# iBetelgeuse: an Augmented Reality Browser for iPhone

Dennis Stevense (1358448)
Peter van der Tak (1358464)

Bachelor's Project (IN3405)
Bachelor of Computer Science
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Finalist IT Group

July 5, 2010

## Attachments

1. Opdrachtomschrijving (in Dutch)
2. Plan van aanpak (in Dutch)
3. Research Report

## Presentation

**date** July 12, 2010—14:00

**location** Finalist Rotterdam
Groothandelsgebouw A4.191
Stationsplein 45
Rotterdam

## Committee

- Kees Pronk, `c.pronk@tudelft.nl` (Delft University of Technology)
- Gerd Gross, `h.g.gross@tudelft.nl` (Delft University of Technology)
- Okke van 't Verlaat, `okke@finalist.com` (Finalist IT Group)
- Jacques Bouman, `jacques@finalist.com` (Finalist IT Group)

**Abstract**

Augmented reality is the concept of taking a representation of the real world, adding something to it and displaying it to the user. Applications for smartphones—so called augmented reality browsers—typically take the image of the integrated camera as a representation of the real world, and project textual or graphical objects on top of this image to create the augmented reality. The user can now look 'through' their phone as if taking a picture to look at the augmented world.

This report describes iBetelgeuse; a novel augmented reality browser for iPhone, designed to be compatible with the Gamaray browser for Android. iBetelgeuse was developed for Finalist IT Group to improve the availability of their current and future augmented reality applications, and is the result of our Bachelor's project. iBetelgeuse supports most features other browsers also support, but focuses on interaction and responsiveness. Like Gamaray, it supports interaction with objects in the virtual world by tapping them, and allows dynamic worlds to be displayed by means of a web service. Unlike Gamaray and most other browsers it is highly responsive to changes in device orientation, which leads to a better user experience.

The project was performed using Scrum; a framework for agile software development. Scrum successfully guided us through the development process, which resulted in an ever working product as we progressed. The first few weeks were dedicated to orientation and research, which proved to be very useful.

iBetelgeuse has a clean architecture and thoroughly tested implementation, fit for maintenance and extension. It employs various algorithms that are quite complex and that require some mathematical background and thorough reading of this document to be understood, but apart from those algorithms the implementation is generally easy to understand and well documented.

Although iBetelgeuse is a fully functional product—currently being reviewed by Apple for publishing in the iPhone App Store—there are still enough areas that can be improved. Our most important recommendations revolve around changes affecting the user experience, although some aspects of the implementation can also be improved, and new features can be added.

Our recommendations for the application of iBetelgeuse, Gamaray, and their web services are, in short, that the interface between the web service and browser should be extended to give the web service more control over what is displayed on the client. We suggest to use a KML based format to replace the currently used GDDF format.

# Preface

Before you lies the report of our final project of the Bachelor of Computer Science program of Delft University of Technology. We have carried out our project by way of a ten-week internship at Finalist IT Group in Rotterdam, which started April 26, 2010. Finalist had the wish to bring an existing open source augmented reality browser (called Gamaray) for the Google Android platform to the Apple iPhone platform. We have been very happy to make that wish come true. The end result is called 'iBetelgeuse'.

This report can be read in various ways. Most chapters can be read individually, depending on the interest of the reader. However, it is recommended to read at least the introduction on page 4 and the chapter about the final product on page 6 to get an idea of what iBetelgeuse is about. Terminology used in this report can be found in the glossary on page 61.

Before starting, however, we would like to express our gratitude to the following parties:

- Gerd Gross and Bernard Sodoyer of Delft University of Technology, for their feedback and support;
- Okke van 't Verlaat and Jacques Bouman of Finalist IT Group, for giving us the opportunity to work on this project, for their feedback and for their continuous trust in us; and
- everybody at Finalist, for the pleasant environment in which we have been able to do our Bachelor's project.

Rotterdam, July 5, 2010

Dennis Stevense
Peter van der Tak

# Contents

# Chapter 1

# Introduction

## 1.1 Augmented reality

In augmented reality, the real world is augmented with artificial data by means of an augmented reality device. In other words, such a device combines actual reality and virtual reality. We will be focusing on augmented reality browsers: software that implements augmented reality on advanced mobile phones.

Such applications typically use the camera to capture a real-time image of the real world and use GPS, a compass and an accelerometer to determine the location and orientation of the device. Based on this knowledge of the spatial state of the device, virtual objects are then projected onto the camera image. These browsers most often communicate with a server to fetch points of interest to display as virtual objects.

With an augmented reality browser, users can browse the augmented reality by holding their mobile phone right in front of them (as if taking a picture) and looking around. Augmented reality can be used for both practical and entertaining purposes. Existing applications range from finding nearby ATMs [4] to projecting alien space ships in the sky [8].

## 1.2 Finalist IT Group

Finalist IT Group[1] is an IT services firm whose headquarters are located in Rotterdam and that specializes in open source and open standards. For Kennisnet[2], a foundation that supports schools in the Netherlands regarding ICT, they did a pilot project called ARena that makes use of augmented reality. In a Google Maps environment, teachers can define a tour through a city by placing content and questions in the virtual world. Students can then use an augmented reality browser to do the tour and answer the questions.

For this project, Finalist forked Gamaray[3], an open source augmented reality browser for the Google Android[4] platform. This fork is called Betelgeuse and is currently almost identical to Gamaray.

Finalist has the desire to use augmented reality for more applications in the near future. Therefore they want to support other mobile platforms as well, of which the Apple iPhone[5] platform is most important.

[1] http://www.finalist.nl/
[2] http://about.kennisnet.nl/
[3] http://www.gamaray.com/
[4] http://www.android.com/
[5] http://www.apple.com/iphone/

## 1.3 iBetelgeuse

The purpose of this project is to develop an augmented reality browser for the iPhone platform. This browser must be compatible with the web service specification used by Gamaray, so that the new browser can be used with the existing ARena web service. The browser will be released as open source. Therefore, it could well become a competitor to existing commercial augmented reality browsers that are more closed in nature, such as Layar[6].

The name that was chosen for this new augmented reality browser is iBetelgeuse. This is a concatenation of iPhone and Betelgeuse. (The name Betelgeuse was chosen by Finalist, because Betelgeuse is star that is going supernova (relatively) soon. And when a star goes supernova, it emits gamma rays.)

## 1.4 Report structure

The purpose of this report is describing the process of developing iBetelgeuse, explaining the choices we have made while implementing it and outlining the results we have obtained.

Chapter 2 describes the finished product. Then, Chapter 3 describes the process of developing the augmented reality browser. Details about the final implementation of the software is given by Chapter 4. Chapter 5 discusses the quality of the product based on a small set of criteria. Next, Chapter 6 lists a few improvements that we think can be made to the software. Finally, Chapters 7 and 8 respectively draw conclusions and do recommendations.

[6] `http://www.layar.com/`

# Chapter 2

# Final product

This chapter describes the features of iBetelgeuse at the end of the project. This version of the application will also be published in the Apple App Store. The current browser supports most features that are also supported by the Gamaray browser for Android and is fully compatible with Finalist's ARena backend.

## 2.1 Features and overlays

The most notable feature of iBetelgeuse is the feature that all augmented reality browsers support: displaying the camera image and drawing images and text on top of it. These images and text items are called *features*. Each feature is drawn on the screen on the projected location in the real world, and is updated as the user rotates the device. This gives the user the idea of the augmented reality we described in our research report [21].

Because it may be hard for the user to find those features by only rotating the device, iBetelgeuse shows a *radar* on the screen. This radar displays features that are located within a specified range as small points, which helps the user to find nearby features, and gives a rough indication of the distance to the feature.

In addition to features, iBetelgeuse supports text and image *overlays*. Overlays, however, do not move when the user moves or rotates the device: their location is defined in screen coordinates, and they will be displayed there as long as they exist.

## 2.2 Dimensions

The collection of features, overlays, and other relevant data is called a *dimension*. Dimensions are loaded from so-called *Gamaray Dimension Description Files* (GDDFs) [3], which are loaded from a URL. It is possible to make dimensions change over time by generating the GDDF dynamically using a custom HTTP web service. The iBetelgeuse browser regularly reloads its dimension, and sends relevant state information (such as its current position) to the server using HTTP POST data in the reload request. The server can process this request and generate a new (altered) dimension, which iBetelgeuse loads and shows to the user.

To allow for user interaction, iBetelgeuse can perform an action when the user taps a feature or an overlay. This either loads a different or updated dimension, or closes the AR browser and opens a specified website. This interaction is unique to Gamaray and iBetelgeuse, and allows users to develop dimensions that can do much more than the virtual worlds implemented by, for example, Layar.

## 2.3 Built-in QR scanner

Another unique feature of iBetelgeuse is that it has a built-in Quick Response (QR) code scanner, which can be used to load dimensions without the need to enter the URL manually. A QR code is a two-dimensional barcode that can contain all kinds of data such as a URL. This feature was added with ARena in mind—because ARena generates QR codes, and using ARena with Android requires the use of an external QR scanner—but is useful for any dimension because scanning a QR code is much easier than typing a complete URL on the keyboard of a mobile phone.

The built-in scanner also allows for QR codes to be placed in the real world, that can conveniently be scanned from within the application to load a new dimension. This enhances the augmented reality.

## 2.4 Good responsiveness

A difference with most other current augmented reality browsers is that iBetelgeuse is very responsive. We use filtering algorithms that are different from similar browsers, leading to quicker responses to orientation changes.

The filter parameters can still be improved to work better outdoors; the current values work best for indoor use because there appears to be less compass noise indoors. For outdoor use, it might be beneficial to trade some responsiveness for better stability by changing the filter parameters, but even then the browser would be much more responsive to orientation changes than, for example, Layar.

## 2.5 App Store product description

Below is the marketing description we have written for iBetelgeuse:

"iBetelgeuse (pronounced: iBeetlejuice) is a free and open augmented reality browser that adds a whole new dimension to your reality. Such a higher dimension can contain all kinds of text and graphics that can be displayed on top of the camera image as you move and look around with your iPhone. A dimension can also be interactive, allowing it to be anything from a simple informational layer to an interactive social game. In addition, iBetelgeuse has a built-in QR-code scanner, making the possibilities endless.

Due to the open nature of iBetelgeuse (even the source code of the app can be found online) you can potentially find dimensions all over the web. They can be made by you and others without restriction using simple tools and published and then used by people all over the world.

One place where you can make simple dimensions is the Betelgeuse Designer:
`http://betelgeusedesigner.appspot.com/`

To learn how to make more advanced dimensions visit the iBetelgeuse page on GitHub:
`http://finalist.github.com/iBetelgeuse/`

Note: iBetelgeuse needs an internet connection, location services, a compass and an accelerometer to work. For the full experience, GPS-accuracy and a camera are a must. Therefore the application will work best on iPhone 3GS and iPhone 4, but also, in a degraded fashion, on iPad."

# Chapter 3

# Development process

This chapter describes the process of developing the augmented reality browser. We will first indicate how we have applied Scrum and agile software development principles in our development process and then, chronologically, list the most important points of what we did over the course of the project.

## 3.1 Scrum and agile software development

Scrum is a framework for agile software development. The most notable aspects of Scrum [7] and agile software development in general [1, 14] that we applied in our development process are:

**Iterative development** As planned [20], we developed the software in five iterations of two weeks, of which three actual development iterations.

**Working software as a measure of progress** Instead of going through the various phases of the waterfall model, we had a working and fairly rounded-off product at the end of each iteration. This made it possible to show the product owner (Finalist) clear results and keep ourselves motivated at the same time.

**Ability to respond to change** We did not make an extensive plan at the start of the project. Instead, we made a product backlog and a sprint backlog at the start of each two-week iteration. See Appendix B on page 45 for more details on our product backlog.

The product backlog initially contained only the most obvious requirements, but was later extended as new insights were gained while working on the software. And yet we were working in an organized fashion, by planning what we needed to implement in the next iteration.

**Test-driven development** Admittedly not always, but most often and especially when fixing a bug, we added unit test cases that verify the functionality before we implemented that functionality. This has proven to be very useful in writing code that is free of the most obvious errors. Also, unit testing in general has helped us to verify functionality that is impossible to verify manually through the user interface of the software.

**Continuous integration** Using git (for version control), Xcode (for building) and GH-Unit (for automated testing) we were able to asses the quality of the software throughout development and after each item on the backlog was implemented. When one member of our team marked an item as finished, the other verified and either accepted or rejected it. We almost never had to solve problems last-minute at the end of an iteration.

**Continuous attention to good design** Since we only had a global design of the application we could freely refactor our code when necessary without having to throw our design out the window. We did so on many occasions, to ensure our classes continued to have well-defined responsibilities, reduce duplication, etc.

What we have not done, as far as agile software development is concerned, is continuous collaboration with the customer. We think this is not a problem in our case: we never defined who our customer is, because of the general use case of an augmented reality browser. Also, augmented reality is fairly new and still changing. Therefore potential users do not have clear expectations or mental models of such software.

## 3.2 Sprint zero

We started 'sprint zero' on Monday, April 26, which was the first day of our internship. This initial sprint was dedicated to preparing for development of the augmented reality browser. This sprint comprised the following tasks:

**Writing the action plan** The action plan [20] describes the background and motivation of the project, the goal of the project, and what deliverables were to be produced. (This is also partially written in the project description [19].) As well as this description of the project, it shows how we planned to achieve good results: it describes the process methodology that would used, the tasks that would be performed, the schedule that would be followed, the facilities that could be used, and the way quality would be monitored and achieved.

**Doing research and writing the research report** After writing the initial version of the action plan, we started researching the problem domain [21]. Since we were both familiar with the concepts of augmented reality, but not much more than that, we compared the most important existing augmented reality browsers and their features and flaws. Also, we retrieved a copy of the Gamaray source code and analyzed it. This showed that Gamaray is not very well designed and documented, and that it is therefore undesirable to be inspired by it too much. The analysis of Gamaray lead to the conclusion that we had to study some theory on geographical algorithms, determining the current orientation of the device, applying signal filtering techniques to the sensor data, and projecting the augmented reality correctly so that it matches the camera image.

**Making a proof-of-concept** To get an idea of the feasibility of the theory we found, we wrote a simple proof-of-concept application. This application worked satisfactory. See Appendix A on page 43 for a detailed description of our proof-of-concept.

**Compiling the product backlog and sprint backlog** Finally, we determined the requirements of the software in the form of a product backlog. Then, we prioritized the items in this backlog and scheduled a subset of these requirements in the sprint backlog of the next sprint. See Appendix B on page 45 for a detailed description of our product backlog.

We also set up Pivotal Tracker[1] to keep track of our backlogs. It functions as a fundamental part of the project management aspect of Scrum: the storyboard. The storyboard shows the status of backlog items: not yet started, scheduled, in progress, finished, accepted, etc.

**Making a global design** When we felt we had a good understanding of the problem domain, we started designing our application. To display and discuss our design, we constructed a class diagram. The class diagram is shown and explained in Appendix C.1 on page 49.

**Setting up the development environment** We will be developing in Xcode, the environment for iPhone software development. We created a new Xcode project and put it on a public GitHub repository[2]. We added the GH-Unit framework[3] to our Xcode project for unit testing. To measure test coverage, we set up gcov[4] and lcov[5]. While the coverage reported by gcov is not perfect for Objective-C code, it gives a sufficient indication of untested methods and branches.

[1] http://www.pivotaltracker.com/
[2] http://github.com/finalist/iBetelgeuse
[3] http://github.com/gabriel/gh-unit
[4] http://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[5] http://ltp.sourceforge.net/coverage/lcov.php

## 3.3 Sprint 1

We started sprint 1 on Friday, May 14. This is the first development sprint. Since all functionality had been scheduled in the sprint backlog, we had a nice overview of what to do.

### 3.3.1 Progress

After the first day, the camera image was displayed correctly on the screen, and the first item on the list was marked “done”. Although we had a pretty tight schedule, we managed to complete all stories of the sprint backlog three days before the sprint ended. In the remaining time, we managed to implement three features that were planned for the next sprint. At the end of sprint 1, our augmented reality browser was working and practically compatible with ARena.

