatrix} x_{eye} & x_{look} & x_{normal} & 0 \\ y_{eye} & y_{look} & y_{normal} & 0 \\ z_{eye} & z_{look} & z_{normal} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

The projection matrix in StarMapper is responsible for converting the sky shell, which has a 3D (spherical) context, onto a 2D device screen [1]. To achieve this, a gnomonic projection is used [25]. A gnomonic projection maps the surface of a sphere onto a flat surface, from the point-of-view of the center of the sphere. The projection matrix used in StarMapper is given in Equation

3.3.

$$M_{projection} = \begin{bmatrix} \frac{1}{A\tan(\frac{FOV}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{FOV}{2})} & 0 & 0 \\ 0 & 0 & -\frac{(Z_{far}+Z_{near})}{(Z_{far}-Z_{near})} & -\frac{2Z_{far}Z_{near}}{(Z_{far}-Z_{near})} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{3.3}$$

where

$$A = \text{Aspect Ratio}$$

$$FOV = \text{Field of View in radians}$$

$$Z_{far} = \text{Furthest projection depth}$$

$$Z_{near} = \text{Nearest projection depth}$$

The field of view (FOV) must be given in radians because the Java trigonometric methods operate on only radians [7].

The product of the model, view, and projection matrices is the MVP (ModelViewProjection) matrix, which is passed into the vertex shader, as shown in Equation 3.4.

$$M_{MVP} = (M_{model} \times M_{view}) \times M_{projection} \tag{3.4}$$

3.4 Celestial Object Managers

The celestial object managers maintain and update the data related to each of the celestial objects displayed by StarMapper's renderer, with one

manager class for each of the seven types of celestial objects in the application. Each manager is instantiated within the StarMapper Renderer class and initialized within the onSurfaceCreated method that is called when the Renderer class is instantiated in the StarMapper Activity. Initialization of each class involves building the object data of each manager, creating the text labels associated with each object, initializing all needed buffers for OpenGL API calls, an initial update of the draw data of each object, and loading the texture data. The following sections will describe in more detail each celestial object manager.

3.4.1 The Constellation Manager

The constellation manager handles the constellation data, which includes the major stars that comprise each constellation and the lines that connect them into the well-known figures such as Ursa Minor and Orion. The data for the stars in each constellation comes from the Yale Bright Star Catalogue, a star catalogue containing data for all stars with stellar magnitude of 6.5 or greater, which is roughly every star visible to the naked eye from Earth [10].

The data in the star catalogue includes information such as a star's apparent magnitude (the magnitude after factoring in atmospheric effects), the right ascension and declination, and the constellation region in which the star resides. The pertinent data from the star catalogue was translated into a text file that is parsed and processed during initialization of the constella-

tion manager. Each line of the text file contains all the data needed by the application for one constellation. It contains the constellation name, each major star's name (Bayer designation) [2], the right ascension, declination, and apparent magnitude of each major star. Each star is also assigned an integer ID that is used to identify the star connectivity of the lines that make up the constellation figure. The right ascension and declination are analogous to latitude and longitude; they are used to position the star on the celestial sphere. The magnitude is used to determine the size of the star texture on the screen, representing the brightness of the star.

A Constellation class was created to hold the star and line data for each constellation, and the constellation manager contains a set holding all of the constellation objects. During each frame, the buffers containing the draw data are updated with the latest data, then drawn using the OpenGL API draw calls. Figure 3.11 below shows the Orion constellation as it is drawn in the application, along with labels of two of its major stars, Betelgeuse and Rigel.

3.4.2 The Planet Manager

The planet manager handles all the planet data, which includes calculating and storing the real-time positions of each planet in the celestial sphere. Because the planets follow elliptical orbits around the sun, their paths can be represented mathematically with special factors known as the *planetary orbital elements*. The six orbital elements used to represent each planet are given in Table 3.1. Each orbital element consists of two components: the *mean* orbital

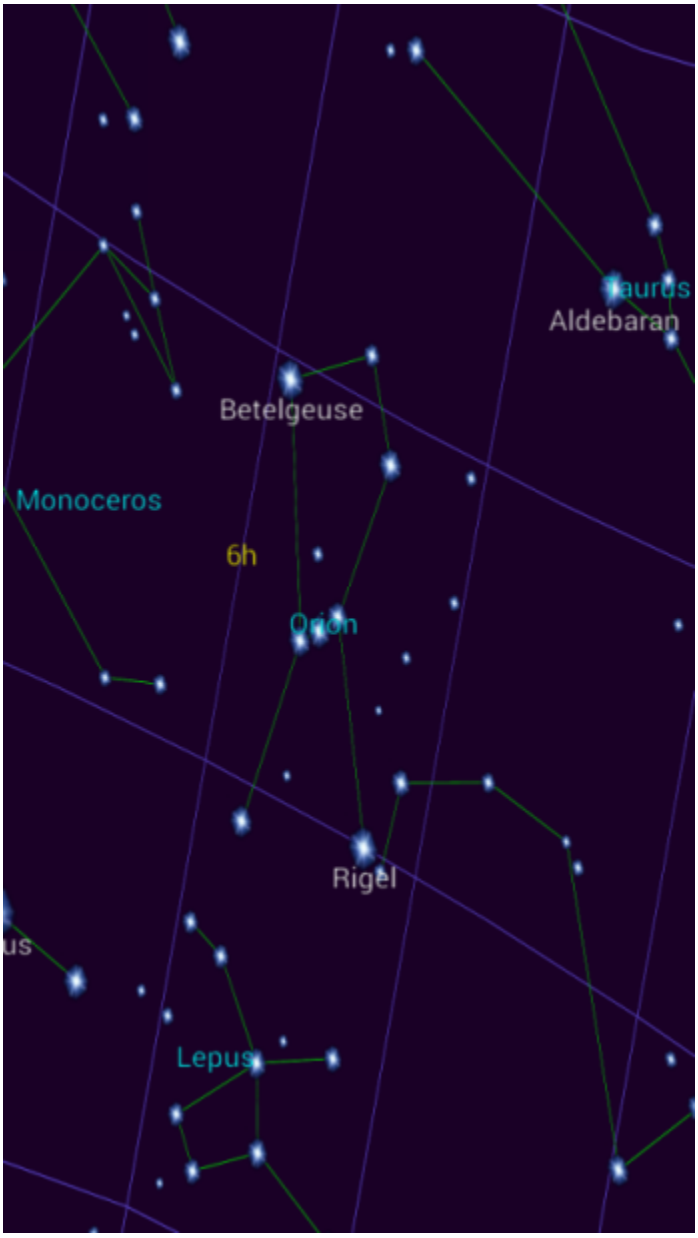

Figure 3.11: The Orion Constellation in StarMapper

Symbol	Name
a	Mean Distance from Sun
e	Eccentricity
i	Inclination
Ω	Ascending Node Longitude
ω	Perihelion Longitude
L	Mean Longitude

Table 3.1: The Planetary Orbital Elements

element value, given by E_{mean}, and the orbital element's *rate of change*, given by E_{roc}. Each orbital element E_{orb} is then given by

$$E_{orb} = E_{mean} + E_{roc}t \tag{3.5}$$

where t equals some amount of time.

If we assume some point in time as the reference point where $t = 0$, and we also know the position of the planet at $t = 0$, then we can calculate the position of the planet at any given time starting from $t = 0$. This reference point in time is known as an *epoch*. At the beginning of the epoch, the orbital elements are equal to their mean orbital element values

$$E_{orb} = E_{mean}$$

StarMapper uses the latest epoch available, called J2000.0, which began at approximately 12:00 GMT on January 1, 2000. The mean orbital element values and rates of change for each planet are precalculated and stored as constants for the J2000.0 epoch, and we need only plug in the time since

J2000.0 to calculate the real-time orbital elements for each planet. The current time is retrieved according to the device's time zone location using Android's GregorianCalendar and TimeZone classes. The J2000.0 time is then subtracted from the current time, and the resulting time is used to calculate and store each real-time orbital element for each planet. It is important to note that the precalculated mean orbital element values and rates of change lose accuracy over time due to gravitational perturbations and inexactitudes, so the latest available epoch should always be used in the calculations to minimize the error.