The features that took the most time to implement were parsing XML data into our data model, determining the orientation of the device and rendering the radar. This roughly corresponded to the effort rating we assigned to these user stories in Pivotal Tracker.

### 3.3.2 Problems

There were only a few unforeseen problems in this sprint:

- We noticed that the Gamaray browser does not scale down features that are farther away, while the perspective transformation we used in our browser scaled features proportionally to distance. Since we wanted to be compatible with Gamaray and since features have no defined real-world size in the Gamaray Dimension Description File format, we decided to scale up features proportionally to distance, canceling out the scaling due to our perspective transformation. This makes it possible to disable this inverse scaling in the future, when and if features do have a defined size.
- Another problem we ran into, which was only partially foreseen, was how to plot the radar on the screen. It was not only hard to implement, but it also was not trivial to consistently define what the user would want the radar to display in every orientation. It is particularly tricky to make sure that objects on the radar appear to stay in the same location if the device is rotated in its own plane when the user is holding it up in front of themselves.

  Eventually, we decided that as a user, you want objects that are in front of you to display towards the top of the radar. Furthermore, it is most likely that the screen of the device directly faces the user. These ideas mostly define the behavior of the radar.

  What remains is defining what should happen when the device is held upside-down over a user’s head, as if they are looking at the sky. The prior definition would cause blips on the radar to jump around at the point where the device is exactly parallel to the ground. Since this is undesirable, and it is impossible to give a meaningful definition of what an upside-down radar should display, we chose to simply clear the radar in this case.

### 3.3.3 Evaluation

Near the end of the sprint, we field-tested our product. We were pleased with the result: everything was working as expected, although the location readings given by the operating system were sometimes quite a bit off from our actual location. We were clearly able to see this in the Maps application that comes with the iPhone. It seemed to be related to our device, since we did not experience the same behavior with another iPhone.

Another issue we noticed while testing was that the accelerometer readings are greatly distorted when walking. The movement likely leads to a lot of variation in the accelerometer readings. It appeared

that we would have to implement pretty aggressive filtering to overcome this issue. It reaffirmed our prediction that sensor signal filtering would be one of the major problems in this project.

## 3.4 Sprint 2

This second development sprint ran from May 31 up to and including June 11. At the start of the sprint we already had various user stories scheduled in Pivotal Tracker for this iteration. During this sprint it was decided that the product would be submitted to the App Store somewhere during sprint 3, so this caused a few tasks to be added or have their priorities changed. In particular, we had to make the user interface more aesthetically pleasing and make sure the application properly adheres to the iPhone conventions (such as support for both portrait and landscape screen orientations). Some less urgent tasks—such as accelerometer calibration—were moved back.

### 3.4.1 Progress

This sprint primarily consisted of improving the functionality implemented in the last sprint in various ways. Either by making the functionality more accurate, or more attractive to the user.

Since virtual objects were not still at all in relation to the camera image, a big chunk of this sprint—in terms of time—was dedicated to filtering the measurements made with the accelerometer and the magnetometer. We had already expected this to be a difficult problem to solve, so this 'set-back' did not interfere with our schedule.

One reasonably big piece of functionality that was added in this sprint is the ability to scan a QR code. Luckily we were able to find a third-party GPL-licensed library called ZBar [10] that is able to scan QR codes in images. This library even had an Objective-C wrapper we could use. Due to this library, adding this functionality went down much easier than expected.

### 3.4.2 Problems

The most obvious problems in this sprint were the following:

- The measurements provided by the accelerometer and the magnetometer are very noisy, making objects jump around on the screen. In addition, the magnetometer is affected by magnetic fields other than the magnetic field of the Earth, making the measurements quite inaccurate. The magnetometer is only able to calibrate itself automatically to magnetic fields that move with the device.

  We first tried to improve readings using a simple exponential low-pass filter, but this did not improve the situation much and it did so at the expense of responsiveness. Next, we started making measurements of both sensors under various conditions, such as when keeping the device still on a table, when rotating it, and when walking with it. Then we plotted these measurements in MATLAB and tried to improve them using various techniques ranging from a simple moving average or median filter to a double exponential filter. Still unsatisfied with the augmented reality experience these filters resulted in, we started researching the Kalman-filter. See Appendix C.3 on page 55 for a full account of this research.

  In this sprint we settled on a reasonably simple filter we created ourselves that approximates the derivative of the measurements and filters that instead of the direct measurements. This smoothes the data sufficiently while still reacting to sudden movement. As can be read later on in this report, this is not the filter that was eventually used in the final version of the software.

- A related problem we had was that, by using the raw magnetometer readings, we were using magnetic North for determining the orientation of the device. While this is accurate enough in

the Netherlands, it is not in other parts of the world (in California, United States for example). Since calculating magnetic declination ourselves proved to be too hard or require many third-party libraries, we instead opted to simply use the difference between the magnetic North heading and true North heading provided by the iPhone as the declination. (Also see our research report [21] for information about magnetic declination.)

## 3.5 Sprint 3

The third development sprint started June 14th. This sprint was a bit more chaotic than the others, because we were still dissatisfied with the accelerometer en magnetometer filtering. Yet, there were about 20 points worth of tasks that were scheduled in Pivotal Tracker that had to be done. There were also some 'chores' scheduled that comprised optimizing code and fixing memory leaks.

### 3.5.1 Progress

During this sprint we settled on a satisfactory filter for the accelerometer and magnetometer. It is roughly based on the derivative-based filter that we had implemented before, but instead uses quaternions to represent orientations. See Appendices C.3 and C.5 for our research on filtering and quaternions. See Section 4.4.3 on page 28 for information about the current filter implementation.

Besides filtering, there was also work done to make the application more attractive and provide better feedback to the user about the application state. For example, the radar rendering was improved and a beep tone was added that plays when a QR code is scanned successfully.

This sprint also took care of making the application ready for submission to the App Store. The application was given an icon and other graphics that are needed for an iPhone-application. (The icon was designed by Finalist.) Also, it was tested and made compatible with iOS (formerly iPhone OS) 4.0 and made ready for the higher resolution screen of the iPhone 4. Furthermore, we spent some time documenting[6] the web service that iBetelgeuse supports, for people that want to make dimensions for it.

Finally, an important activity in this sprint was finding and fixing memory leaks, and optimizing and documenting code. Apple supplies a profiling tool called Instruments with its development environment, which was very useful in tracking down slow code, measuring screen refresh rates and finding memory management issues. For example, we experienced very low frame rates when there were a lot of features on-screen at once. It turned out these were being drawn to off-screen buffers before being blitted to the screen. Telling the system that these views did not need to be clipped improved frame rates considerably.

### 3.5.2 Evaluation

We did another field-test of our product at the end of this sprint. There were three issues we found.

One issue was regarding the wording of an (expected) error message we got when entering the tunnel underneath Rotterdam Central Station. This issue was resolved afterwards by improving the wording of the message.

The second issue was that we still noticed some erratic behavior of on-screen items when walking. However, this is a trade-off between responsiveness and smoothness of our sensor filtering, and we were quite satisfied with the behavior when looking around standing still.

The third issue was that we concluded (again) that the assisted GPS of the iPhone is not accurate enough for nearby real-world objects and their on-screen equivalents to line up. Unfortunately, this wholly depends on the location services provided by the device.

[6] `http://finalist.github.com/iBetelgeuse/`

All in all we were very satisfied. We think that we have been able to match or even surpass the quality of the behavior of leading augmented reality browsers such as Layar. See our research report [21] for a discussion about Layar.

## 3.6 Final sprint

The final sprint started on June 28th. Since we finished the augmented reality browser right as planned, this sprint is dedicated to writing this report and making a presentation on our project. At the time of writing, this sprint was still in progress.

In addition, during this sprint we submitted iBetelgeuse to Apple's review process for placement in their App Store and finished the product page[7] describing how to get content into the browser. At the time this report was finished, we had not received feedback from Apple yet.

[7] `http://finalist.github.com/iBetelgeuse`

# Chapter 4

# Implementation details

This chapter describes various aspects of the final implementation of iBetelgeuse. It first describes the general architecture of the program, followed by descriptions of our solutions to the most important challenges as they are currently implemented. Some of these solutions have changed over time; the intermediate solutions can be found in Appendix C.

## 4.1 Architecture

This section discusses the structure of our implementation. The application has a model-view-controller architecture. See Figure 4.1 on the following page for an overview.

This section assumes some knowledge of the Objective-C programming language, although this is not absolutely necessary for understanding the text below. Due to the lack of namespacing in Objective-C, it is common to prefix class names. We have used the prefix `AR` (for Augmented Reality).

Note that the class diagrams in this section are only there to illustrate the structure of the application; for brevity, the diagrams are incomplete and should therefore not be used as a reference. Doxygen can be used to generate a reference of all classes.

### 4.1.1 Model

In the model-part of the application, we can distinguish between classes that are related to modeling a dimension, classes that represent filters and classes that are responsible for managing some specific task. In addition, there are some sets of C-based interfaces for doing various calculations. This is illustrated in Figure 4.1 on the next page. The text below discusses each of these domains.

#### Dimension

The classes in this domain model a dimension, i.e. the virtual world and everything it contains. The classes directly corresponds to the elements of the Gamaray Dimension Description File [3], so their meaning will not be reiterated here. See Figure 4.2 on the following page for a diagram of these classes.

All of these classes respond to a message named `startParsingWithXMLParser:element:attributes:notifyTarget:selector:userInfo:`. This class method allows users of the class to create an instance by parsing it from an XML document. See Section 4.2 on page 21 about resource loading for more information about how this works.

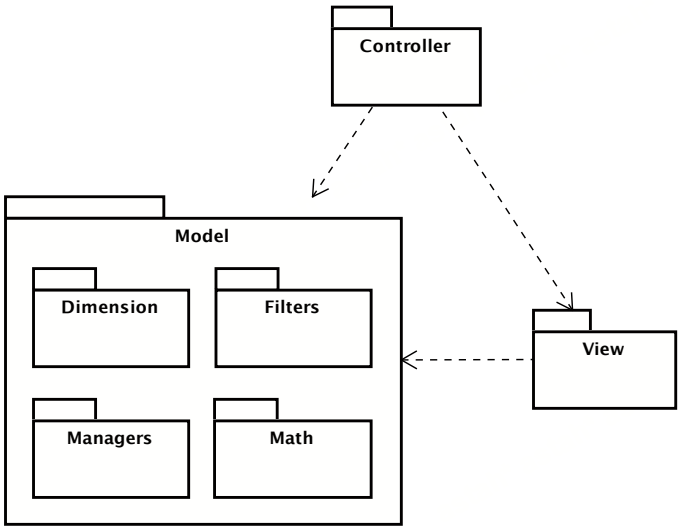



Figure 4.1: Overview of application structure.

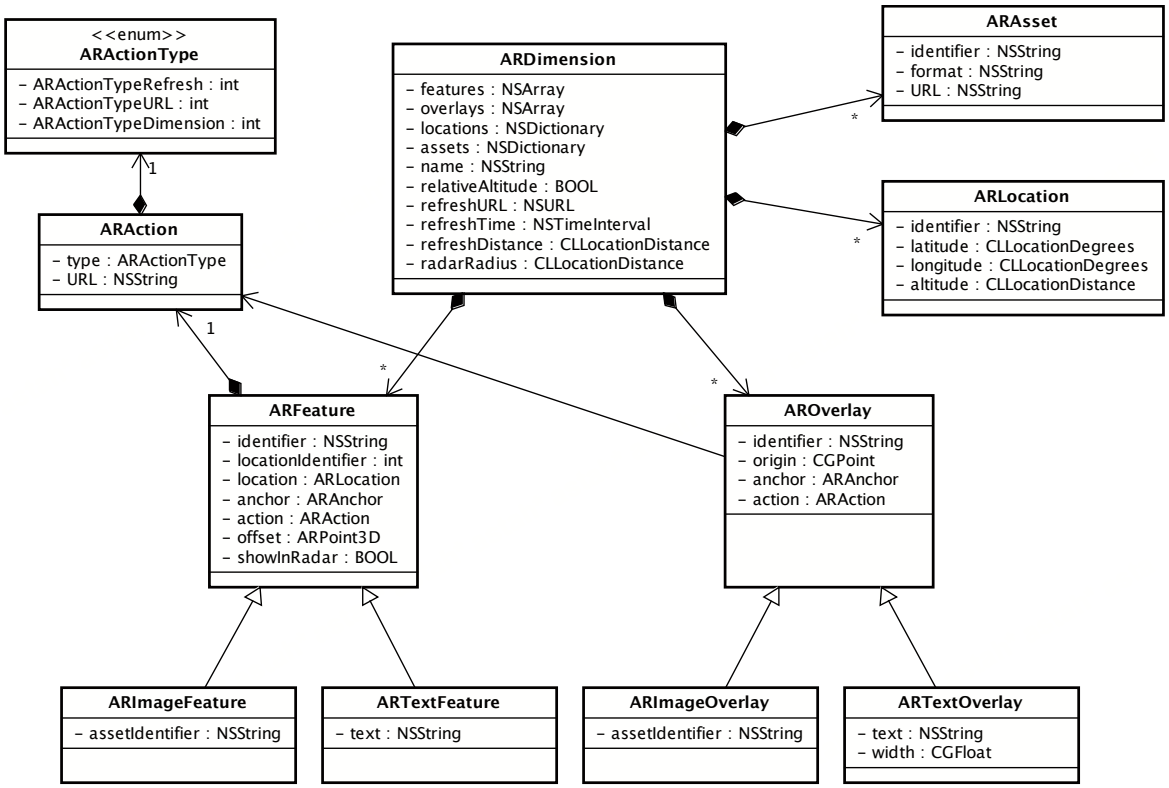



Figure 4.2: Class diagram of dimension model.

### Filters

The classes in this domain take care of filtering the accelerometer and magnetometer values. See Figure 4.3 on the following page for a diagram of these classes. The hierarchy of these classes may seem a bit complicated. We will explain the classes by the type of value they filter:

**singular values** The abstract `ARFilter` class takes and returns a singular value, hence the following Objective-C method signature:

```
- (ARFilterValue)filterWithInput:(ARFilterValue)input
                       timestamp:(NSTimeInterval)timestamp;
```

The timestamp arguments may be used by specialized filters that, for example, need to approximate the derivative of a signal.

The only concrete subclass `ARDelayFilter` simply delays the input by a certain amount of samples.

**arrays of values** The `ARArrayFilter` class takes and returns an array of values of a certain size:

```
- (void)filterWithInputArray:(const ARFilterValue *)input
                 outputArray:(ARFilterValue *)output
                   timestamp:(NSTimeInterval)timestamp;
```

This filter takes an instance of a subclass of `ARFilterFactory` to create singular filters for each element in the array. This means the array is filtered element-wise.

**points** The abstract `ARPoint3DFilter` class takes and returns a 3D point:

```
- (ARPoint3D)filterWithInput:(ARPoint3D)input
                   timestamp:(NSTimeInterval)timestamp;
```

The concrete subclass `ARSimplePoint3DFilter` uses an `ARArrayFilter` to filter the coordinates of points element-wise.

The concrete subclass `ARAccelerometerFilter` is used to filter the accelerometer values before they are used to determine the device orientation. It uses an uses a `ARSimplePoint3DFilter` with an `ARDelayFilterFactory`.

**quaternions** The abstract `ARQuaternionFilter` class takes and returns a quaternion:

```
- (ARQuaternion)filterWithInput:(ARQuaternion)input
                     timestamp:(NSTimeInterval)timestamp;
```

The concrete subclass `ARSphericalMovingAverageQuaternionFilter` applies a spherical moving average filter to its input values. See Appendix C.5.2 on page 58 for more details about the spherical moving average of quaternions.

The concrete subclass `ARDerivativeSmoothQuaternionFilter` applies our self-invented derivative-based smoothing to its input values. It uses a `ARSphericalMovingAverageQuaternionFilter` and a `ARWeightedMovingAverageQuaternionFilter` to do its work. See Section 4.4.3 on page 28 for more details about this filtering.