To calculate the real-time positions of each planet in the celestial sphere, Schlyter [15] has created simplified versions of Van Flandern's and Pulkkinen's Low-Precision Formulae for Planetary Positions [21]. We must first calculate the eccentric anomaly E. From Schlyter [15], we have

$$E = M + e\sin(M)(1.0 + e\cos(M)) \tag{3.6}$$

where

$$M = L - \omega$$

If more accuracy for the eccentric anomaly is required, we can set $E0 = E$, and use the iterative formula [15] given by Equation 3.7 until the delta between $E0$ and $E1$ is sufficiently close.

$$E = E0 - \frac{E0 - esin(E0) - M}{1 - ecos(E0)} \tag{3.7}$$

Next, the true anomaly, V, must be calculated. Again, from Schlyter [15], we have

$$V = 2\arctan\left(\sqrt{\frac{1+e}{1-e}}\tan\left(\frac{E}{2}\right)\right) \tag{3.8}$$

With the true anomaly, the heliocentric coordinates of the planets can be calculated as follows [15]:

$$X_{helio} = R_{helio}\cos(\Omega)\cos(V+\omega-\Omega) - \sin(\Omega)\sin(V+\omega-\Omega)\cos(i) \quad (3.9)$$

$$Y_{helio} = R_{helio}\sin(\Omega)\cos(V+\omega-\Omega) + \cos(\Omega)\sin(V+\omega-\Omega)\cos(i) \quad (3.10)$$

$$Z_{helio} = R_{helio}\sin(V+\omega-\Omega)\sin(i) \quad (3.11)$$

where R_{helio} is the heliocentric radius of the planet, and is given by

$$R_{helio} = \frac{a(1-e^2)}{(1+e)\cos(V)}$$

We now have the *heliocentric* (Sun-centered) coordinates of the planet. However, we want the *geocentric* (Earth-centered) coordinates of the planet. We can obtain ecliptic geocentric coordinates by subtracting the Earth's heliocentric coordinates from the planet's heliocentric coordinates [15], as in Equations 3.12, 3.13, and 3.14.

$$X_{geo,ecl} = X_{helio} - X_{helio,Earth} \quad (3.12)$$

$$Y_{geo,ecl} = Y_{helio} - Y_{helio,Earth} \quad (3.13)$$

$$Z_{geo,ecl} = Z_{helio} - Z_{helio,Earth} \quad (3.14)$$

Earth's heliocentric coordinates can be calculated using Equations 3.9, 3.10, and 3.11.

Because the Earth travels along its own orbit at an angle to the celestial equator, and not directly on the celestial equator, we must account for this angle when converting to geocentric coordinates on the celestial sphere. This

angle between Earth's orbit and the celestial equator is called the *ecliptic obliquity*, and is shown in Figure 3.12. The ecliptic obliquity, ϵ, is calculated

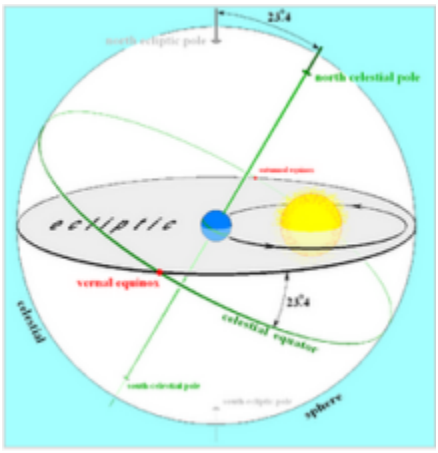

Figure 3.12: Earth's Ecliptic Obliquity

on a per-epoch basis, and is considered a constant for that epoch. The value of the ecliptic obliquity for the J2000.0 epoch and used by StarMapper is given in Equation 3.15.

$$\epsilon = 23.439281 \text{ (degrees)} \tag{3.15}$$

The final geocentric coordinates of the planet, from Schlyter [15], are then given by

$$X_{geo} = X_{geo,ecl} \tag{3.16}$$

$$Y_{geo} = Y_{geo,ecl} \cos(\epsilon) - Z_{geo,ecl} \sin(\epsilon) \tag{3.17}$$

$$Z_{geo} = Y_{geo,ecl} \sin(\epsilon) + Z_{geo,ecl} \cos(\epsilon) \tag{3.18}$$

From these, the planet's right ascension and declination can be easily calcu-

lated [15]:

$$RA = \arctan\left(\frac{Y_{geo}}{X_{geo}}\right)$$
$$Dec = \arctan\left(\frac{Z_{geo}}{\sqrt{X_{geo}^2 + Y_{geo}^2}}\right)$$

A Planet class was created to hold all the planet position data calculated by the manager, and the manager contains a Planet set that holds all the planet objects. Each frame, the planet manager's draw method is called to draw the planets to the screen in their calculated positions. Figure 3.13 below is a screenshot of StarMapper showing the planet Mars at the top of the screen, along with the constellations Crater, Corvus, and Virgo.

3.4.3 The Sun Manager

The sun manager is similar to the planet manager in that it calculates and stores the real-time position of the sun in the celestial sphere. The calculations are also similar to the planets in that the sun also has *orbital element values* that are used to determine its position. While in actuality the Earth orbits the Sun, we can treat the Earth as if it is centered, and the Sun orbits around it. This allows us to represent the orbit of the Sun using the orbital elements. In addition, the calculations are made simpler by the fact that we do not need to do heliocentric-to-geocentric coordinate conversions, since representing the Earth as centered will allow us to calculate geocentric coordinates by definition.

To calculate the sun's real-time position, we introduce a few new orbital

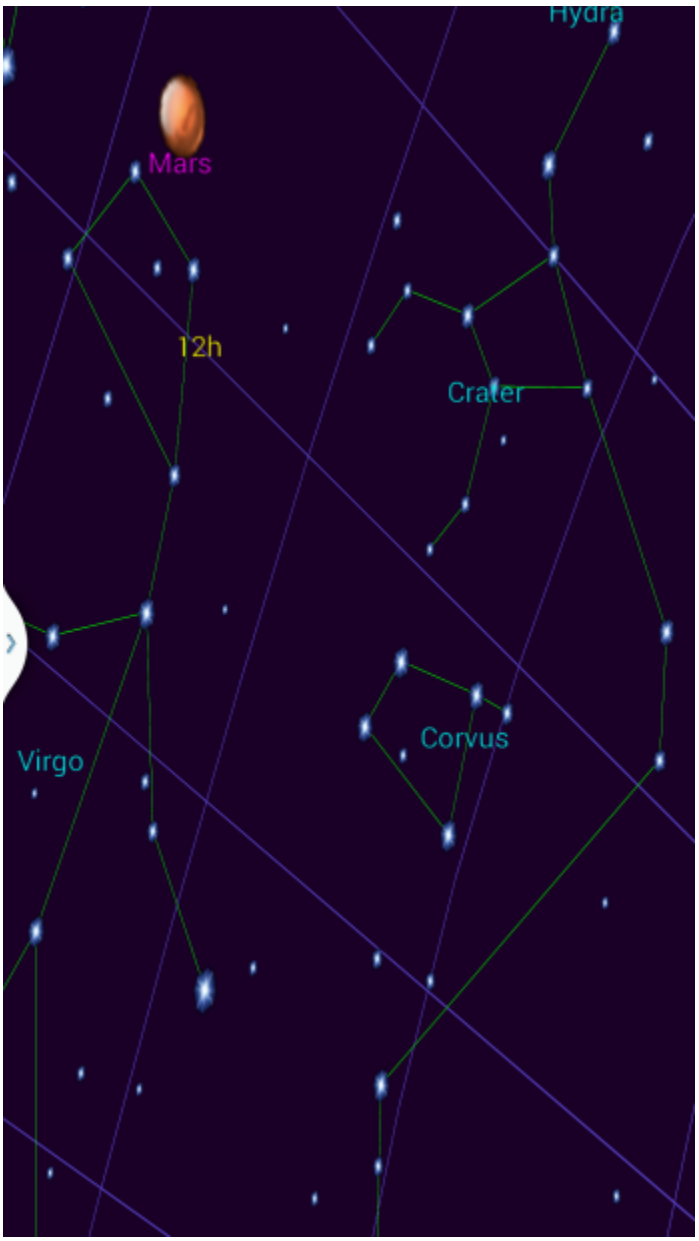

Figure 3.13: Screenshot of the Planet Mars in StarMapper

elements in Table 3.2. The sun manager uses the same epoch (J2000.0) and

Symbol	**Name**
g	Mean Anomaly of Sun
q	Mean Longitude of Sun
L	Geocentric Apparent Ecliptic
e	Obliquity of Sun's Ecliptic