The concrete subclass `AROrientationFilter` is used to filter device orientation. It is actually just a wrapper around `ARDerivativeSmoothQuaternionFilter`, allowing different filtering techniques to be used without changing the class name.

The following classes accept additional parameters and therefore do not fit within the above categories:

**weighted singular values** The `ARWeightedMovingAverageFilter` class is in interface similar to the `ARFilter` class, but takes an input weight instead of a timestamp:

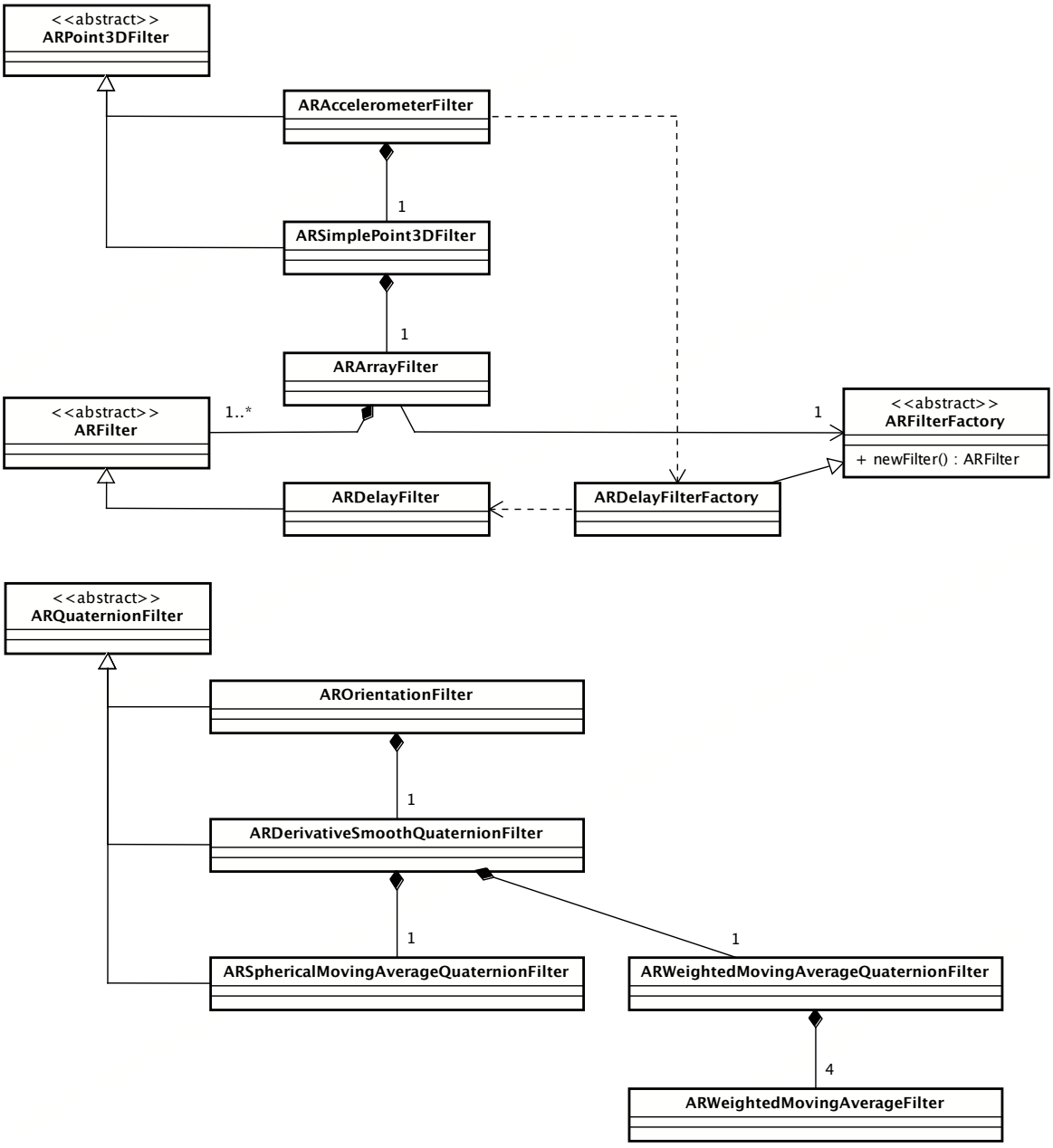



Figure 4.3: Class diagram of filters.

```
- (ARFilterValue)filterWithInput:(ARFilterValue)input
                          weight:(double)weight;
```

This filter calculates the weighted moving average using a certain window size.

**weighted angular velocity** The `ARWeightedMovingAverageQuaternionFilter` class is similar to the `ARWeightedMovingAverageFilter` class, but then takes and returns a quaternion:

```
- (ARQuaternion)filterWithInput:(ARQuaternion)input
                         weight:(double)weight;
```

This filter uses multiple instance of `ARWeightedMovingAverageFilter` to take the element-wise moving average of a quaternion. We use it to filter angular velocities. Note that this filter is not generally correct for quaternions that represent orientations. See Appendix C.5.1 on page 58 for more details about quaternions and angular velocity.

### Managers

The classes in this domain manage some process or activity. There are three such classes:

**ARSpatialStateManager** Continuously determines the spatial state (location and orientation) of the device in relation to Earth. It uses the assisted GPS, magnetometer and accelerometer of the iPhone to do this. See Section 4.4 on page 27 for more details about spatial state determination.

It notifies a delegate object (which implements the `ARSpatialStateManagerDelegate` protocol) at regular intervals to tell it that the device's spatial state has changed.

Snapshots of the device's current spatial state are represented by instances of `ARSpatialState`. See Section 4.3 on page 25 to get more information about the coordinate systems that `ARSpatialState` uses.

**ARDimensionRequest** Performs an asynchronous dimension refresh request, using the protocol defined by GDDF [3]. Users of the class should implement the `ARDimensionRequestDelegate` protocol to be notified when the request has failed or succeeded. See Section 4.2 on page 21 for more information about resource loading.

**ARAssetManager** Manages the asynchronous loading of one or more assets. Assets are instances of `ARAsset`. The result of loading an asset is an `NSData` object, which contains binary data. Users of the class should implement the `ARAssetManagerDelegate` protocol to be notified when an asset has been loaded or failed to load. See Section 4.2 on page 21 for more information about resource loading.

### Math

There are four C-based interfaces to do all kinds of calculations:

**ARWGS84** There is currently one function in this interface: `ARWGS84GetECEF`. It takes WGS84 coordinates (latitude, longitude, altitude) and converts it to ECEF coordinates (x, y, z). See Section 4.3.2 on page 25 for information about ECEF.

**ARPoint3D** This interface defines a struct that represents a 3D point. It is also used as a value type for vectors in $\mathbb{R}^3$. There are many functions for working with these points, including addition, scaling, dot product and cross product.

**ARTransform3D** This interface is an extension to the `CATransform3D` struct of the Core Animation system framework. This structure represents a 4-by-4 homogeneous 3D transformation matrix. Additional functions are made available for creating and manipulating those transformations, as well as multiplying vectors (`ARPoint3D`) with those matrices.

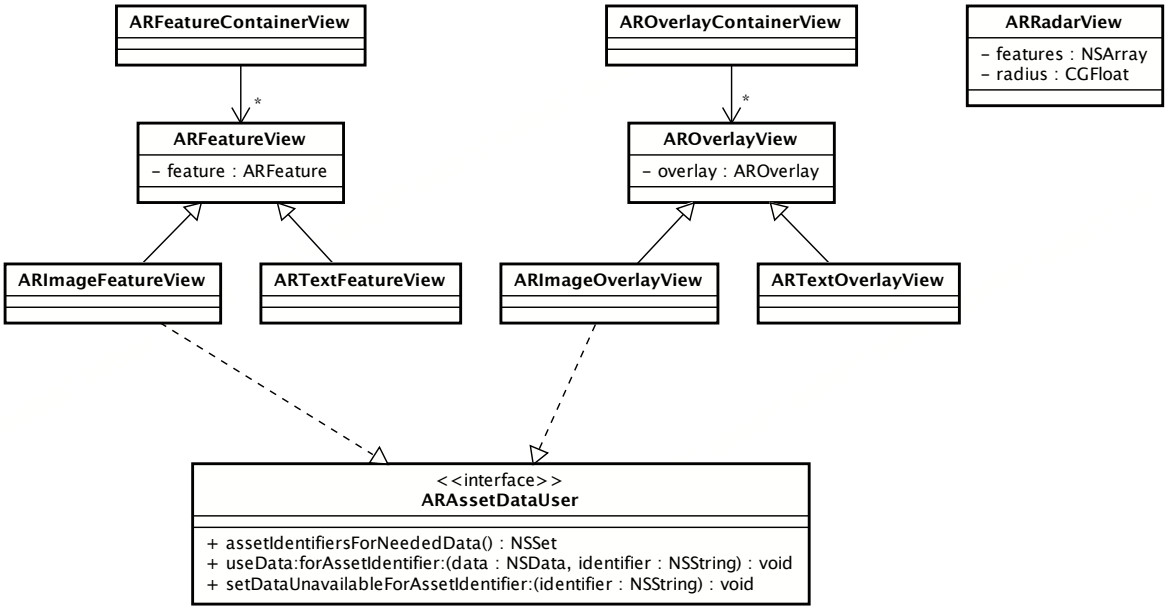



Figure 4.4: Class diagram of view classes.

**ARQuaternion** This interface defines a struct that represents a quaternion and many functions for working with quaternions. See Section C.5 on page 58 for more information about quaternions.

### 4.1.2 View

This set of classes represent all the views of the application. See Figure 4.4 for an overview of these classes.

On top of the camera view (a view class that belongs to the system frameworks), the `ARFeatureContainerView` will be placed. This view merely functions as a container for `ARFeatureView` views and does appropriate hit testing (for user interaction). On top of that, the `AROverlayContainerView` is displayed. This view functions as a container for `AROverlayView` views. Last of all, the `ARRadarView` is displayed on top and renders the radar.

The `ARFeatureView` and `AROverlayView` and their subclasses respectively display features and overlays of a dimension.

The `ARImageFeatureView` and `ARImageOverlayView` both implement the `ARAssetDataUser` protocol that allows them to specify for which asset identifiers they need image data. Some other class (in our case, the `ARMainController`) can subsequently give them the appropriate data for those identifiers.

### 4.1.3 Controller

Last but not least, the controller classes are responsible for setting up the model and views and managing the life cycle of the application. See Figure 4.5 on the next page for the controller classes and their relation to other parts of the application.

**ARApplicationDelegate** Manages the application lifecycle and is responsible for setting up the application after it has been launched. It will create an instance of `ARMainController` to do this.

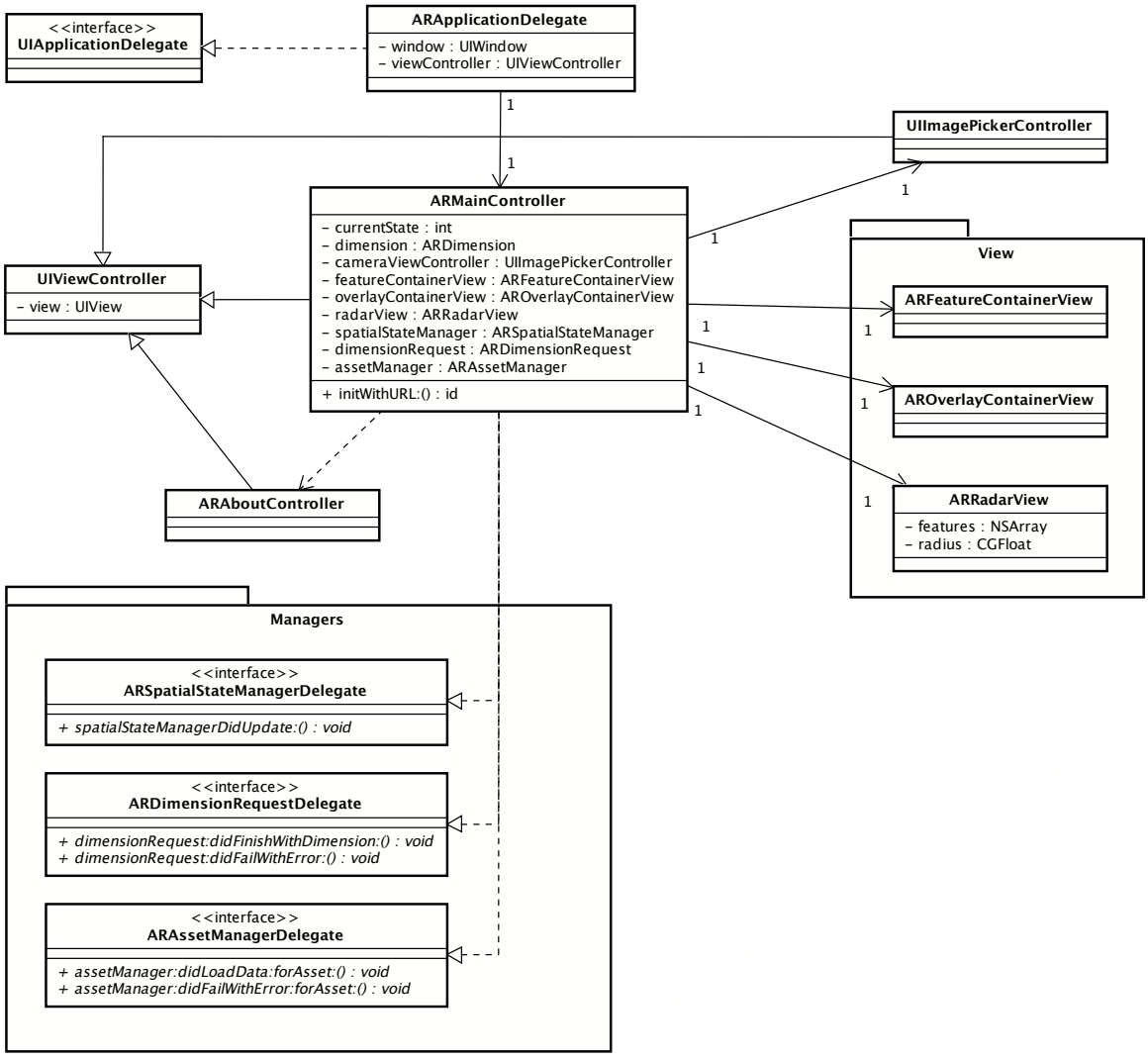



Figure 4.5: Class diagram of controller classes and their relation to other classes.

This class fulfills the role defined by the `UIApplicationDelegate` protocol of the system framework.

**ARMainController** Main view controller of the application that manages the various model and view classes. It controls the augmented reality view of a dimension as well as the scanning of QR codes. This class uses and therefore implements all delegate protocols of the manager classes described in Section 4.1.1 on page 18.

An instance of the `UIImagePickerController` system class is used to display a live image of the camera.

**ARAboutController** View controller that simply displays the localized About.html file in the application bundle. The main controller displays this view controller modally when the user chooses the Info menu item.

## 4.2 Resource loading

This section discusses how resources are loaded. We recognize two types of resources: dimensions, and assets. Dimensions are defined in an XML-based format which needs to be parsed. Assets are usually images.

### 4.2.1 Dimension loading and parsing

Dimensions are loaded asynchronously by the `ARMainController` using a one-shot `ARDimensionRequest`. The `ARDimensionRequest` uses an `NSURLConnection` to fetch the data from the URL and an `NSXMLParser` to parse the data into an instance of `ARDimension`. The delegation pattern common to the Apple frameworks is used to send notifications between classes upon success or failure. See Figure 4.6 on the next page for a sequence diagram of this process.

In Figure 4.7 on page 23 a sequence diagram illustrating the XML parsing process is given. The indices in the explanation below refer to the indices of the method calls in that diagram.