Table 3.2: Orbital Elements of the Sun

current time (GregorianCalendar and TimeZone-based) as the planet manager. The real-time orbital element values for the sun can be calculated using Equation 3.5. Using precalculated values from the U.S. Naval Observatory Astronomical Applications Department (USNOAAD) [4], we have

$$g = 357.529 + 0.98560028t$$

$$q = 280.459 + 0.98564736t$$

where t is the time since the J2000.0 epoch, and, from the USNOAAD [4],

$$L = q + 1.915\sin(g) + 0.020\sin(2g)$$

The sun's ecliptic obliquity can be calculated by

$$e = 23.429 - 0.00000036t$$

Then the sun's real-time right ascension and declination can be calculated [4]:

$$RA = \arctan\left(\frac{\cos(e)\sin(L)}{\cos(L)}\right)$$
$$Dec = \arcsin\Big(\sin(e)\sin(L)\Big)$$

An instantiation of the Sun class in the sun manager holds the sun object along with the real-time position data. Like the other managers, the sun manager's draw method is called once per frame from the *onDrawFrame* method in the Renderer class. Figure 3.14 shows the sun as it is rendered in StarMapper.

3.4.4 The Moon Manager

The moon manager handles the position calculations and data storage for the moon in the celestial sphere. While the calculations of the moon's position may seem more complicated due to the moon orbiting around both the Earth and the Sun, they are actually simplified by the fact that we only care about the orbit in relation to Earth. Because of this, we can use the exact same equations described in the Section 3.3.2 *The Planet Manager*, with one main exception, the orbital element a. For the planets, a is equal to the mean distance from the sun. However, for the moon, a is equal to the mean distance from the Earth. By making this substitution, Equations 3.9, 3.10, and 3.11 give the *geocentric* coordinates of the moon, rather than *heliocentric*.

While the geocentric coordinates are directly returned, they must still be adjusted for the ecliptic of the Earth. Furthermore, because the moon's position is tied to the Earth, which travels along the ecliptic, we cannot simply use the ecliptic obliquity constant as in Equations 3.16, 3.17, and 3.18. We must account for the *longitudinal ecliptic*, $\epsilon_{moon,lon}$, and *latitudinal ecliptic*, $\epsilon_{moon,lat}$, of the moon relative to the Earth's ecliptic. Given that the moon's

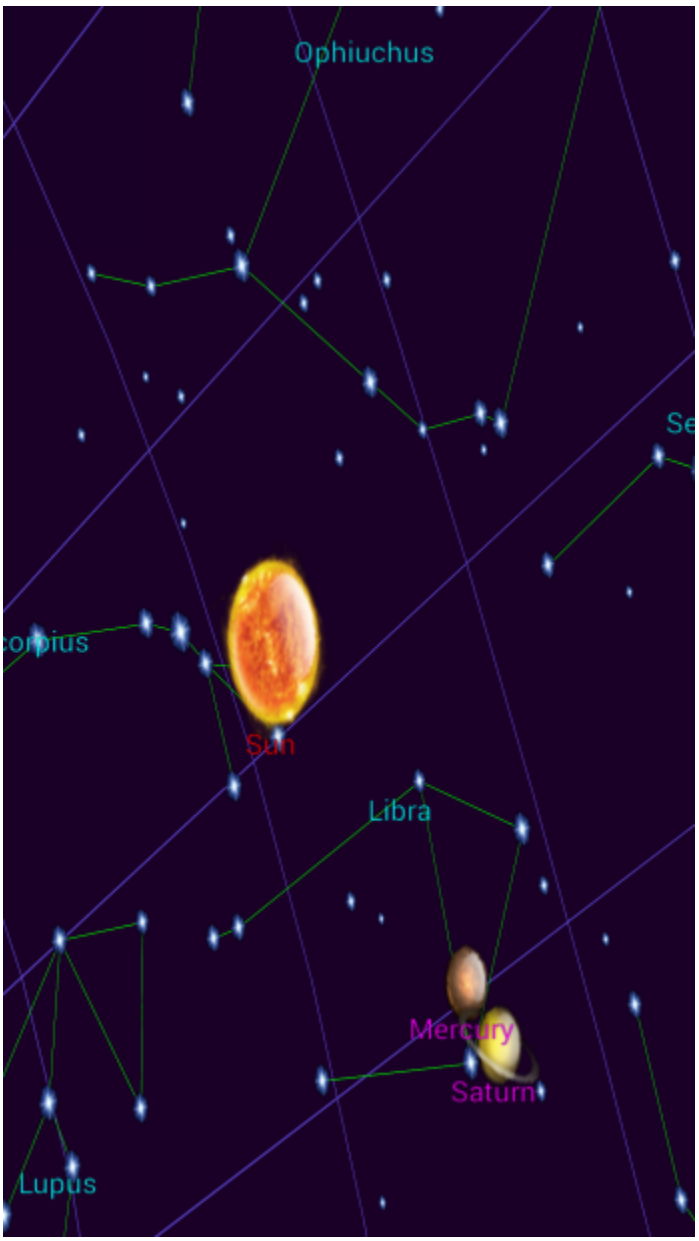

Figure 3.14: Screenshot of the Sun in StarMapper

ecliptic coordinates $X_{moon,ecl}$, $Y_{moon,ecl}$, and $Z_{moon,ecl}$ are calculated using Equations 3.9, 3.10, and 3.11 respectively, from Schlyter [15], the moon's ecliptics are calculated as follows:

$$\epsilon_{moon,lon} = \arctan\left(\frac{Y_{moon,ecl}}{X_{moon,ecl}}\right)$$
$$\epsilon_{moon,lat} = \arctan\left(\frac{Z_{moon,ecl}}{\sqrt{X^2_{moon,ecl} + Y^2_{moon,ecl}}}\right)$$

The geocentric ecliptic coordinates of the moon can then be calculated [15]:

$$X_{geo,ecl} = a\cos(\epsilon_{moon,lon})\cos(\epsilon_{moon,lat})$$
$$Y_{geo,ecl} = a\sin(\epsilon_{moon,lon})\cos(\epsilon_{moon,lat})$$
$$Z_{geo,ecl} = a\sin(\epsilon_{moon,lat})$$

Finally, from Schlyter [15], the geocentric coordinates of the moon are calculated using the ecliptic obliquity constant from Equation 3.15:

$$X_{geo} = X_{geo,ecl}$$
$$Y_{geo} = Y_{geo,ecl}\cos(\epsilon) - Z_{geo,ecl}\sin(\epsilon)$$
$$Z_{geo} = Y_{geo,ecl}\sin(\epsilon) + Z_{geo,ecl}\cos(\epsilon)$$

And the right ascension and declination [15]:

$$RA = \arctan\left(\frac{Y_{geo}}{X_{geo}}\right)$$
$$Dec = \arctan\left(\frac{Z_{geo}}{\sqrt{X^2_{geo} + Y^2_{geo}}}\right)$$

Because of the small size of the moon relative to the other masses in the solar system, and its close proximity to the Earth, the moon is also affected by other perturbations, such as the *Evection* and the *Variation* that cause a small margin of error in regards to the previous calculations. While it is possible to account for these errors through additional equations, the position calculations up to this point will always be within two degrees of their true value [15]. Because of the complexity of adding the perturbation calculations for a minor improvement in accuracy, it was decided that two degrees of error in the moon's position was acceptable. Also, changes in the waxing and waning of the moon were not included in the moon manager. Showing the current phase of the moon is a prime candidate for future enhancement of the StarMapper application.

The Moon class is instantiated in the moon manager and holds data relevant to the moon. Figure 3.15 shows the moon in StarMapper.

3.4.5 The Grid Manager

The grid manager is responsible for creating and drawing the celestial grid lines representing the right ascension and declination coordinates in StarMapper. There are a total of 24 right ascension lines (one per 15 degrees) and 10 declination lines (one per 18 degrees). Each of the grid lines is separated into segments with vertices at each intersection of segments. This can give a line the appearance of bending if needed. The more vertices a line has, the smoother the bending will appear.

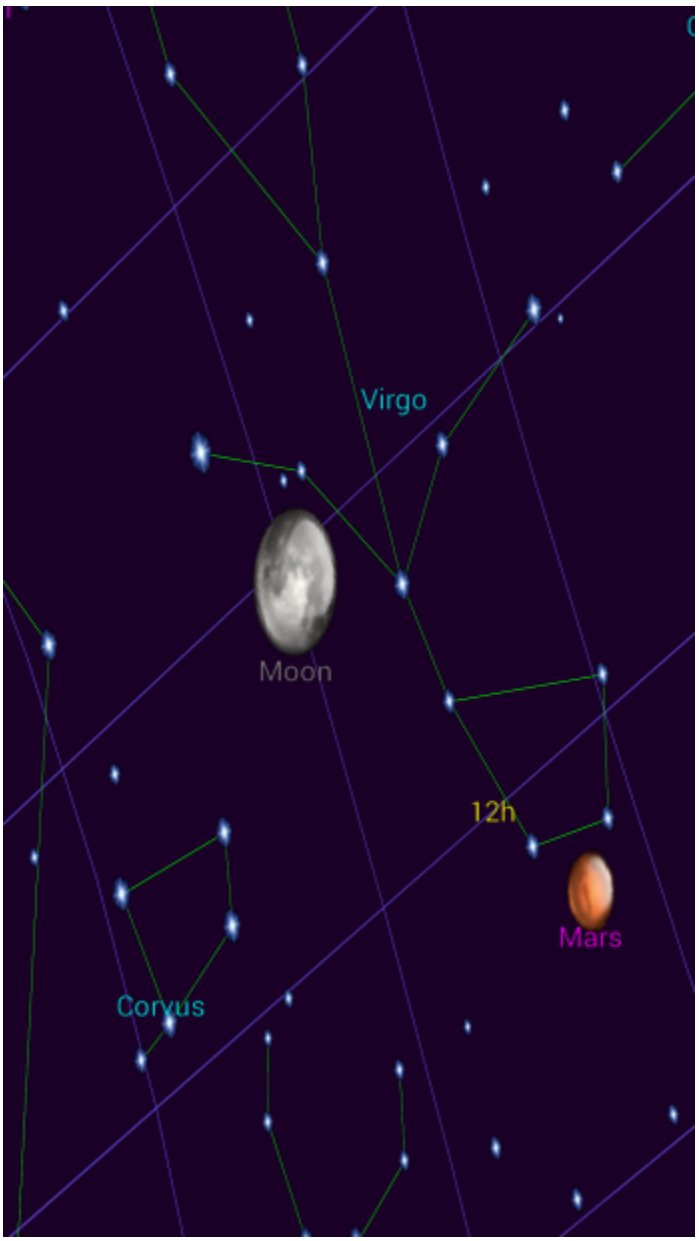

Figure 3.15: The Moon in StarMapper

The right ascension lines represent great circles on the celestial sphere, and so will always appear to be straight lines in a gnomonic projection. Each right ascension line therefore only has three vertices, one at each celestial pole, and one at the celestial equator. The declination lines each have thirty-six vertices, since they will always appear to bend except at the celestial equator.

A Grid class instantiated inside the grid manager stores all the data relevant to the grid, for both right ascension and declination lines. Figure 3.16 shows the grid in StarMapper.

3.4.6 Background Star Manager

The background star manager randomly generates stars that are not part of any constellations to make the map display more visually appealing. Each of the stars that are part of the background have an apparent magnitude of between 4.5 and 6.0. This makes them generally less bright than all except the smallest constellation stars.

The random star generator randomly creates a right ascension and declination value for each background star. Due to the structure of the celestial sphere, a weighted random generator needed to be used. A linear random generator would have resulted in stars that appeared bunched at the celestial poles, and sparse near the celestial equator. Therefore, the celestial sphere was divided into 11 declination zones. A number was randomly generated between 0 and 100 that determined which declination zone a star was placed in according to Table 3.3. The same Star class that is used by the constellation

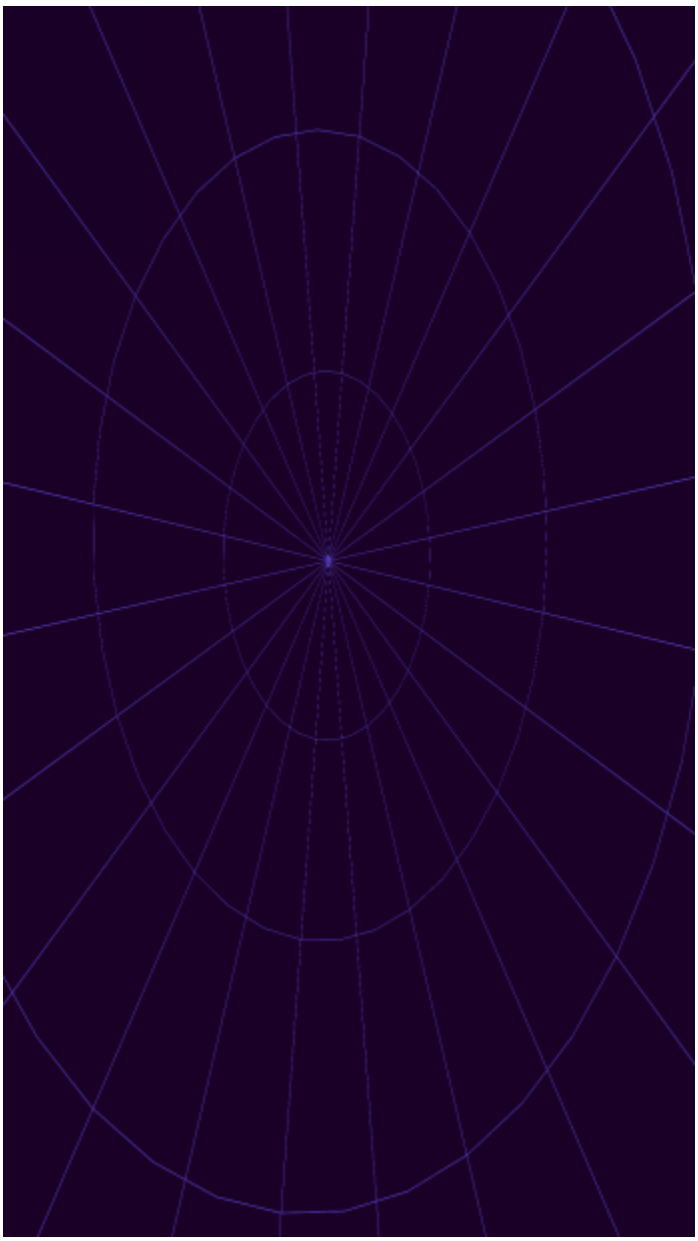

Figure 3.16: The Celestial Grid in StarMapper

Random Number	Declination Zone
0-3	0
4-9	1
10-18	2
19-30	3
31-42	4
43-57	5
58-69	6
70-81	7
82-90	8
91-96	9
97-100	10

Table 3.3: Declination Zone Placement of Randomly Generated Background Stars

manager is used to create and store the data of each background star. A set of these stars is then stored in the background star manager. Figure 3.17 shows some background stars in StarMapper.