`NSXMLParser` also makes extensive use of the delegation pattern. (Consult the Apple documentation for more details about this class.) When calling the `parse` method of the parser (2), it will start to send messages to its delegate (2.1, 2.2, 2.3) when it has parsed a new piece of XML content. The signature of the method that is called on the delegate when a new opening tag has been found is:

```
- (void)parser:(NSXMLParser *)parser
           didStartElement:(NSString *)elementName
              namespaceURI:(NSString *)namespaceURI
             qualifiedName:(NSString *)qualifiedName
                attributes:(NSDictionary *)attributeDict;
```

All dimension model classes are able to create an instance of themselves by parsing their own part of the XML tree using an existing `NSXMLParser`. This can be done using a class method of the following signature:

```
+ (void)startParsingWithXMLParser:(NSXMLParser *)parser
                          element:(NSString *)element
                       attributes:(NSDictionary *)attributes
                     notifyTarget:(id)target
                         selector:(SEL)selector
                         userInfo:(id)userInfo;
```

This method is often called from the parser callback (2.1) above. What happens when this method is called on, for example, the `ARDimension` class (2.1.1) is:

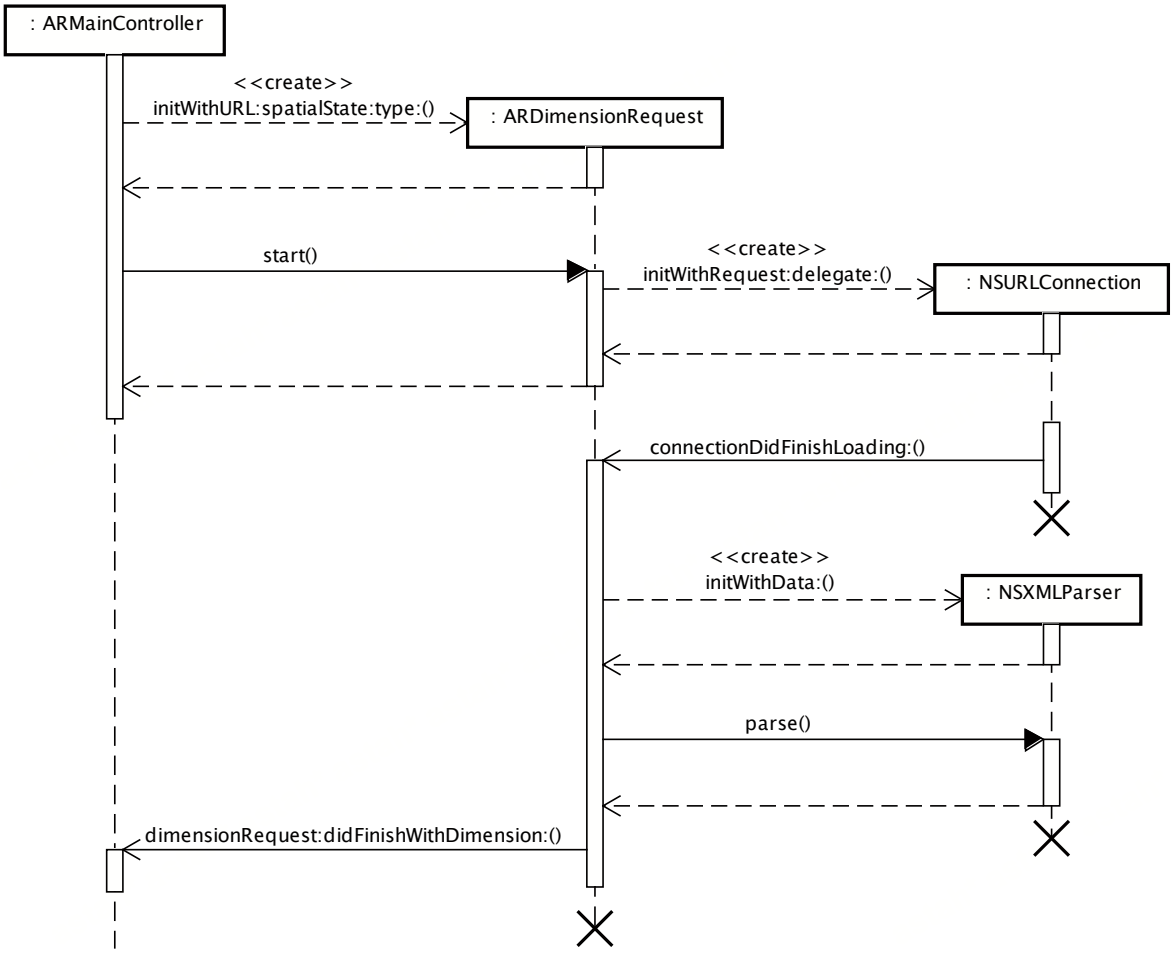



Figure 4.6: Sequence diagram of dimension loading.

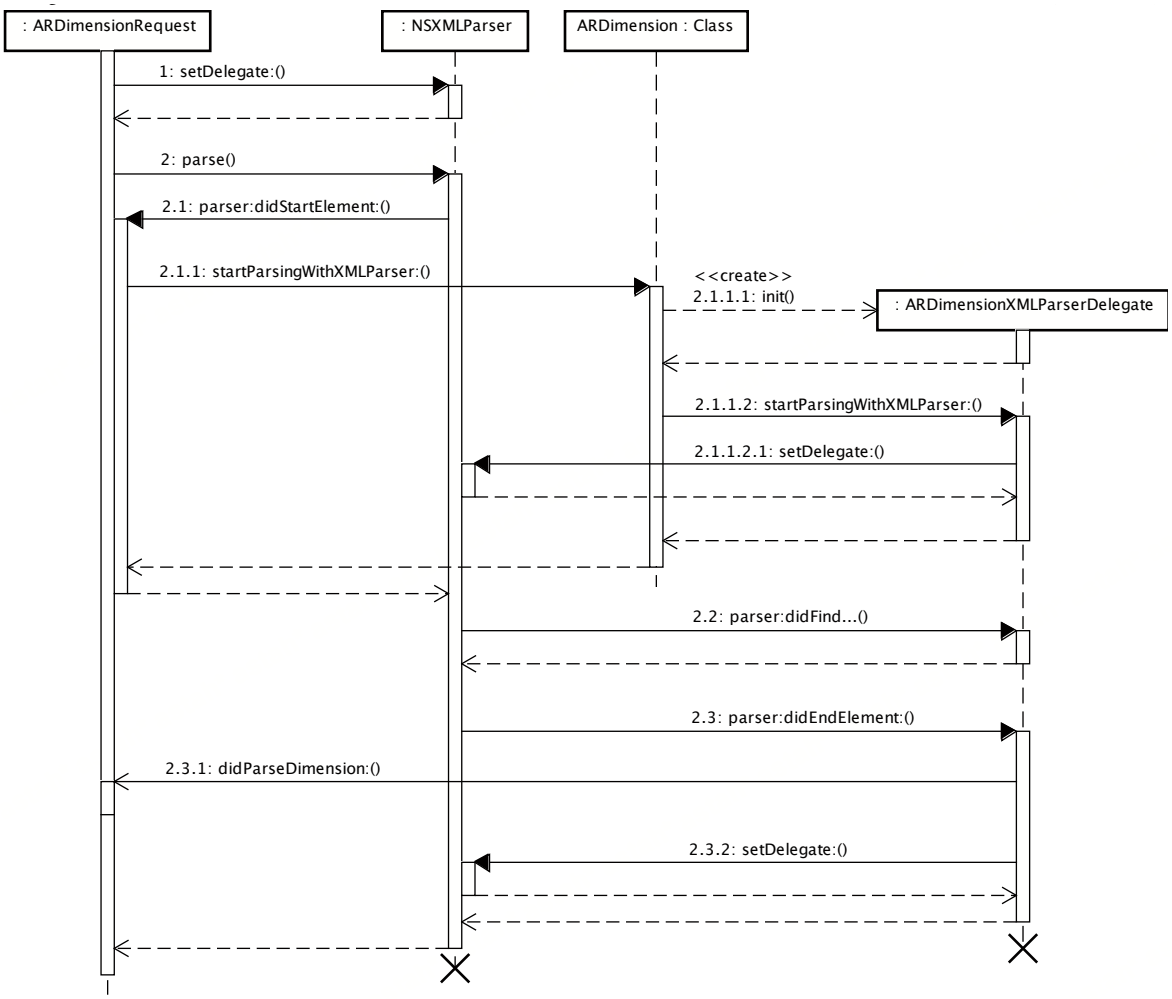



Figure 4.7: Sequence diagram of dimension parsing.

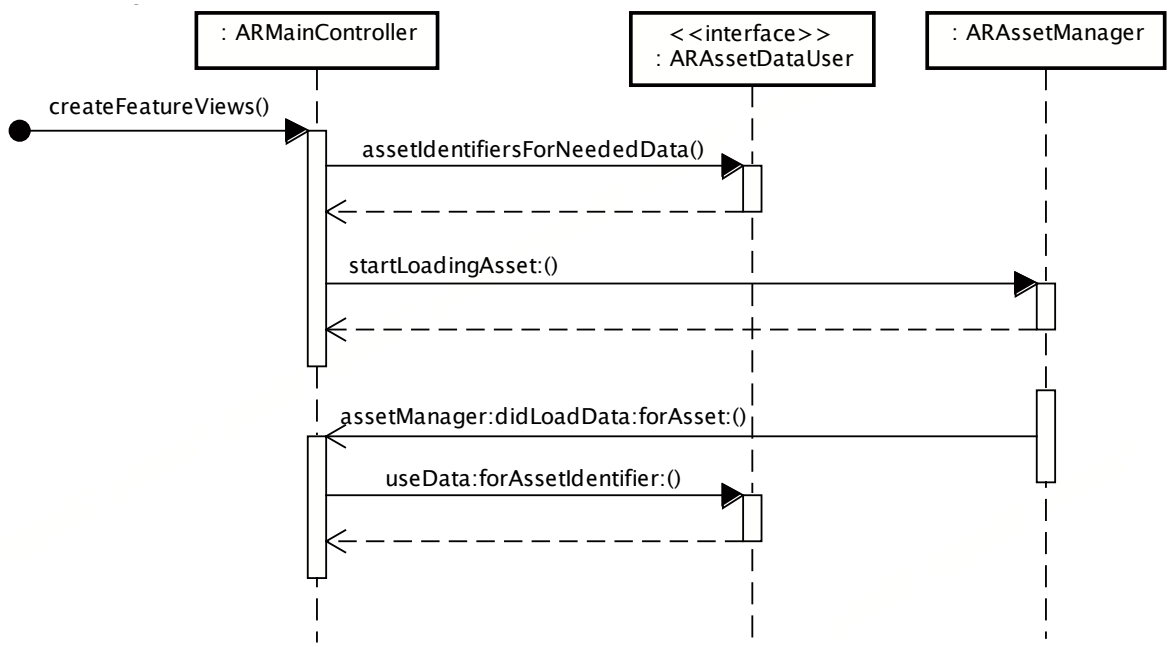



Figure 4.8: Sequence diagram of asset loading.

1. an instance of `ARDimensionXMLParserDelegate` is created (2.1.1.1), and
2. this newly created object saves the current delegate of the XML parser and tells it to be the new delegate of the XML parser (2.1.1.2, 2.1.1.2.1).

Then, after returning from `parser:didStartElement:...`, the `NSXMLParser` instance will send all subsequent parsing callback messages to the instance of `ARDimensionXMLParserDelegate` (2.2, 2.3). When this delegate object is done parsing its part of the XML tree (for example, when finding a matching closing tag, 2.3), it will restore the original XML parser delegate (2.3.2). In addition, it will notify the target (2.3.1) that was passed as an argument to `startParsingWithXMLParser:...`.

This way, the XML parsing effort is distributed among all model classes, and each model class only needs to know about its direct children.

### 4.2.2 Asset loading

All classes that need asset data (in our case, `ARImageFeatureView` and `ARImageOverlayView`) implement the `ARAssetDataUser` protocol. This protocol allows those classes to tell for what asset identifiers they need to have data and allows other classes to provide the data when it becomes available. In our case, `ARMainController` is responsible for asking what data is needed and providing the data.

Assets are loaded asynchronously by the `ARMainController` using an `ARAssetManager`. The `ARAssetManager` uses an `NSOperationQueue` to fetch the data of many assets on dedicated threads. (The operation queue is a system class that creates as many threads as is useful, given available system resources. In practice, on an iPhone, it will probably create a single thread that sequentially executes all operations.) The delegation pattern is used to notify relevant objects of success and failure.

See Figure 4.8 for a sequence diagram that illustrates this process.

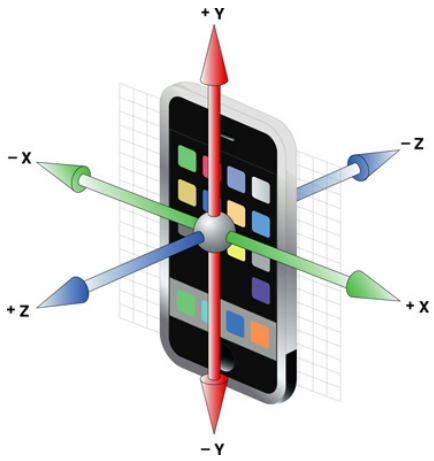



Figure 4.9: Orientation of the device axes [11].

## 4.3 Coordinate spaces

Because the accelerometer and magnetometer measure the direction values relative to the device, and expressing feature locations in device coordinates directly is not practical, iBetelgeuse uses multiple coordinate spaces and appropriate transformations to convert between these coordinate spaces. This section describes the coordinate spaces used by iBetelgeuse, and what these coordinate systems are used for.

### 4.3.1 Device space

Device space in iBetelgeuse is defined as the coordinate space in which sensor values are returned. Its origin is at the center of the device, where the X and Y axis respectively point towards the right and top of the device and the Z axis points out of the screen towards the user, as seen in Figure 4.9.

Device space is not only used to represent sensor values, but it is also the “view” space used for rendering the features, because the camera’s and device’s orientation are equal, and the difference in location (from center of the device to the camera) is insignificantly small.

### 4.3.2 Earth-Centered, Earth-Fixed space

Earth-Centered, Earth-Fixed (ECEF) is a Cartesian coordinate system in metric units with its origin in the center of the Earth, its X axis towards the the intersection of the prime meridian and the equator, and its Z axis towards the North pole (see Figure 4.10). Its Y axis is perpendicular, passing through the intersection of the equator and the $90^o$ meridian. [6]

All WGS84 coordinates are converted to ECEF coordinates before any processing is done, mainly because metric coordinates are easier to work with for determining distances and orientations of objects.

ECEF coordinates have a magnitude in the order of $1 \cdot 10^6$ m and our application needs to be able to represent differences with an order of magnitude of 1 m, single precision floating points, which store about 7 significant digits, are only just sufficient for representing an accurate ECEF point. Performing computations with these values however quickly introduces loss of precision, which makes single precision floating point ECEF values unusable for our purposes.

Because the iPhone view transformation matrices always use single precision floating point values, using ECEF coordinates directly in these matrices does not work well. For this reason, Earth-Fixed (EF) space was introduced. EF space is a translated version of ECEF, so that the coordinate values become smaller and floating point values have sufficient precision to represent and manipulate position

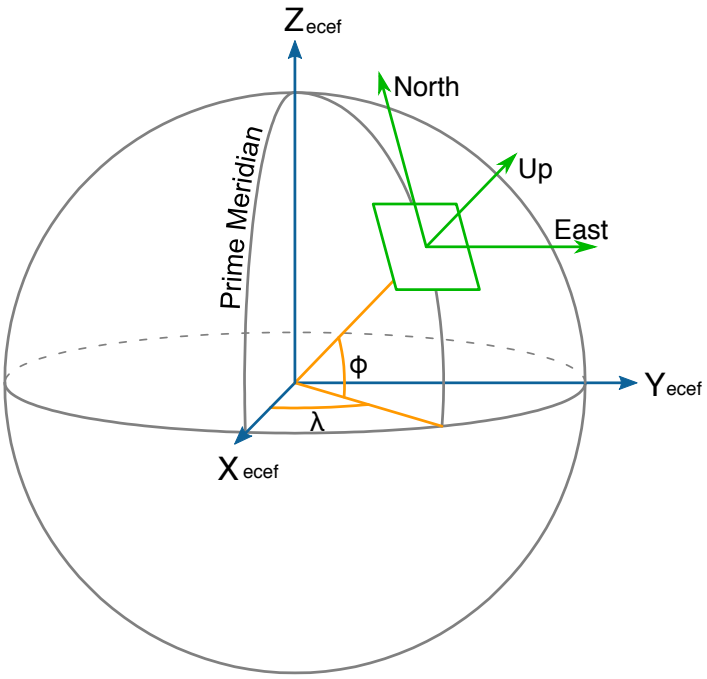



Figure 4.10: ECEF (in blue) and ENU (in green, where X is East, Y is North and Z is Up) coordinate systems [11].

coordinates accurately. Currently, this offset is set so that EF's origin is in the current location of the device, but choosing a value that never changes may significantly improve performance (see Section 6.11).