3.5 Touch Input

The StarMapper application is designed to be user interactive, and part of that interaction involves the touch interface that is integrated into most mobile devices today. The StarMapper Activity class has a method for handling touch events called *onTouchEvent* that is passed a MotionEvent object as a parameter. The MotionEvent object describes the type of touch being passed into the method. How the onTouchEvent handles the touch input is dependent on both the type of touch that occurred and the mode of operation

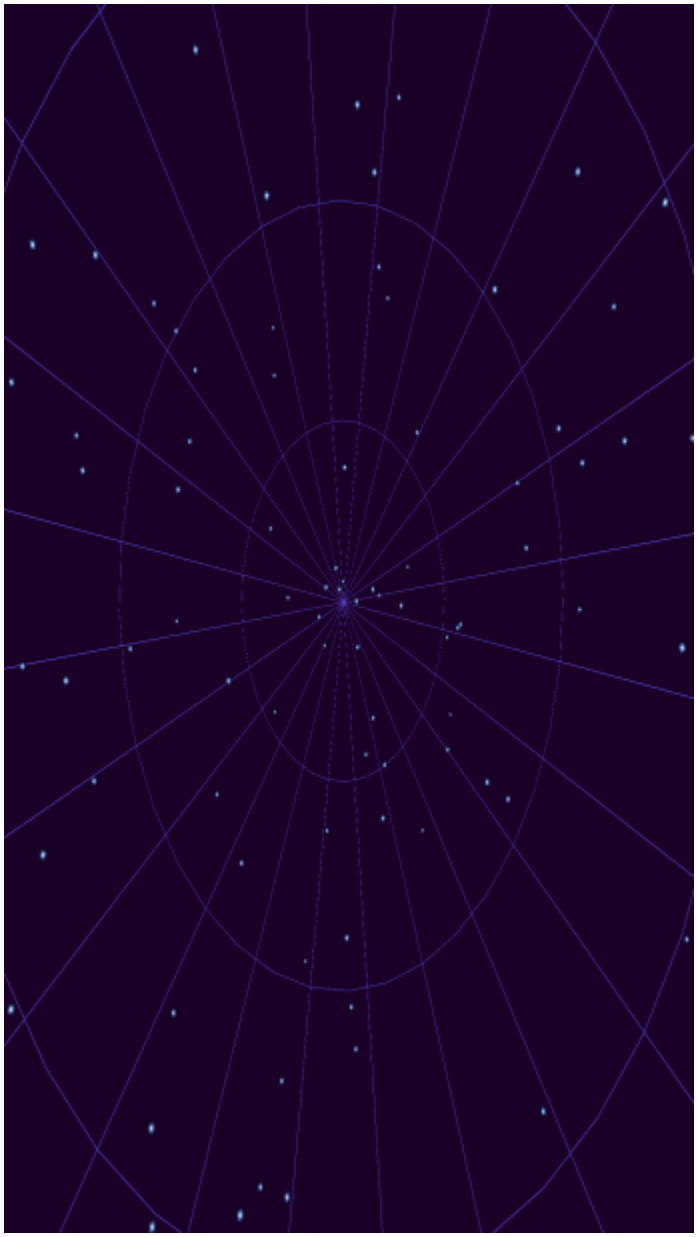

Figure 3.17: Background Stars in StarMapper

that StarMapper is in at the time of the touch input. The only exception is clicking on celestial object labels. In this case, the mode of operation is irrelevant.

The two modes of operation that StarMapper can operate in are auto and manual mode. The user can alternate between modes by repeatedly selecting the auto/manual screen button on StarMapper's main options menu (see Figure 3.6). In auto mode, the application uses the hardware sensors to determine what section of the star map to display on the device screen. This is based on how the user is holding or pointing the device. In manual mode, the user applies touch inputs to move around the sky map, with additional display features added specifically for manual-mode. The following sections will describe in more detail how touch input is handled based on the mode of operation.

3.5.1 Manual Mode Touch

In manual mode, the user must touch the device screen to navigate around the star map. If the user simply presses down on the screen with a single finger and drags it deliberately a detectable distance, the map will be dragged in the same direction accordingly. The hardware sensors are also disabled when in manual mode. This is done by unregistering the listeners associated with each hardware sensor. Changes in sensor readings are then ignored by StarMapper.

Certain navigation features for StarMapper are available only when the

application is in manual mode. They include *Zoom*, *Rotation*, and *Flinging*. Each of these navigation features are described below.

3.5.1.1 Zoom

The zoom feature allows the user to zoom in and out to see more or less of the star map on the screen. The maximum field of view when zooming out (min zoom) is 90 degrees, and the minimum field of view when zooming in (max zoom) is 15 degrees. Using the zoom feature requires two simultaneous touches of the device screen. To zoom in, the two touches must drag apart diagonally in opposite directions, one towards the bottom left of the screen, the other towards the upper right. To zoom out, both touches must drag towards the center of the screen, one from the bottom left, the other from the upper right. The zoom feature is implemented as a separate Zoom class to give better encapsulation, allowing the min/max zoom limits to be easily modified. The Zoom class is instantiated inside the StarMapper Activity and utilized inside the onTouchEvent method. It implements a method called ZoomBy that handles both zooming in and out, and returns the updated field of view for the application after zoom.

3.5.1.2 Rotation

The rotation feature allows the user to rotate the star map around an axis that projects perpendicular to the device screen (the Z-axis). Like the zoom feature, it also requires two simultaneous touches of the device screen.