### 4.3.3 East, North, Up space

The East, North, Up (ENU) coordinate system is defined relative to a given location on the Earth [6]—in iBetelgeuse, the current location of the device. This location is used as the origin of ENU, and ENU's X, Y, and Z axis point towards the East, towards the North, and towards the Sky respectively (see Figure 4.10). This space uses metric units.

The ENU coordinate system is used in iBetelgeuse for various reasons. One reason is that it is more convenient to compute the device-to-ENU and ENU-to-ECEF transformations separately than to compute the device-to-ECEF transformation directly, because the device-to-ENU transform depends only on the device's orientation and the ENU-to-ECEF transform depends only on the current location. Another reason is that the GDDF specification defines an offset in ENU space for text and image features.

### 4.3.4 Other coordinate spaces

Other coordinate spaces that are used in iBetelgeuse are the screen, perspective, map, and radar spaces. Since these are special-purpose spaces used for rendering only, they will be described in the sections about rendering: Sections 4.5 and 4.6.

## 4.4 Spatial state determination

This section describes the methods employed to determine the device's spatial state. The spatial state can be subdivided in a position component and an orientation component. This section first describes how the position is determined, followed by a description of how the orientation is determined. Finally, it describes the filtering that is required to smoothen on-screen movements of features due to orientation changes.

### 4.4.1 Location determination

Determining the device's current position is relatively easy. iPhone already provides functions to request the current location, which returns the current WGS84 coordinates, and several statistics about the accuracy of this location. It supports GPS position determination whenever there is an unobstructed line of sight between the device and a sufficient number of GPS satellites. Otherwise, it can estimate the position based on cell tower or Wi-Fi reception.

Since position updates are delivered infrequently (once per second at best) and are imprecise (often, the same location is returned even after moving the device) filtering these values is not beneficial. It is possible to interpolate the values as navigation software does [18], but this is mostly useful at higher speeds. Also, it likely introduces more problems than it solves. For example, virtual objects may continue to move even though the user is already standing still.

In our implementation, location updates are only filtered to remove undesired noise spikes, caused by the initial cell tower-based position estimation. For this purpose, we define a maximum acceptable speed $v_{max}$ and store the accuracy $r_{n-1}$ and timestamp $t_{n-1}$ of previous samples. When a new sample $(r_n, t_n)$ is received, we extrapolate the accuracy of the previous sample by the maximum velocity: $r'_{n-1} = r_{n-1} + v_{max}(t_n - t_{n-1})$. If $r'_{n-1} < r_n$, the previous sample was more accurate than the new sample, and the new sample is ignored completely. Since our testing showed that spikes of inaccuracy have an order of magnitude of $1 \cdot 10^3$ m, even pretty high values of $v_{max}$ work well. To be safe, iBetelgeuse uses $v_{max} = 200$ m/s, this would allow the application to be used in a commercial airliner.

### 4.4.2 Orientation determination

The iPhone has two important 3-axes sensors built-in, that can be used to determine the device's current orientation: the magneto- and accelerometer. These sensors return a vector towards the magnetic North and a vector towards the center of the Earth (unless accelerating). The 3D vectors returned by these sensors are generally not parallel, and can therefore be used to unambiguously identify the current orientation of the device.

Before any processing is done on the compass, it is corrected for magnetic declination. Magnetic declination is the angle between the direction towards the true North pole—as used on maps—and the magnetic North pole—as measured by the sensor. The magnetic declination can be approximated by a model that depends on the current location on the Earth's surface and the current date, but iPhone is fortunately able to do this computation for us; the magnetic declination can be extracted from iPhone's location data.

From the vector towards the true North and the vector towards the sky—we store the inverted accelerometer value for convenience—a transformation matrix from ENU to device space can be computed. Since the accelerometer is much more accurate and suffers from much less noise than the magnetometer[1], the up vector measured by the accelerometer is directly used as the definition of the Z axis of ENU in device space. This leaves only the heading undefined, which is found by projecting the North vector as measured by the sensor on the XY plane, and then using the projected North vector as

[1] If the device is moved, the accelerometer's values may be inaccurate due to the acceleration, but it generally still is more accurate than the magnetometer.

the ENU Y axis in device space. Device and ENU space are both defined in the location of the device, hence no translation is necessary.

Once the transformation matrix is computed, the matrix is converted to a quaternion in order to apply filtering.

### 4.4.3 Orientation filtering

In order to improve the smoothness of the determined orientation, we apply filtering on the quaternion that is found by combining the accelerometer and magnetometers into a single quaternion as described above. There are two reasons for filtering:

- the accuracy of measured samples can be improved by combining multiple samples, and
- the signal can be smoothened to look nicer on the screen.

iBetelgeuse achieves both effects by implementing a custom filter, designed specifically for iBetelgeuse. This filter models the orientation as a quaternion, which changes over time due to angular velocity. This angular velocity is assumed to change infrequently, and therefore be approximately constant over a time window of about one second.

Whenever a new sample is received, the filter approximates the angular velocity by taking the weighted average of the angular velocity as measured in the last $d$ samples—currently $d = 33$, which corresponds to about 1 second—and computes an estimate of the device's new orientation $e_n = \text{normalize}(q_{n-1} + \frac{1}{2} q_n \omega_n \Delta t)$, where $q_n$ and $q_{n-1}$ are the current and previous output values, $\omega_n$ is the approximated angular velocity, and $\Delta t$ is the difference in time between sample $q_{n-1}$ and sample $q_n$. This formula basically extrapolates the orientation by integrating the angular velocity over time, using the quaternion derivative $dq = \frac{1}{2} q \omega_n$.

Because the approximated angular velocity nor the integration step are exact, $e_n$ will diverge from the true orientation after a while. For this reason, a correction step must be performed. We use an exponentially converging corrector: the correction magnitude is proportional to the estimated error with a ratio $c$. To avoid jitter, the input value $q_n$ that is used for computing the estimated error $(q_n - e_n)$ is first filtered using a spherical moving average filter with a relatively small window, currently configured to be about 0.2 seconds.

To improve performance, two changes to this filter have been implemented:

1. To reduce overshoot and achieve faster convergence, the correction factor $c$ is increased proportionally to the magnitude of the angular velocity. When the device moves fast, $c$ is increased, causing the filter to converge quicker by allowing more jitter; this is not a problem because when the device moves, this jitter is less noticeable.
2. After determining the angular velocity $\omega_n$, but before computing the estimate $e_n$, the angular velocity's magnitude is stabilized (reduced) by a constant value $s$.

   This reduction is cancels out angular velocities smaller than $s$ are completely: the angular velocity is then assumed to be zero. Angular velocities with magnitudes larger than $s$ are changed so that their direction remains unchanged, but their magnitude is subtracted by $s$. Integrating this indeed leads to slightly incorrect estimates, but that does not matter much as it is automatically corrected by the corrector; not reducing estimates with large magnitudes at all, however, leads to irregular movement of features on the screen.

## 4.5 Feature rendering

In order to display the virtual world on the screen of the device, the features in the dimension need to be projected onto the screen depending on the orientation of the device and the relative location of the

features. This section describes how this projection is done.

Let the following homogeneous coordinates be in the coordinate system of a feature:

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Then we need to perform the following transformations in order to project these coordinates onto the screen of the device:

$$\mathbf{x}' = SPD^{-1}L^{-1}O\mathbf{x}.$$

where:

- $O$ is the object transformation, that transforms object coordinates to EF coordinates (see Section 4.3.2 on page 25) depending on the location and orientation of the object in the world;
- $L^{-1}$ transforms EF coordinates to ENU coordinates, depending on the location of the device (see Section 4.3.3 on page 26);
- $D^{-1}$ transforms ENU coordinates to device or view coordinates, depending on the orientation of the device (see Section 4.3.1 on page 25);
- $P$ performs a perspective transformation, depending on the properties of the camera of the iPhone; and
- $S$ performs a screen transformation, depending on the properties of the screen of the iPhone.

Each of these transformations will now be described in more detail.

### 4.5.1 Object transformation

The object transformation determines the location and orientation of the object in the world.

Each feature has a latitude, longitude and altitude corresponding to a location in the real world. These coordinates are first converted to the EF coordinate system, which is a relatively simple conversion from WGS84. In addition, each feature has a certain offset in meters in the local ENU coordinate system. This offset is also converted to EF and then added to the coordinates previously calculated.

The orientation of the object is set such that it faces the viewer. In other words, it must face the location of the device. This is done by calculating the appropriate 'look-at' transformation matrix. (This is similar to what the `gluLookAt` function of OpenGL does.)

Together, the translation in EF coordinates and the look-at orientation matrix form the object transformation. This transformation is calculated in the `updateWithSpatialState:usingRelativeAltitude:withDistanceFactor:` method of the `ARFeatureView` class.

### 4.5.2 Local transformation

The local transformation transforms the local ENU coordinate system to the global EF coordinate system. (Note that the inverse of this transformation was used at the start of this section.) This transformation is determined by the location of the device and is calculated by `ARSpatialState`.

### 4.5.3 Device transformation

The device transformation transforms the device coordinate system to the local ENU coordinate system. (Note that the inverse of this transformation was used at the start of this section.) This transformation is determined by the orientation of the device and is also calculated by `ARSpatialState`. It is the matrix representation of the quaternion that defines the device's orientation.

### 4.5.4 Perspective transformation

The perspective transformation projects the three-dimension world onto a two-dimensional view plane. Since we are trying to match the virtual world to the real world, which is being captured by a camera, the parameters of this perspective transform are determined by the properties of this camera.

The appropriate perspective transform for the current camera is calculated by the `ARCamera` class from the focal length (of the lens) and the size of the image plane (image sensor) using simple physics rules. The view plane is a $[-1, 1] \times [-1, 1]$ plane that corresponds to the smallest square that encloses the image plane.

Note that camera distortions are not included in the calculation.

### 4.5.5 Screen transformation

The screen transform scales and translates the $[-1, 1] \times [-1, 1]$ view plane, onto the smallest square that encloses the screen bounds. This transformation is calculated in `ARFeatureContainerView`.

### 4.5.6 Applying the transformations

The object transformation $O$ is applied by `ARFeatureView` to itself. In `ARFeatureContainerView`, the inverse of the local transformation, the inverse of the device transformation, the perspective transformation and the screen transformation are concatenated (i.e. $SPD^{-1}L^{-1}$) and the result applied to all subviews (which are features views).

The graphics system of the iPhone will take care of applying these transformations when the views are being drawn to the screen buffer.

## 4.6 Radar rendering

The radar is drawn as a collection of three layers that are rendered on top of each other:

1. the *background* layer, which basically shows a big gray circle showing the range of the radar;
2. the *extent of view* layer, indicating in what direction the user is looking, and which objects are visible when looking in that direction; and
3. the *blips* layer, which contains all radar blips[2].

### 4.6.1 The background layer

The background layer is the simplest of the three; it is rendered once and never changes orientation. The other two layers are displayed on top of this layer, and should never render anything outside of this area.

[2] a radar blip is the representation of an object as a small circle on the radar

### 4.6.2 The extent of view layer

The extent of view layer displays a triangular gradient that indicates in what direction you are looking. This view is rotated so that the triangle is always facing the sky. If the device is held horizontally, the looking direction is undefined, and the triangle is hidden.

This layer is redrawn every time the orientation changes, since the size of the triangle depends on the pitch of the device. It computes the current heading and draws the triangle in the reverse direction of this heading, so that it always faces the sky.

### 4.6.3 The blips layer

For performance reasons, the blips layer can not be drawn every time the orientation changes. With more than ten features, this would cause display lag because the radar rendering would become too slow. Therefore, this view is only drawn when the features or the current location changes. In other cases, this layer as a whole is only rotated to match the user's current looking direction.

Drawing the blips on the layer's image buffer is done by taking the ENU coordinates for each feature, ignoring the Z coordinate and scaling the X and Y coordinates to match the radar's range. If this coordinate fits in the radar's window, a blip is drawn at that location on the image.

### 4.6.4 Flattening the radar

Since the radar is a 2D view, a choice must be made in how the radar is projected. First, assume that the device is horizontal; then the radar should work pretty much like a compass: features in any direction should match the displayed direction on the radar. This means that features that lie towards the East are also displayed on the radar towards the East. When the device is not held horizontally, we assume that the topmost point of the radar's view should always match the direction in which the user is looking. This means that if the user moves forward, the feature blips move down on the screen. Basically, this means that the device should be pitched so that it is horizontal with the screen facing upwards.

Pitching the device down in iBetelgeuse is done by introducing *map space* and *radar space*. Map space originates in the current location, with the Z axis pointing upwards and the Y axis pointing in the (projected) viewing direction. Radar space is the "flattened" device space. i.e. device space pitched down so that the pitch is -90°.

Map space is merely used as an intermediate step in computing radar space; it splits up the radar $\rightarrow$ ENU transform to a radar $\rightarrow$ map and a map $\rightarrow$ ENU transform. The consecutive transforms can intuitively be seen as first applying the heading, followed by the roll (the pitch is already -90°, since the Z-axis of ENU points upwards).

### 4.6.5 Rendering the radar

To use radar space for rendering blips on the radar, the radar $\rightarrow$ ENU transformation is applied to the blips layer. This seems counter-intuitive, because the blips layer is drawn in ENU space, and the resulting points should be coordinates in radar space. Our only intuition here is that, because the radar rotates with the device, this rotation has to be cancelled. It can however be proven to be correct:

First, note that by drawing any coordinate system to the screen, an implicit transformation is performed to device space. Recall that the blip layer is defined in scaled ENU space. Assuming we draw the radar's blip layer directly to the device (using an identity transformation for the layer), a point $p$ that is drawn on the screen ends up in the real world at position $T_{ENU \rightarrow device}p$, where $T_{ENU \rightarrow device}$ is the ENU $\rightarrow$ device transformation matrix.

Ideally, we want to be able to draw points on the radar so that they will appear exactly in that location in the real world. i.e. if we draw a point in ENU coordinate $a$, then after drawing, we want that point to be

placed in the real world ENU location $a$.[3] This means that an identity transformation must be performed. However, as explained above, there will always be an implicit ENU $\rightarrow$ device transformation. To correct this transformation, consider the following consecutive transformations:

1. ENU $\rightarrow$ device—the implicit transformation caused by drawing the image on the screen;
2. device $\rightarrow$ radar—a transformation that pitches the device's orientation down;
3. radar $\rightarrow$ ENU—a rotation to get back to ENU space.

Since ENU $\rightarrow$ device $\rightarrow$ radar and radar $\rightarrow$ ENU are both rotations around the Z axis, their order can be changed to:

1. radar $\rightarrow$ ENU;
2. ENU $\rightarrow$ device;
3. device $\rightarrow$ radar.

Note that now, transformation 1 represents the transformation that should be applied to the view (which is done first), transformation 2 represents the implicit drawing transformation, and 3 represents rotation pitching the radar down so that it is flat. This third rotation can not be performed properly by the device, because the radar must show a top-down view of the world. This rotation is left for the user's interpretation: if the device is held vertically, the user should interpret any point that is shown on the top of the device to be in front of him rather than above him.

Also note that due to the reordering, the coordinate transformations no longer match: the consecutive application of the three transformations above give a radar $\rightarrow$ radar transform, but this transformation is applied to coordinates in ENU space, which led to the intuition that using this transformation seemed wrong.

## 4.7 QR scanning

Instead of implementing the QR scanner ourselves, we used a library—ZBar [10]—that scans QR codes as well as other types of (bar)codes. Since ZBar is available as an application for the iPhone, using it is very easy and comes down to sending an image to the QR scanner and receiving details about the QR codes that were found, if any. To reduce battery power consumption caused by the CPU intensive QR scans, iBetelgeuse only scans for QR codes once per second.

[3] This is normally not possible because the screen plane is 2D and bounded, but that will be covered later.

# Chapter 5

# Product quality

This chapter attempts to give an assessment of the quality of the software by exploring a small number of criteria: functionality, usability, performance, reliability, maintainability and portability.

## 5.1 Functionality

iBetelgeuse was written to be compatible with the ARena backend, and implements everything that is required to work well with ARena. It is also mostly compatible with Betelgeuse for Android, with the exception of 3D models, but there are currently no tools to create 3D models for Gamaray either.

The integrated QR scanner makes loading dimensions easy, but the dimensions must always come from an external source. This product is of very limited use without a backend feeding it with information, just like Betelgeuse for Android. Combined with a backend, however, it is more flexible than comparable browsers such as Layar, because anyone can build a new dimension, and because these dimensions are aimed towards interaction, allowing for dynamic worlds.

## 5.2 Usability

The browser is designed to be usable without reading any documentation, and follows the iPhone usability guidelines as closely as possible. It works similar to other browsers, and should therefore be comfortable to users who already used a different AR browser.

The user interface mostly consists of the camera image, features, and overlays. It should be clear to users that this represents the augmented world, since this is the main goal of the application. The usability of the virtual world itself mostly depends on the loaded dimension; i.e. whether features have clear names and whether it is obvious what action tappable features and overlays perform.

Other than that, it shows a radar, which was designed to match what the user typically wants: items that are displayed on the top of the device should be located in front of the user, regardless of the orientation of the device, so that the user can see in what direction the device should be rotated to view the feature.

The QR scanner is somewhat less convenient because the user can not start scanning a QR code immediately: they must first enter the scan mode. The modes are clearly distinguishable, but only once the user is used to the application. Users may forget to enter this mode, and wonder why the QR code is not scanned. The separate mode is necessary, however, for performance and energy consumption reasons. The QR scanner does, however, lead to higher usability since scanning a QR code is probably the easiest way to load a dimension.

Finally, iBetelgeuse shows icons indicating its status: whether the loaded dimension is up-to-date,

and whether the location and orientation are accurately determined. These icons should be self explanatory once the user has seen them a few times, but may be unclear the first time they show up.

## 5.3 Performance

iBetelgeuse was tested to perform well on the iPhone 3GS, even when several hundred features or overlays are simultaneously displayed on the screen. Various optimizations are possible, but iBetelgeuse should already perform well in any realistic situation in its current state. Note that the backend is responsible for appropriately filtering the features in large dimensions, not only to avoid performance and bandwidth problems, but also to avoid presenting a chaotic world to the user.

## 5.4 Reliability

This application relies on the availability of an internet connection and the availability of reasonably accurate location data.

When the internet connection fails or when the backend fails to provide a valid dimension, the application will display this to the user and stop refreshing automatically. The user can manually choose if and when to retry. The application will continue to work with the most recently loaded dimension.

Whenever the location or position data is inaccurate, an indicator icon is displayed to the user. If the compass detects interference, instructions to calibrate the compass are displayed to the user. The application will continue to use the inaccurate spatial state, which could lead to an inconsistent view, but it can not perform better if no more accurate data is available.

## 5.5 Maintainability

Various techniques were applied to keep the code maintainable, most importantly: a solid architecture, unit testing, documentation, and clearly defined coding conventions.

The architecture was designed to have no cyclic dependencies. Delegation is used to avoid cyclic dependencies. With the exception of `ARMainController`, each class has one dedicated purpose. `ARMainController` should in fact rally be refactored, as is commented on in Chapter 6 on page 36.

A large part of iBetelgeuse was extensively tested using unit tests. Classes fit for unit testing have a statement coverage of at least 90%, and 97% on average. Average branch coverage is above 80%, but many cases are not correctly detected by the coverage testing tool (such as assertions and some Objective-C language constructs). Not all classes are fit for testing, such as view and controller classes. Functionality of these classes was tested manually, although we do have a determined way for doing such a test.

Documentation was added inline with the code, and written for all classes and properties—describing what a particular class or property does—and all methods—describing what a particular method's purpose is, what its pre- and post conditions are and what the arguments and return value represent. Documentation is written in Javadoc style, which can be parsed by Doxygen to generate HTML documentation. In addition to the in-code documentation, this report describes the overall architecture of the program, and details various aspects. This report, or part thereof, will be copied to the Finalist internal wiki to serve as documentation.

The code uses consistent conventions and a verbose naming scheme. Preference was given to longer descriptive names over shorter unclear names. There are some inconsistencies in conventions, but these are minor and infrequent.

## 5.6 Portability

Most of iBetelgeuse is written in Objective-C; this language was chosen because the iPhone API is only works with Objective-C. It is possible to combine Objective-C with C or C++, which has been done for some classes, but using two languages simultaneously is less convenient and clear, and was therefore mostly avoided.

All math types and their associated functions are written in C, since it is more convenient and efficient to be able to work with these value types on the stack and pass them by value. These types and functions are portable and can be used in any other project (as long as the definition of CATransform3D is also copied).

Other model classes are written in Objective-C and will likely have to be ported to another language if they are to be reused. They are designed to be reusable, so porting or reusing part of the classes without the need for porting unrelated classes is possible.

The rendering code and main controller are probably the least portable aspects of iBetelgeuse: they are specific to the iPhone and this application, and will have to be reimplemented if iBetelgeuse is ported to another platform.

# Chapter 6

# Possible improvements

Although the browser that has been created is a fully functional product, there are still several aspects that can be improved. This section describes improvements that we may have implemented or researched if we had more time. These improvements are specific to the implementation of the iBetelgeuse client. Section 8 describes more general recommendations, for example for changes that also involve modifications of the backend and GDDF specification.

## 6.1 Refactoring ARMainController

During the project, the implementation of ARMainController grew larger and larger; in hindsight, this class' functionality should have been split over multiple classes, with different responsibilities. We suggest splitting this class into:

- `ARDimensionView`, a subclass of `UIView` that contains the views that are visible in dimension viewing mode: the camera view, the feature view, the radar view, the menu button, and the status indicators;
- `ARQRScannerView`, a subclass of `UIView` that contains the views that are visible in QR code scanning mode: the camera view, the cancel button and the rectangle indicating that the QR scanning mode is active;
- `ARDimensionController`, a controller class for `ARDimensionView` that controls dimension refreshes and handles UI events;
- `ARQRScannerController`, a controller class for `ARQRScannerView`, that controls background QR code scans and handles UI events; and
- `ARMainController`, this class should control the current operating mode—dimension or QR scanning mode—and it should manage the camera view, since only one should exist, and both dimension and QR scanning mode need to show it when they are active.

## 6.2 Tweaking filtering parameters

Observations during testing indicates that sensor noise differs significantly between indoor and outdoor use. The compass appears to be, counterintuitively, less accurate outdoors. Because we tweaked our filter values indoors, the values are less well configured for outdoor use. Currently, the filtering is not aggressive enough when used outside, which means that the features move around too much when using the application while walking. The filter's stability outside can likely be improved significantly by tweaking the filtering parameters for outdoor use.

## 6.3 Compensating for user-induced acceleration

Because the accelerometer does not only measure gravity, but measures acceleration as well, the measured gravity vector is unreliable when the device is moving. Since any non-horizontal acceleration changes the magnitude of the gravity vector, a change in magnitude could indicate movement. When this happens, the filter values should be chosen to be more aggressive: they should stay closer to their previous value. Theoretically, this would improve stability when moving the device, for example when walking.

## 6.4 Splitting the filter

Orientation filtering is performed for two reasons: combining previous sensor values and smoothing the results for the eyes of the user. It may be beneficial to split these up in two filter steps; a Kalman filter for computing the best orientation given those measurements, followed by a smoothing or stabilizing filter to present the results nicely to the user.

## 6.5 Adding a distance indicator

Currently, it is hard to see how far away a feature is from the current location. Because features currently do not scale with distance, it is not possible to determine the distance to the feature by merely looking at their projected location, and only a very rough estimate can be made by looking at the radar view. It would be convenient for the user to display the distance to the feature that the user is facing on the screen.

## 6.6 Improving feature display

When two features are located in a straight line from the user's perspective, their views overlap, which sometimes leads to flickering as the features are not rendered in the right order. It may be inherent to the way the iPhone renders the views and therefore unfixable, but there may be a workaround to fix this issue.

Another problem regarding overlapped features is that currently, likely due to transparency settings, a feature overlapping another feature is not well readable, even if they do not flicker. This can be fixed by, for example, applying a birds-eye view—as seen in Layar—that places further away objects higher on the screen.

## 6.7 Improving dimension refreshes

Currently, when the dimension is refreshed, all of the dimension's content is removed and reloaded. Because image objects and assets are recreated as well, the asset is reloaded from the web. Even though HTTP caching is used, reloading the assets does not happen instantaneously: the images flash when a dimension is reloaded, which looks awkward. A solution would be to reuse views and assets that have the same identifier; it may be slightly more efficient in general as well, but it is not very easy to implement. It is likely better to implement a small instantaneous caching system: store all currently loaded assets based on URL, and return the already fetched asset if their URL is requested again.

## 6.8 Adding calibration

As is discussed in Section C.4 on page 57, we have observed that objects that should be parallel to the horizon do not always line up with the real horizon. This may due to the fact that the accelerometer is not always put perfectly straight into the iPhone. This can be solved by calibrating the accelerometer relative to the device. We were unable to get a fully satisfactory solution to this calibration problem.

A less satisfactory but possibly sufficient solution to this problem could be a simpler calibration procedure that asks the user to keep the device right in front of him/her and straight in relation to the horizon, and only compensate for this case (vs. for all axes as was proposed in C.4).

## 6.9 Updating for iOS 4 and iPhone 4

The next generation iPhone, iPhone 4, which was announced and released when this project had already been started, has a gyroscope in addition to an accelerometer and a magnetometer.

As indicated in Appendix C.3 a gyroscope is often used to track the orientation of an object and could significantly improve the browser's capability to determine the orientation of the device.

In addition to the hardware changes, the new version of the operating system, iOS 4, which accompanies the iPhone 4, has a new framework called Core Motion dedicated to tracking the orientation of the device. It has a class similar to `ARSpatialStateManager` that takes advantage of the new gyroscope.

iBetelgeuse may also benefit from Core Motion. However, we do not know whether the framework is backwards compatible with devices without a gyroscope, such as the iPhone 3GS with which we have been working. We also do not know whether the orientation data provided by the framework has any undesired filtering applied to it.

## 6.10 Testing in different places and environments

Our testing has been limited to indoors at Finalist and outdoors in Rotterdam. The magnetometer is very sensible to interference and must also be corrected for magnetic declination (the difference between magnetic North and true North). Both of these factors are dependent on different environments and the place on Earth. For example, the magnetic declination in the Netherlands is near zero, but can be over $20^{o}$ at the west coast of the United States.

Note that the behavior of the browser is undefined when used very close to the magnetic North or South poles, because it will be unable to determine the direction of North. This problem can probably not easily be solved. However, using an augmented reality browser at the magnetic North pole is a highly unlikely use case.

## 6.11 Precomputing feature locations

It is possible to significantly reduce the amount of transformations required for rendering by choosing the offset between ECEF and EF once (at application startup), and then never updating it again. The initially configured EF frame should be more than accurate enough for local coordinates, which only introduces problems if the user travels all around the world without ever closing the application; an unlikely scenario.

By only computing the EF offset once, whenever a dimension is loaded, all feature positions in EF space can be precomputed. Now, only the orientation of the features has to be changed when the user moves the device, but their locations no longer have to be recomputed every time the EF coordinate

space changes. Relative altitudes (as specified in the GDDF) can be implemented by fixing the device's altitude to zero, instead of changing every feature's altitude. For reasonable altitudes, this should lead to almost identical results.

Our tests indicate that the matrix transformations are currently not a bottleneck, but if they ever become a bottleneck due to different usage in the future, this could be a significant performance enhancement.

# Chapter 7

# Conclusion

The goal of this project was to develop an augmented reality browser for the iPhone platform, that can be used by Finalist for their existing ARena web service as well as for future applications. This chapter briefly recalls some of the results we have obtained and describes why we think we have succeeded in reaching the stated goal.

The application we have developed, iBetelgeuse, is an iPhone application that projects virtual objects over the camera image of the device, creating the idea of an augmented reality. It is compatible with the Gamaray browser, and is therefore fully compatible with the existing ARena backend. The browser is easy to use, partially due to its integrated QR scanner and simple interface, and is focused on responsiveness and interaction; two aspects that are unique to iBetelgeuse.

Additionally, the browser is ready for future extension. It has a fairly clean and well documented implementation, making it very maintainable. Also, the placement and rendering of virtual objects is realistic[1], not limiting future applications to what is currently possible with the Gamaray Dimension Description Files.

[1] Unlike Gamaray, which only approximates the location of objects by using WGS84 as if it was Cartesian.

# Chapter 8

# Recommendations

In this final chapter of this report, we would like to do some general recommendations regarding augmented reality as well as some recommendations to Finalist concerning further development of the Betelgeuse augmented reality browsers for Android and iPhone.

## 8.1 Augmented reality

First, we want to recommend everybody who considers augmented reality on a mobile device as a solution to a problem to carefully consider whether it adds real value instead of just being a gimmick.

For example, if a user wants to find the nearest ATM (in a city not laid out in a grid), it is much easier to find it using a map and a compass than using current augmented reality browsers. Current browsers show a marker in the ATM's general direction, often without indicating a route or even a distance. This is as difficult as walking the shortest route to a tall building that is a kilometer away. If the augmented reality browser would project the walking route onto the street, for example, this problem would partly be taken away. However, we think current browsers (and compasses found in mobile phones in particular) may not be accurate enough to do this.

Of course, in the case of ARena, giving students a map would be much less delightful than using augmented reality, making it a good solution in this case.

## 8.2 Mobile devices

Second, we would like to recommend manufacturers of mobile phones to include a gyroscope in their devices in addition to an accelerometer and a compass. This allows augmented reality browsers to determine the device orientation more accurately. This improves the augmented reality experience, which is currently suboptimal in our opinion.

## 8.3 Augmented reality format

Third, we want to give some recommendations to Finalist, since they are planning to extend the Gamaray Dimension Description File. Our experience is that this format is very limiting. Some of the problems we encountered are:

- Overlays can only be positioned using absolute screen coordinates. There is no way to reliably position overlays in a static dimension without knowing something about the screen size. While web

services are made aware of the screen size, they still cannot know where default user interface elements—such as the radar—might be displayed.

- Features can only be positioned using absolute WGS84 coordinates. They cannot be placed in relation to the user's location (except for the altitude). This may be useful for certain features that move with the user, such as signs, or for static dimensions that do not depend on the user's location (only the device orientation).
- Features are always rotated to face the user. While this can be useful, it is unrealistic and makes it impossible to project arrows on the street or put posters on sides of buildings, for example.
- Both overlays and features do not have a defined size. Their size is dependent on how text is rendered by the client and what the size of an image is. It is also impossible to display an appropriately sized placeholder for images that are currently being loaded or failed to load. Also, features do not scale with distance, which makes it hard to judge the distance to an object.
- The Gamaray Dimension Description File is badly designed. First of all, XML child elements are used where XML attributes would have been more appropriate. Second, default values or default behavior are undefined for many of the elements. Third, the purpose of `waitForAssets` element is unclear to us and impossible to adhere to when a client would want to load assets lazily. Last, inspection of the Gamaray client source code led us to believe that there are several undocumented elements in the format.