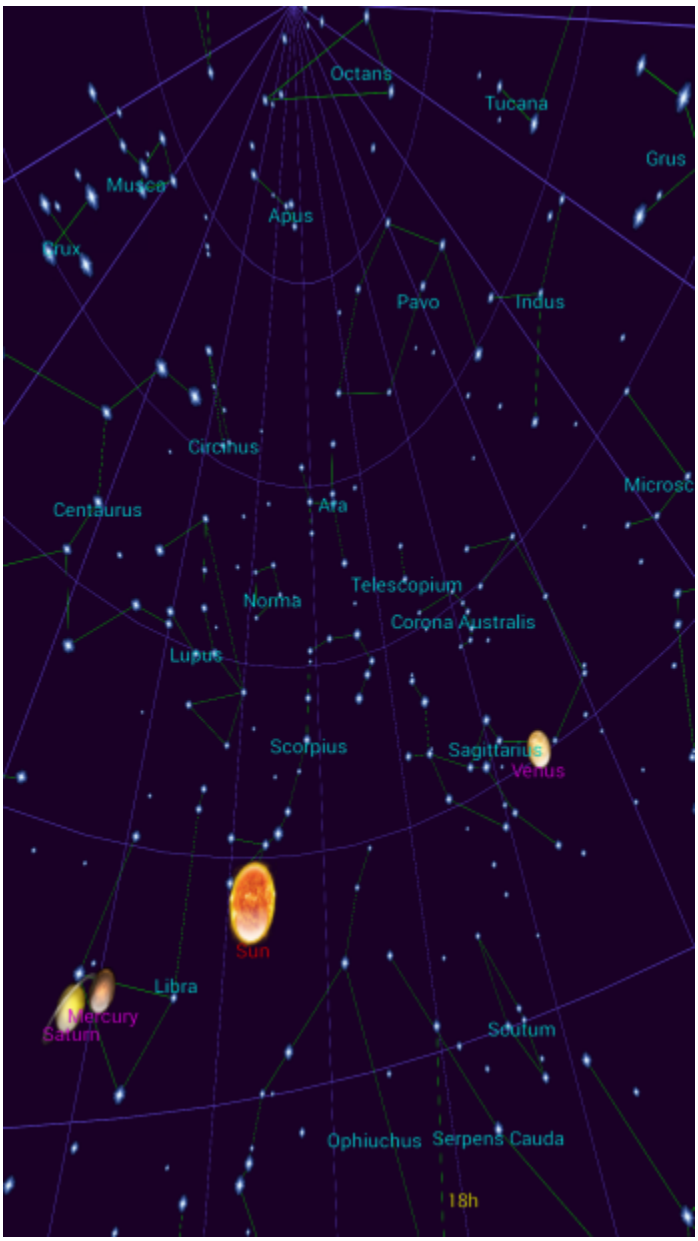

Figure 3.18: StarMapper in Minimum Zoom

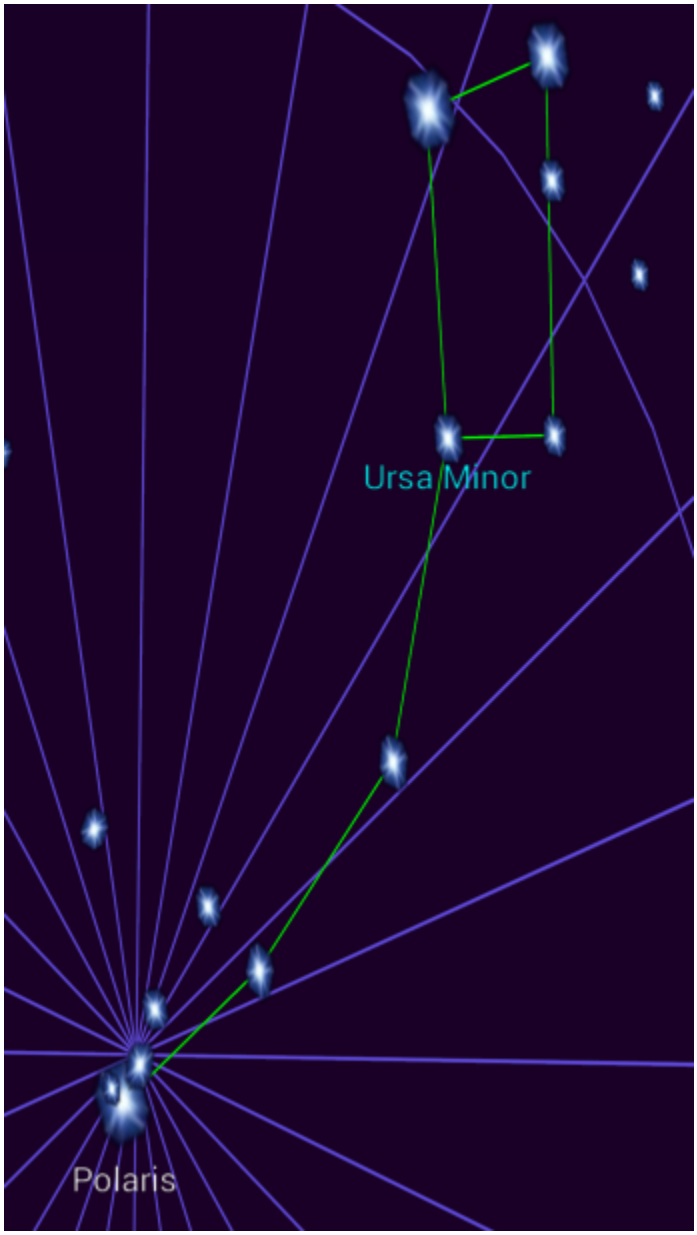

Figure 3.19: Maximum Zoom on Ursa Minor (The Little Dipper)

However, the dragging motion of the touches for rotation is distinct from zooming. Whereas zooming requires diagonally dragging apart or together, rotation requires rotating the two touches in either a clockwise or counter-clockwise manner. The star map will rotate accordingly, while maintaining the same field of view.

The mathematics behind the rotation feature is complex, requiring the creation of a trigonometric rotation matrix based on the angle of rotation that is then used for matrix/vector multiplication, but essentially the rotation feature updates the Normal vector in the OpenGL View matrix (see Equation 3.2) while maintaining the Look and Eye vectors.

3.5.1.3 Flinging

Flinging is a feature that simulates inertia in the star map. It is activated by pressing down with one finger on the device screen, then quickly sliding and releasing. The result is that the star map is flung a certain distance, depending on how fast the user slid across the screen. The star map will initially move quickly, then slow down over time, and eventually come to a complete stop. It is useful for quickly navigating across the star map in manual mode.

Flinging is defined in a separate Flinger class and is an extension of a GestureDetector Listener. The onFling GestureDetector method is overridden to customize the amount of inertia the star map simulates. The actual flinging task runs on a separate, concurrent thread under a Java Scheduled Executor

Service. This Service will run the fling task periodically until the star map's calculated inertia (a measure of the rate of change in the X and Y direction) falls below a certain threshold. The concurrent thread of the Executor Service is necessary, since the flinging task cannot run on the main thread or it would freeze the application while waiting for each screen refresh from the fling.

3.5.2 Celestial Object Labels

Clicking the celestial object labels will open up the Android web browser to the Wikipedia page of the object whose label was clicked, regardless of whether the application is currently in auto or manual mode. The label clicking feature is designed as a quick way for the user to obtain information about a celestial object of interest in the star map. Figure 3.20 shows the mobile Wikipedia page of the sun after clicking on the Sun label in StarMapper.

The celestial object label clicking feature works by keeping track of the X and Y coordinate box where each label resides. If the user clicks somewhere in a label's box, then releases within a twenty pixel threshold of where the click occurred, the Wikipedia URL is built into a string and passed into an Intent, which itself is passed into a *startActivity* method. Android will then parse the URI within the Intent and open the web browser to the Wikipedia page specified by the URL packaged inside the Intent.

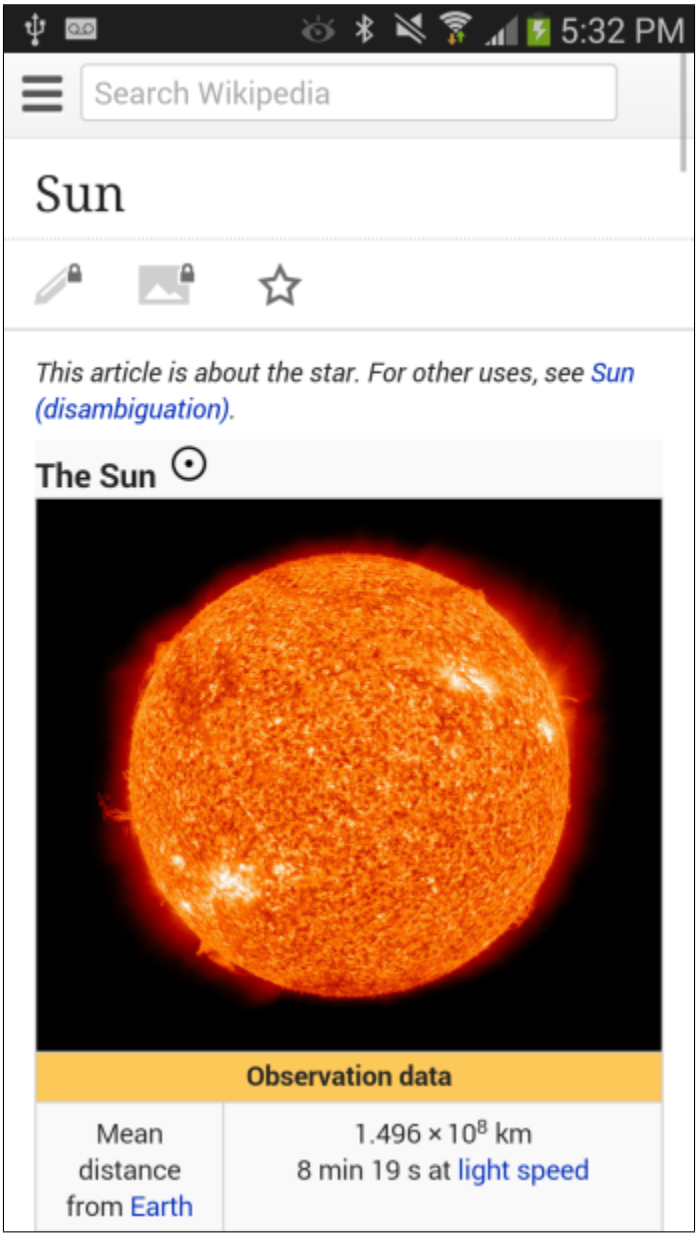

Figure 3.20: Wikipedia Page of the Sun

3.6 Hardware Sensors

There are two hardware sensors used in auto mode of the StarMapper application: the accelerometer sensor and the magnetic field sensor. The accelerometer sensor determines the tilt of the device, such as if it were flat on a table, or perpendicular to the floor, or at some angle inbetween. The magnetic field sensor is used to determine the direction of North relative to where the device is located. With these two readings, along with the GPS coordinates of the device and the current time, StarMapper is able to accurately map the stars to the device screen, revealing the constellations and other celestial objects, as the user points the device around the sky.

An AccelerometerModel and MagneticFieldModel class are created to model the hardware sensors and provide sensor data to StarMapper by implementing sensor event listeners.

3.6.1 Sensor Service

When the StarMapper application is first started, an instance of Android's Sensor Service is retrieved. The Sensor Service contains sensor information for the current device, including which sensors are available. StarMapper checks to ensure both an accelerometer and magnetic field sensor are available; if not, auto mode is disabled and the application only runs in manual mode. Otherwise, listeners for the hardware sensors are registered with the Sensor Service. The sensors then provide continuous readouts that can be retrieved by overriding each sensor's *onSensorChanged* method.