These problems lead us to recommend against extending the GDDF specification. We recommend adopting a KML-based format, such as the existing KARML[1] or ARML[2] formats. There recently has been a proposal to form a W3C working group to create an open standard for such a format [9], but it is not yet clear whether this proposal is going to be accepted by the W3C body.

[1] `https://research.cc.gatech.edu/polaris/content/karml-reference`
[2] `http://openarml.org/`

# Appendix A

# Proof-of-concept

During sprint zero we made a proof-of-concept to determine if we could implement the augmented reality view the way we had planned. This proof-of-concept has various labels positioned in the ECEF coordinate system. It displays them on a gray background at the right location on the screen when the user looks around with the device. See Figure A.1.

Through various Core Animation layers, each label with homogeneous coordinate $\mathbf{x}$ is projected onto the screen as follows:

$$\mathbf{x}' = SPD^{-1}L^{-1}O\mathbf{x}.$$

The matrix $O$ translates, rotates and scales each label (or object) to the desired location and orientation in ECEF.

The matrix $L^{-1}$ transforms the ECEF coordinate space to the local ENU coordinate space. If the device's current location in ECEF is $\mathbf{c}$, then this matrix is derived as follows:

$$L = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ \mathbf{l}_x & \mathbf{l}_y & \mathbf{l}_z & \mathbf{c} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

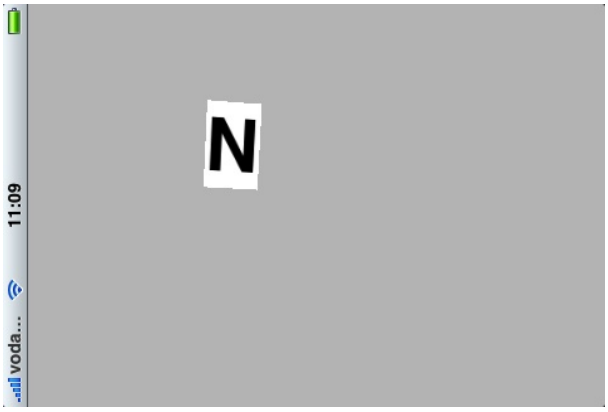



Figure A.1: The proof-of-concept, in this case when looking with the device in the direction of the North pole.

where

$$\mathbf{l}_z = \frac{\mathbf{c}}{|\mathbf{c}|},$$

$$\mathbf{l}_x = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \times \mathbf{l}_z, \quad (\times \text{ is cross product})$$

$$\mathbf{l}_y = \mathbf{l}_z \times \mathbf{l}_x.$$

The matrix $D^{-1}$ transforms the ENU coordinate space to device coordinate space. If the accelerometer readings are given by **a** (vector pointing towards gravity) and the magnetometer readings by **m** (vector pointing towards magnetic North), then this matrix is derived as follows:

$$D^{-1} = \begin{bmatrix} \vdots & \vdots & \vdots & 0 \\ \mathbf{d}_x & \mathbf{d}_y & \mathbf{d}_z & 0 \\ \vdots & \vdots & \vdots & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where

$$\mathbf{d}_x = \frac{\mathbf{m}}{|\mathbf{m}|} \times -\frac{\mathbf{a}}{|\mathbf{a}|},$$

$$\mathbf{d}_y = -\frac{\mathbf{a}}{|\mathbf{a}|} \times \mathbf{d}_x,$$

$$\mathbf{d}_z = -\frac{\mathbf{a}}{|\mathbf{a}|}.$$

The matrix $P$ is a perspective transformation that is derived as follows:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}$$

$$\alpha = 2 \arctan \frac{1}{d}$$

where $\alpha$ is the field of view. In our research report [21], we determined that the smallest field of view of the iPhone camera is about $40^o$, resulting in $d \approx 2.8$.

Last but not least, the matrix $S$ is a transformation that is used to translate and scale the $[-1,-1] \times [1,1]$ domain to the screen domain (while maintaining aspect ratio).

The proof-of-concept worked as expected, so this is a satisfactory technical design for the augmented display.

# Appendix B

# Product backlog

This appendix describes the requirements of the product and the planning of its implementation. The requirements and the planning are defined by the product backlog and the sprint backlog produced in the development process as part of the Scrum methodology [20].

## B.1 About the backlogs

The product backlog is essentially a list of functional as well non-functional requirements that give the product value. As requirements can change for various reasons, the product backlog can be adapted over the course of the project.

Most of the items on the product backlog are formatted as user stories: "as an [*actor*] I want to [*feature*] so that [*business value*]". We left out the actor-part, since an augmented reality browser has only one user role. We also left out the business value-part in cases where it is trivial or obvious. Each user story has a number of points assigned to it, indicating an estimate of the relative effort needed to implement it. Other kinds of items are bugs (defects in existing functionality), chores (necessary without directly adding value) and releases (milestones).

A sprint backlog is a subset of the product backlog that has been chosen to be implemented in a single sprint or iteration. It is roughly ordered by priority and adds up to a certain number of points. At the end of each sprint, there should be a working product that fulfills the requirements given by the sprint backlog. In our case, we had three sprints of two weeks and about 20 points (10 per person) each.

## B.2 Backlogs

Below are the sprint backlogs for the three iterations. Together, these sprint backlogs make up the product backlog.

At the start of the implementation phase of the project, all the items up to and including the milestone *Feature complete for ARena* were decided upon, since this was the primary goal of the project.

Furthermore, at the start of iteration 1 and 2, all items for those iterations were determined. Only in the case of iteration 3, not all items were known from the beginning but instead were added and planned as new issues or missing features were discovered.

| Description | Points | Accepted |
|---|---|---|
| **Iteration 1** (May 14 - May 28) | **20** | |

| Description | Points | Accepted |
|---|---|---|
| I want to see the camera image on the screen so that I can see the real world | 1 | May 17 |
| I want the dimension to be loaded from the URL I opened without the UI being blocked | 3 | May 18 |
| I want to see text overlays on the screen | 1 | May 19 |
| I want to see text features in the virtual world | 2 | May 21 |
| I want to see the iPhone activity indicator when stuff is being loaded over the network | 1 | May 22 |
| I want the screen to update when I rotate the device | 2 | May 22 |
| I want the screen to update when my location changes | 1 | May 22 |
| I want to open gamaray:// URLs so that it works with an external QR scanner | 1 | May 22 |
| I want to see image overlays on the screen with placeholder images | 1 | May 25 |
| I want to see image features in the virtual world with placeholder images | 1 | May 25 |
| I want features with relative altitude to display correctly | 1 | May 25 |
| I want images to be loaded without the UI being blocked | 2 | May 25 |
| I want features not to scale by distance because I expect the same behavior as in the Gamaray client | (bug) | May 25 |
| I want to see the relevant features on the radar | 3 | May 27 |
| **Iteration 2** (May 31 - Jun 11) | **22** | |
| I don't want the browser to crash when an object is at my exact position | (bug) | May 31 |
| I want the dimension to be updated after a certain amount of time | 1 | May 31 |
| I want the dimension to be updated after walking a certain distance | 1 | May 31 |
| I want the browser to respond when I tap a feature or an overlay | 1 | May 31 |
| *Feature complete for ARena* (Deadline: Jun 11) | (release) | May 31 |
| I want the compass to use the true heading instead of the north heading so that the augmented world view better matches the camera view | 2 | May 31 |
| I don't want the browser to update the screen more than necessary so that my battery doesn't run dry | 1 | June 1 |
| I want to scan a QR code to open a dimension | 2 | June 7 |
| I want text overlays to look prettier | 1 | June 7 |
| I want image overlays to look prettier | 1 | June 7 |
| I want text features to look prettier | 1 | June 7 |
| I want image features to look prettier | 1 | June 7 |
| I want the menu button to be less ugly | 1 | June 7 |
| I want the browser to remember the last dimension so that I don't need to re-enter a URL or re-scan a QR code when something happens | 2 | June 8 |
| I want to be able to tap close to a small feature or overlay and still have its action be triggered | 2 | June 8 |
| I don't want the browser to accept just any random valid XML document | 1 | June 8 |
| Fix initial values for filters, to prevent unwanted startup effects | (bug) | June 9 |
| I want the browser to work in landscape mode instead of portrait mode | 1 | June 9 |
| I want the features to move smoothly | 3 | June 9 |

| Description | Points | Accepted |
|---|---|---|
| Set proper UIRequiredDeviceCapabilities values in Info.plist and abstract away device details | (chore) | June 11 |
| showOnRadar setting is ignored | (bug) | June 11 |
| I want system alerts to show in their proper orientation when using the AR browser | (bug) | June 11 |
| **Iteration 3** (June 14 - Jun 25) | **21** | |
| I want to be able to recognize the application by its icon | 0 | June 15 |
| Provide attribution to zbar | (chore) | June 15 |
| I want to see a proper animation when launching the application | 0 | June 15 |
| I want to get feedback about what the application is doing regarding scanning a QR code | 2 | June 15 |
| Add high resolution graphics for iPhone 4 | (chore) | June 16 |
| I want the application to be ready for iOS 4 | 1 | June 16 |
| I want the radar to be prettier | 2 | June 16 |
| I want to be able to refresh the dimension manually, like I can in a web browser | 1 | June 16 |
| I want to get feedback about what the application is doing regarding dimensions | 2 | June 16 |
| Use quaternion-based smoothing for more reliable and natural movement | (chore) | June 16 |
| Add content to about screen | (chore) | June 16 |
| Find memory leaks | (chore) | June 16 |
| Fix location identifier referencing | (bug) | June 17 |
| I want the undocumented radar radius setting to be honored | 1 | June 18 |
| I want to see the center point of the radar when the device is flat | 1 | June 18 |
| I want the performance of text features to be improved so that the browser works smoothly with more than 10 features | 2 | June 18 |
| Improve the speed of the radar because it slows the browser down | 2 | June 18 |
| I don't want to see the location warning icon when my location hasn't changed | 1 | June 18 |
| Find code fit for optimization | (chore) | June 18 |
| Most recent dimension not always loaded without refreshing manually | (bug) | June 18 |
| I don't want 500 alert views to show on top of each other | (bug) | June 18 |
| I want my bearing and pitch to be sent to the server | 1 | June 18 |
| I want to see when a dimension has been loaded and when not | 1 | June 18 |
| I want to see the name of the current dimension | 1 | June 18 |
| I don't want any alerts to popup when the action menu is open | (bug) | June 18 |
| *Ready for App Store* (Deadline: Jun 18) | (release) | June 18 |
| Remove unused code and classes | (chore) | June 22 |
| I want location data to be filtered | 3 | June 25 |
| Find untested code that does need testing | (chore) | June 25 |

## B.3 Burn-down chart

In Figure B.1 on the following page a post-project burn-down chart is given that illustrates the progress made over time in terms of number of points left to be implemented.

The blue line indicates the actual progress made. It matches the expected progress (grey-dotted

baseline) quite well. This indicates that our point estimates for the individual stories were reasonably consistent and our iteration goals of about 20 points were well fit.

The yellow and red lines indicate the progress necessary in order to make the two milestones in the product backlog. Apparently, the target for the first milestone was set too conservatively, and the target for the second milestone (which was determined at a later time) was right on.

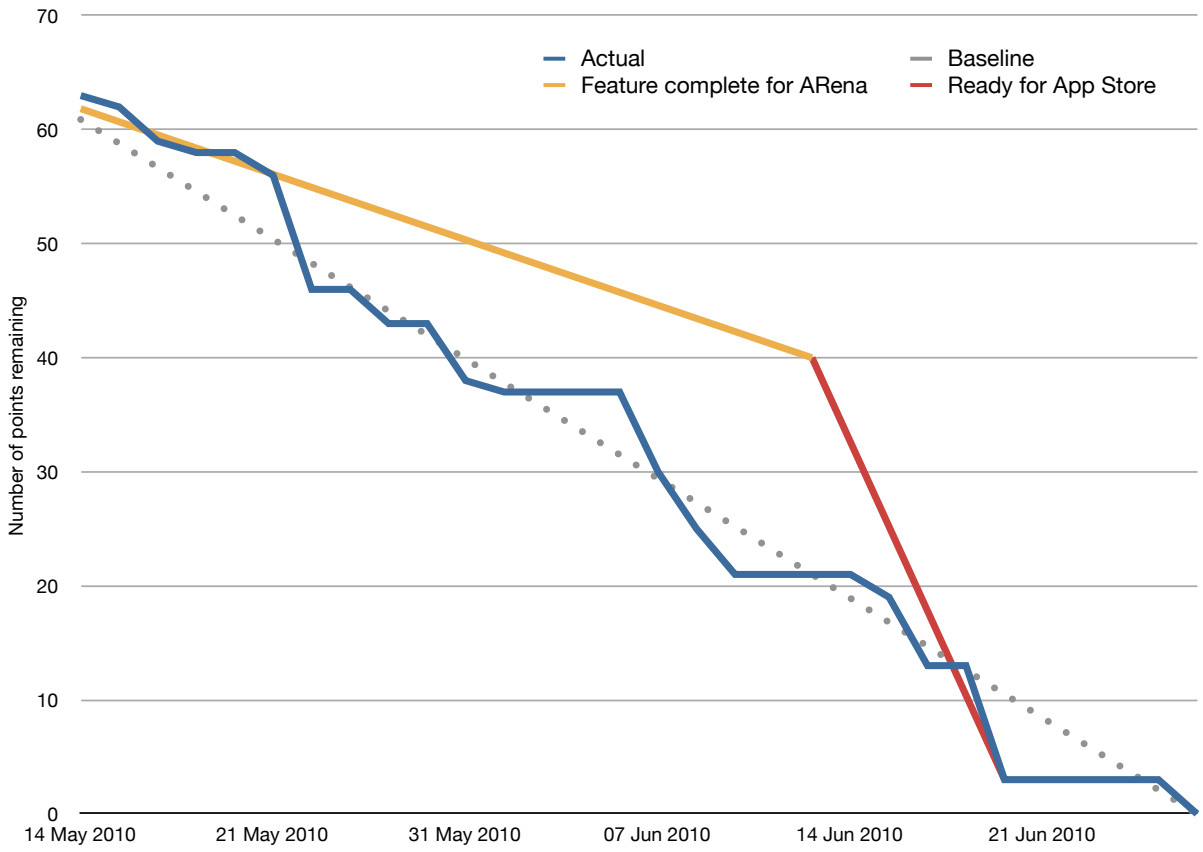



Figure B.1: Product backlog burn-down chart

# Appendix C

# Development history

This appendix contains various pieces of design and research that have been produced over the course of the project. The primary purpose of this appendix is to illustrate the development process described in Chapter 3 on page 8. All content in this appendix should be considered deprecated; in most cases we indicate why it has been superseded or discarded in an evaluation paragraph. Go to Chapter 4 on page 14 for documentation on the current state of affairs.

All design and research was done as needed and when it was needed, except for the more general research described in our research report [21].

## C.1 Initial class diagram

This section describes the class diagram that was made at the start of the project. See Figure C.1 on the next page and Figure C.2 on page 51 for the class diagram. Below, we will briefly explain the role of the various classes.

### C.1.1 Classes

Note: due to the lack of namespacing in Objective-C, it is common to prefix class names. We will use the prefix `AR` (for Augmented Reality).

**ARApplicationDelegate** Manages the application lifecycle and is responsible for setting up the application after it has been launched. It fulfills the role defined by the `UIApplicationDelegate` protocol of the system framework.

**ARMainController** Main view controller of the application that manages the various model and view classes. The model includes the 'dimension' (containing information about objects in the virtual environment) and a class that computes the view transformation from the device's location and orientation. The view includes a camera view, a virtual view, an overlay view and a radar view.

The main view controller may be initialized with a dimension URL, which it starts to load immediately using a `ARDimensionRequest`. If it is not initialized with a URL, it simply displays the camera view along with an empty radar. Later on, the main view controller may obtain a URL through other means, such as by allowing the user to scan a QR code.