3.6.2 Low-Pass Filter Sensor Smoothing Algorithm

Because the sensors provide continuous readouts, using the raw data results in a jittery screen from sporadic outlier readings. This jitter can be resolved through the use of a low-pass filter. The filter mitigates the impact of outlier readings by giving weight to each reading. Thus, the cumulative measurement of previous readings is not affected as much by the outliers, resulting in a smoother screen. From Nichols [12],the algorithm code for the low-pass filter used by both sensors is given in Figure 3.21.

```
//smoothing code
for (int i = 0; i < 3; ++i) {
    last[i] = current[i];
    float diff = event.values[i] - last[i];
    float correction = diff * alpha;
    for (int j = 1; j < exponent;  ++j) {
        correction *= Math.abs(diff);
    }
    if (correction > Math.abs(diff) || correction < -Math.abs(diff)) {
        correction = diff;
    }
    current[i] = last[i] + correction;
}
```

Figure 3.21: Algorithm Code for Low-Pass Filter

3.7 The User Model

The user model represents the user of the application. It is defined as a class named User and is instantiated in the top-level StarMapper Activity during application startup. The user model stores information such as the latitude and longitude of the user (obtained from the location provider men-

tioned in Section 3.1.1), the current acceleration and magnetic field vectors of the user's device, and the zenith vector from the user's location.

The user instance in the StarMapper Activity is defined as a public field because many other classes constantly interact with it. For example, each sensor readout updates its respective vector stored in the user instance, which are then used to create transform matrices that update the user's look and normal vectors. The renderer also accesses the look and normal vectors to update the OpenGL view matrix during each frame update.

Chapter 4

Conclusion

This report has described StarMapper, an Android application that maps the celestial sky and provides users with information about celestial objects. In this chapter, we conclude the report with a project development discussion of StarMapper, along with some key learnings and possible future enhancements.

4.1 Development Overview

The project duration was approximately five months, spanning July 2013 – November 2013. Table 4.1 divides the timeline by component. The time listed for each section includes the time needed to obtain or research all necessary and relevant information, in addition to time needed for code development. For example, the Constellations row lists 2.0 weeks committed toward development. This includes the time needed to obtain the Yale Bright Star Catalog, use it to create the text file of constellation data, create the parser method for the text file, and develop the Constellation and Constellation Manager classes.

The StarMapper application was developed in the Eclipse Integrated

Project Component	**Time**
StarMapper Activity	
User Model	1.0 weeks
Preferences	1.0 weeks
Renderer	
OpenGL	2.5 weeks
Constellations	2.0 weeks
Stars	1.5 weeks
Planets	1.5 weeks
Sun	0.5 weeks
Moon	1.0 weeks
Grid	0.5 weeks
Labels	1.5 weeks
Sensors	
Accelerometer	1.0 weeks
Magnetic Field	1.0 weeks
Touch Input	
Drag	0.5 weeks
Fling	0.5 weeks
Zoom	0.5 weeks
Rotate	1.0 weeks
Wikipedia	1.0 week
Utility Classes	1.5 weeks
TOTAL:	20.0 weeks

Table 4.1: Project Development Timeline

Development Environment (IDE) for Windows with Android SDK Ver. 21 plugin. The StarMapper application is divided into several packages for better organization and encapsulation. Table 4.2 below shows the packages along with their size in LOC (lines of code). The total LOC count is 5161 lines. The *program* package is by far the largest because it contains the top-level StarMapper Activity class, StarMapper Renderer class, and all the celestial object Manager classes.

Package	**Size** (LOC)
com.starmapper.android.celestial	289
com.starmapper.android.constants	421
com.starmapper.android.grid	100
com.starmapper.android.math	149
com.starmapper.android.program	3042
com.starmapper.android.sensors	204
com.starmapper.android.user	181
com.starmapper.android.utils	775
TOTAL:	5161

Table 4.2: StarMapper Packages and LOC Sizes

Initial mock-ups of the user interface were created during the planning stages of the StarMapper project, and used as a guide during development. Figures 4.1, 4.2, and 4.3 show some initial ideas for how the StarMapper application would look.

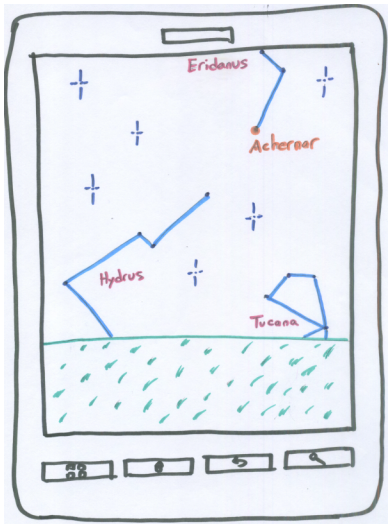

Figure 4.1: UI Mock of the StarMapper Map

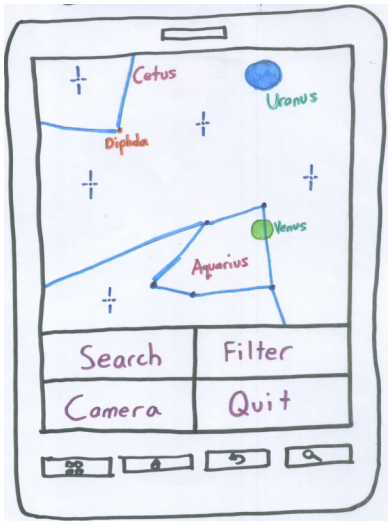

Figure 4.2: UI Mock of the Main Menu in StarMapper

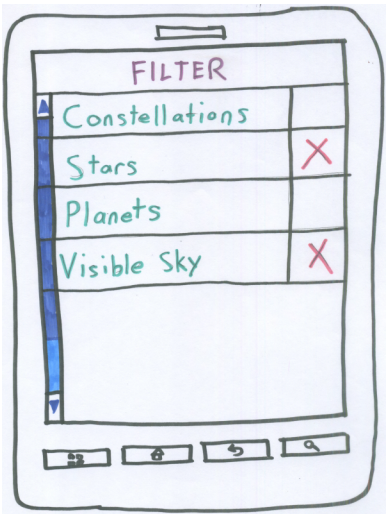

Figure 4.3: UI Mock of the Preferences Menu in StarMapper

4.2 Development Challenges and Testing

During the project, both hardware and software challenges needed to be overcome for the application to be successful. Many challenges were both identified and resolved through thorough testing of the application.

4.2.1 Hardware Challenges

One of the primary issues with the hardware sensors was the method by which the data was delivered to the application. Continuous readouts of raw data were constantly delivered to and utilized by the application, and sporadic outlier readings resulted in a jittery, unreadable screen during initial hardware sensor testing. Through online research I discovered a low-pass filter algorithm by Nichols [12] that, when implemented on the sensor readings, alleviated the jitter and smoothed the screen.

4.2.2 Software Challenges

The Eclipse IDE Android Virtual Device (AVD) is an Android device emulator that can be used for testing an application during development. However, the latest version of the AVD was very slow and nearly impossible to just run StarMapper, let alone debug it. Fortunately, during the project I upgraded my personal smartphone to a Samsung Galaxy S4 running Android OS Ver 4.2 (Jelly Bean) and was able to use that as the debug device. The Galaxy S4 was one of the most powerful smartphones available at the time of its release and could be used very easily to download, debug, and test StarMapper during development.

Coming from a background with no graphics software development experience, OpenGL was a time-consuming and difficult endeavor to undertake. Learning and debugging the API was a major software challenge that consumed a large portion of the project schedule. The OpenGL renderer ended up as the major bottleneck in the project, and the only solution was to commit the time and effort to develop it.

4.2.3 Testing

StarMapper was tested using both the built-in Eclipse IDE AVD emulator and a Samsung Galaxy S4. Initial testing was done with the Eclipse emulator, but I quickly discovered that the emulator was very slow and only useful for very coarse testing, such as checking for application crashes. Furthermore, it was impossible to simulate the hardware sensors in real-time by

using the emulator. Therefore, after the first few weeks, most testing was done by downloading the application apk to the Samsung Galaxy S4.

4.3 Key Learnings

This section describes some of the key learnings obtained during the StarMapper project.

4.3.1 Java Development

Being unfamiliar with the Android platform and the Java programming language in general, the StarMapper project presented a great opportunity to learn more about mobile application software development using JDK and the Android SDK. The knowledge obtained will be greatly beneficial to the goal of gaining a more holistic understanding of mobile technology, given my background in IC design targeted for the mobile market.

4.3.2 Reusability of Android

The Android team developed the Android platform with the idea of reusing its components to save on redundancy and, by extension, time [16]. Almost all Android applications leverage this feature and StarMapper is no different. StarMapper utilizes several providers and APIs built into the Android system, such as the GPS location provider and hardware sensor APIs for the accelerometer and magnetic field sensor. This availability application developers to focus on the unique aspects of their applications, and not waste

time rehashing existing code.

4.3.3 OpenGL

Learning and utilizing the OpenGL API correctly was the most difficult aspect of the the StarMapper project. OpenGL has many components that must be developed or utilized correctly for proper functionality, such as the matrix calculations, shader program development, and correct API calls. The project timeline in Table 4.1 supports this assertion, with the OpenGL component of the project taking 2.5 weeks, longer than any other component. Adding to the complexity of using OpenGL was the intention of using a gnomonic projection in the projection matrix. In any future projects using OpenGL, sufficient time should be allotted for the development and testing of the OpenGL component.

4.3.4 Other Learnings

For others who may be embarking on an Android application project, there are a few things that would be beneficial to know beforehand:

- Ensure you have a solid understanding of Java before focusing on the Android aspect of the project, because Java is the foundation of Android application development.
- Utilize resources such as `developer.android.com` for information and BKMs (best-known methods) about Android.

- Android is built on the premise of reusability. Make sure to take advantage by leveraging code and Android components that may already exist in other applications
- Have an Android device, or, if available, several Android devices available for testing during development. The device used for testing StarMapper was a Samsung Galaxy S4.
- Plan out the entire project beforehand as best as possible with milestones along the way to make the project more manageable.

There are also a few things to avoid during project development:

- If at all possible, avoid using the Eclipse IDE Android Virtual Device (AVD) for testing. It is very slow. A better option is using an actual Android device for testing.
- Avoid 'reinventing the wheel'. Many Android components that are needed in your project may already exist in other applications. Leverage them instead of rewriting a component.
- Do not wait until code development is finished before testing the application. Because of the encapsulation properties of Java and Android, it is easy to test components in isolation. While the StarMapper project did not develop them, Java Unit Tests can be developed for this very purpose.

- Do not prioritize design aesthetics before functionality. Exorbitant amounts of time can be spent trying to make an application visually appealing or intuitive or 'cool', but if it doesn't work, nothing else matters.

- Do not make the application require a newer version of Android than what is minimally required to function E.g. do not require Android SDK ver. 22 if your application only requires Android SDK ver. 17. Doing so will unnecessarily shrink the total available market for your application.

4.4 Future Enhancements

While StarMapper is fully functional, there are some enhancements that could improve the overall experience for the user.

4.4.1 Moon Phase Display

Currently, StarMapper will always display the full moon in its position in the celestial sky (even for a New Moon). This could be enhanced to display the current phase of the moon, rendering a more accurate depiction of what is actually in the sky. This enhancement would require additional textures of the moon for each phase to be rendered, and the addition of some form of moon calendar that can be compared against the current time to determine the moon phase.

4.4.2 Additional Celestial Objects

Currently, StarMapper is capable of displaying all 88 modern constellations, major stars of the constellations, the planets, the sun, and the moon (in addition to random background stars, the celestial grid, and labels for the celestial objects). Future enhancements could include additional celestial objects, such as the International Space Station, comets, or distant well-known galaxies.

4.4.3 Touch Features in Auto Mode

Most of the touch features in StarMapper are only enabled in manual mode. However, auto mode could make use of additional touch features, such as the ability to zoom in and out while still pointing the device around the sky.

4.4.4 Improved Background Stars

The background star manager currently randomly generates all stars that are not part of a constellation in StarMapper. However, the background star manager can generate actual star locations by using data from the Yale Bright Star catalog to present a more accurate experience for the user. The best implementation would leverage code used for generating the constellations to generate the background stars.

4.4.5 Notifications

Notifications can be added to StarMapper that would inform the user about special celestial events that may be occurring at the moment, such as the 'Dance of the Planets' occurrence mentioned in User Story 2, a lunar eclipse, or meteor showers that may be occurring at the moment. Knowing about these rare occasions would enhance the experience for the user when they may otherwise miss them.

4.5 Final Thoughts

Stargazing is an interesting and thought-provoking hobby that was the inspiration for the StarMapper application. The two main goals of this project were to provide an enjoyable and informative user experience for those Android users interested in astronomy, and to also develop my own knowledge foundation of both the Android platform and mobile application development in general. While there is still room for enhancement in the application, I believe this project accomplishes both goals. StarMapper will hopefully both entertain and educate its users, and has certainly increased my knowledge of Android application development.

Bibliography

[1] Marco Alamia. Article - World, View, and Projection Transformation Matrices. http://www.codinglabs.net/article_world_view_projection_matrix.aspx.

[2] Johann Bayer. Star Bayer Designation. http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Bayer_designation.html.

[3] Kevin Brothaler. *OpenGL ES 2 for Android: A Quick-Start Guide*. Pragmatic Bookshelf, 2013.

[4] U.S. Naval Observatory Astronomical Applications Department. Approximate Solar Coordinates. http://aa.usno.navy.mil/faq/docs/SunApprox.php.

[5] Learn OpenGL ES. Understanding OpenGL's Matrices. http://www.learnopengles.com/understanding-opengls-matrices.

[6] The Apache Software Foundation. Apache Harmony. http://harmony.apache.org.

[7] Jeff Friesen. *Learn Java for Android Development*. Apress, 2nd edition, 2010.

[8] Sergei Goorov. Android Javadocs. http://www.androidjavadoc.com.

[9] Sayed Hashimi, Satya Komatineni, and Dave MacLean. *Pro Android 2*. Apress, 2010.

[10] Ellen Dorrit Hoffleit. Yale Bright Star Catalogue. `http://tdc-www.harvard.edu/catalogs/bsc5.html`.

[11] Zigurd Mednieks, Laird Dornin, G. Blake Meike, and Masumi Nakamura. *Programming Android*. O'Reilly, 2012.

[12] Thom Nichols. Smoothing Sensor Data with a Low-Pass Filter. `http://blog.thomnichols.org`.

[13] OpenHandsetAlliance. Open Handset Alliance Android Overview. `http://www.openhandsetalliance.com/android_overview.html`.

[14] The Android Open Source Project. Dalvik Libraries. `http://milk.com/kodebase/dalvik-docs-mirror/docs/libraries.html`.

[15] Paul Schlyter. How to Compute Planetary Positions. `http://www.stjarnhimlen.se/comp/ppcomp.html#5b`.

[16] Android Development Team. Androidology - Part 1 of 3 - Architecture Overview. `http://www.youtube.com/watch?v=QBGfUs9mQYY`.

[17] Android Development Team. Application Fundamentals. `http://developer.android.com/guide/components/fundamentals.html`.

[18] Techotopia. The Anatomy of an Android Application. `http://www.techotopia.com/index.php/The_Anatomy_of_an_Android_Application`.

[19] Techotopia. Overview of Android Architecture. http://www.techotopia.com/index.php/An_Overview_of_the_Android_Architecture.

[20] TutorialsPoint. What is Android? http://www.tutorialspoint.com/android/android_overview.htm.

[21] T. C. van Flanden and K. F. Pulkkinen. Low Precision Formulae for Planetary Positions. *The Astrophysical Journal Supplement Series*, 41:391–411, 1979.

[22] Mike Wall. 3 Planets Performing Rare Night Sky Show: How to See It. http://www.space.com/21233-jupiter-venus-mercury-rare-planets.html.

[23] Wikibooks. OpenGL ES Overview. http://en.wikibooks.org/wiki/OpenGL_Programming/OpenGL_ES_Overview.

[24] Wikipedia. Android (Operating System). http://en.wikipedia.org/wiki/Android_(operating_system).

[25] Wikipedia. Gnomonic Projection. http://en.wikipedia.org/wiki/Gnomonic_projection.