**ARSpatialStateManager** Class that continuously determines the device's orientation and location and computes the appropriate transformation from device coordinate space to ECEF coordinate space from this information. It notifies its delegate when the device's spatial state has changed.

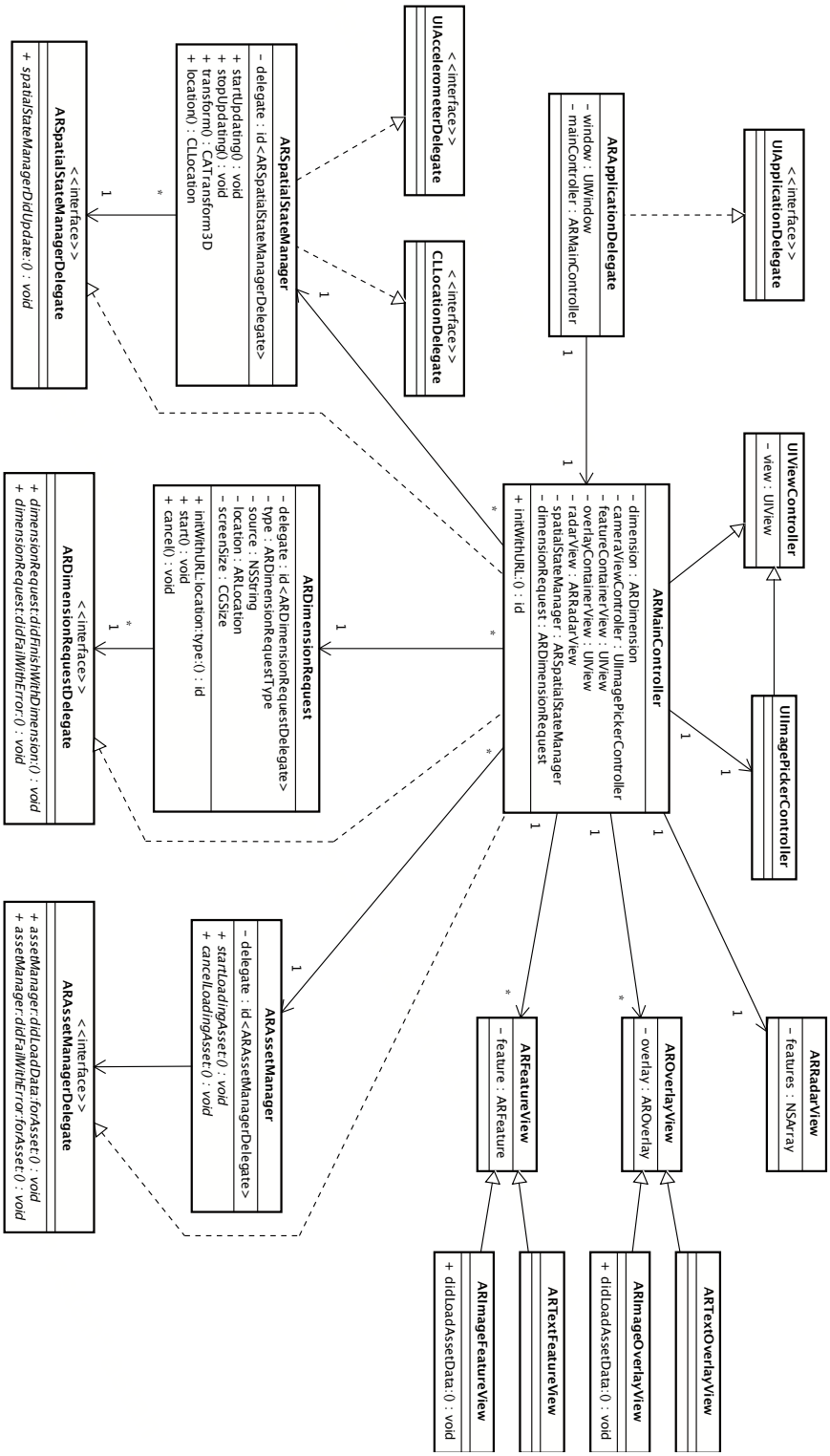



Figure C.1: Initial class diagram, part 1 of 2.

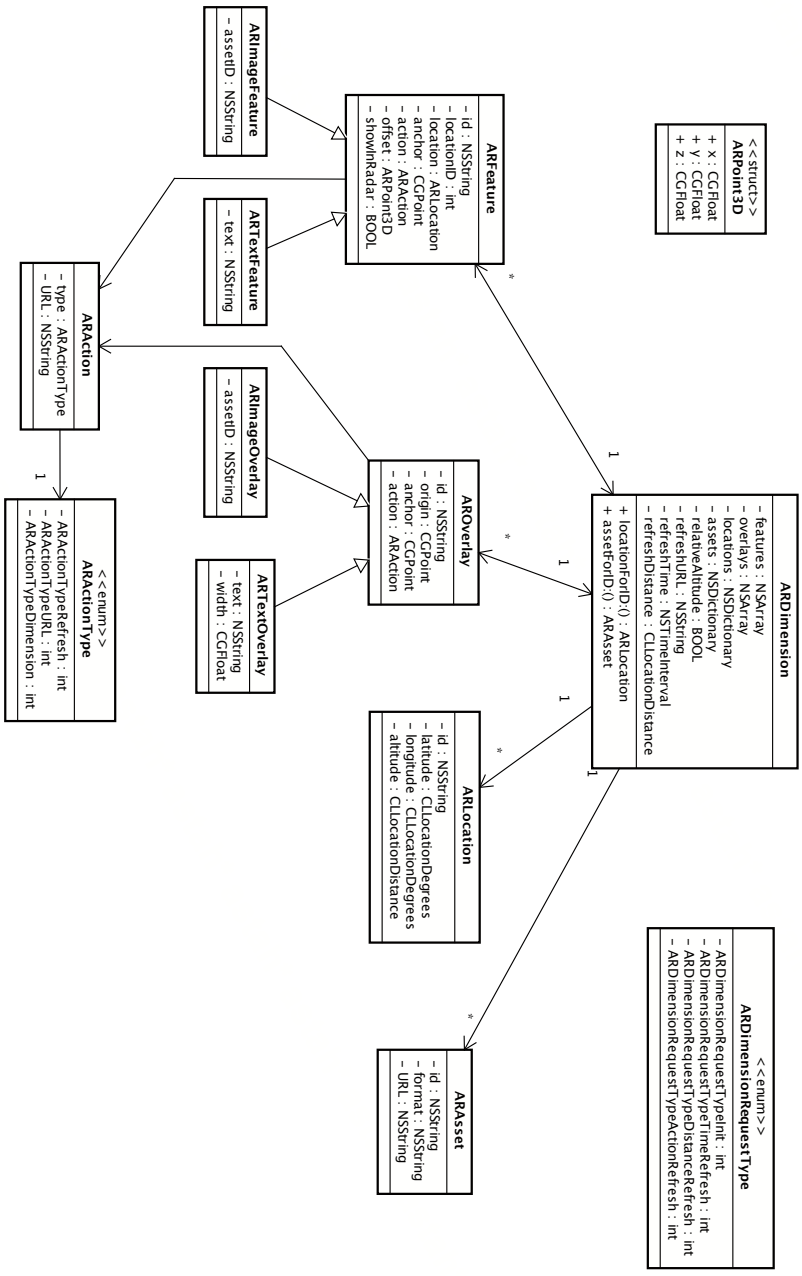



Figure C.2: Initial class diagram, part 2 of 2.

**ARDimensionRequest** Class that can be used to fetch a dimension asynchronously. The dimension URL, the device's location and the event type must be provided at initialization. Parameters like the unique ID and a timestamp are dealt with automatically. Other parameters, like the screen size and event source will have suitable defaults but can be customized.

**ARAssetManager** Class that can load assets asynchronously. It can be loading more than one asset at any given time (even though the actual loading may very well be done in sequence) and will provide its delegate with an `NSData` object when it is done loading an asset. This class may have a caching mechanism.

**ARRadarView** View class that, when given a set of features, displays a radar with markers indicating the location of features relative to the device.

We have not yet defined how it will determine the relative position of features and who's responsibility that is.

**AROverlayView** View class and subclasses that display a certain overlay. The main controller will put these views in its `overlayContainerView` and position them according to the location given in the overlay. The `ARImageOverlayView` will be provided with asset data whenever the `ARMainController` can or wants to do so. In the meantime, it is responsible for displaying a placeholder image.

**ARFeatureView** View class and subclasses that display a certain feature. The main controller will put these views in its `featureContainerView` and position them in ECEF coordinate space according to the location given in the feature. In addition, the main controller will rotate them so that they face the user. The `ARImageFeatureView` will be provided with asset data whenever the `ARMainController` can or wants to do so. In the meantime, it is responsible for displaying a placeholder image.

**ARDimension** Model class with associated classes that as a whole represents a dimension as defined by Gamaray.

**ARPoint3D** Structure that represents a vector in a three-dimensional space. Will have various utility methods associated with it, including:

- `ARPoint3D ARPoint3DCrossProduct(ARPoint3D p1, ARPoint3D p2);`
- `CGFloat ARPoint3DLength(ARPoint3D p);`
- `ARPoint3D ARPoint3DNormalize(ARPoint3D p);`
- `CATransform3D ARPoint3DGetLookAtTransform3D(ARPoint3D origin, ARPoint3D target, ARPoint3D up);`

### C.1.2 Evaluation

In hindsight, we adhered to our initial class diagram very well. Almost only additions have been made to it in comparison with the class diagram in Chapter 4.

The initial class diagram has been a good model of the global structure of the application, while still allowing room for expansion for more specific functionality.

## C.2 Coding conventions

This section describes the coding conventions that are used in this project, which were defined to aid readability and therefore maintainability. The conventions generally follow the conventions used in the Apple documentation. The naming conventions are described in detail on CocoaDevCentral[1].

[1] http://cocoadevcentral.com/articles/000082.php

The example files below define the coding conventions used in this project for the most common constructs.

### C.2.1 Header file

```
//
//  ARCodingConventions.h
//  iBetelgeuse
//
//  Copyright 2010 Finalist IT Group. All rights reserved.
//
//  This file is part of iBetelgeuse.
//
//  iBetelgeuse is free software: you can redistribute it and/or modify
//  it under the terms of the GNU General Public License as published by
//  the Free Software Foundation, either version 3 of the License, or
//  (at your option) any later version.
//
//  iBetelgeuse is distributed in the hope that it will be useful,
//  but WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//  GNU General Public License for more details.
//
//  You should have received a copy of the GNU General Public License
//  along with iBetelgeuse.  If not, see <http://www.gnu.org/licenses/>.
//


#import <Foundation/Foundation.h>


typedef enum {
        ARCodingConventionsTypeLoose,
        ARCodingConventionsTypeStrict,
} ARCodingConventionsType;


@interface ARCodingConventions : NSObject {
@private
        NSString *text;
        ARCodingConventionsType type;
}

@property(nonatomic, copy) NSString *text;

- (void)displayText;
- (void)displayTextWithColor:(UIColor *)aColor;

@end
```

### C.2.2 Implementation file

```
//
//  ARCodingConventions.m
```

```
//  iBetelgeuse
//
//  Copyright 2010 Finalist IT Group. All rights reserved.
//
//  This file is part of iBetelgeuse.
//
//  iBetelgeuse is free software: you can redistribute it and/or modify
//  it under the terms of the GNU General Public License as published by
//  the Free Software Foundation, either version 3 of the License, or
//  (at your option) any later version.
//
//  iBetelgeuse is distributed in the hope that it will be useful,
//  but WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//  GNU General Public License for more details.
//
//  You should have received a copy of the GNU General Public License
//  along with iBetelgeuse.  If not, see <http://www.gnu.org/licenses/>.
//


#import "ARCodingConventions.h"


@interface ARCodingConventions ()

@property(nonatomic, readonly) ARCodingConventionsType type;

@end


@implementation ARCodingConventions

@synthesize text, type;

#pragma mark NSObject

- (void)dealloc {
        [text release];
        [super dealloc];
}

#pragma mark ARCodingConventions

- (void)displayText {
        [self displayTextWithColor:[UIColor redColor]];
}

- (void)displayTextWithColor:(UIColor *)aColor {
        NSAssert(aColor != nil, @"Expected␣non-nil␣color.");

        if ([self text] == nil) {
                // Do something
        } else {
                // Do something else
        }
```

```
        switch ([self type]) {
                case ARCodingConventionsTypeLoose:
                        NSAssert(NO, @"No␣loose␣convention.");
                        break;

                case ARCodingConventionsTypeStrict:
                        // TODO: Implement
                        break;

                default:
                        NSAssert(NO, @"Unreachable␣code.");
                        break;
        }

        for (int i = 0; i < 100; ++i) {
                // :)
        }

//      for (Type var in array) {
//
//      }

        // Infinite loop
        while (YES) {
                do {
                        break;
                } while (NO);
        }
}

@end
```

### C.2.3 Evaluation

The coding conventions were mostly followed in our project. A few small violations exist, regarding different placement of brackets near *else* statements, different usage of whitespace, and usage of *#pragma mark* constructs. Apart from these minor deviations however, the code is consistently formatted.

Not everything was covered in the coding conventions example files. Direct initialization of struct members, for example, was not specified. In these cases, we followed a style similar to the examples shown above. Since these cases were relatively rare, there was no reason for explicitly defining them; using any convention would be fine.

## C.3 Filtering

When our browser supported all required functionality, sensor noise largely influenced the position of features on the screen. It was clear that filtering had to be implemented. Properly filtering the sensor values turned out to be one of the greatest challenges in developing iBetelgeuse.

When using our application, the user wants the objects in the virtual world—projected on the screen—to move and rotate as he moves and rotates the device. Ideally, these objects would exactly match the camera image, but for various reasons this is impossible:

- the sensor values include significant noise,

- the compass is highly biased–easily 25°–and this bias depends on the position and orientation of the device, and
- the camera image is stabilized by an unknown algorithm.

For these reasons, the filter must not only aim to determine the most likely spatial state of the device, but must also ensure a smooth result, so that it looks nice to the user. Additionally, it should respond fast to rotation of the device. Responding fast to position changes is desirable, but not reasonably possible given the frequency and accuracy of GPS samples—the samples are received once per second at best, but do not change between every subsequent sample when walking; the accuracy depends on the situation, but is hardly ever better than 25 meters; normally worse.

### C.3.1 The simple approach

Our first approaches were very simple: since we have high frequency noise and the noise is limited in amplitude, we attempted to use simple filters such as

- an exponential filter that computes a weighted average of the previous output value and the new sample value,
- a double exponential filter, [16]
- a moving average filter that stores the *n* most recent samples and computes their average value, and
- a moving median filter that stores the *n* most recent samples and computes their median value.
- a *step filter* that returns a constant value that is set to a new sample value when the difference between the new sample value and the previous output value is above a threshold value.

None of these filters worked as desired. Most filters indeed reduced noise, but at the cost of response time. To achieve acceptable stability, most filters were to be configured so that it would take a few seconds for the new value to update. The step filter showed very nice stability, but did not move smoothly when rotating the device. After trying various combinations and modifications of these filters, we concluded that we needed more advanced filtering to achieve acceptable results.

### C.3.2 Filtering techniques used by other browsers

Since our AR browser is not the first of its kind, we looked at the filtering that other browsers implement, and the how well they performed on stability and response time. We mainly investigated Layar—because of its popularity—and Gamaray—because we have its source code and it is the Android version of our browser.

Although we do not have Layar's source code, we can see that Layar implements some kind of lawpass filtering, possibly moving weighted average or exponential filtering. The objects that are displayed by Layer are pretty stable on the screen, but the response time is rather bad. If rotating quickly, it can take over three seconds for the objects to be relocated. Although this may be acceptable, we wanted something better.

Gamaray implements some kind of moving average filter: whenever the spatial state changes, it computes a transformation matrix representing the current orientation. It then applies a moving average on each field of the matrix. This seems to be incorrect, especially since the matrix is not renormalized after averaging, but apparently works well enough for Gamaray. We were however not convinced by the performance of Gamaray's filtering, which reaffirmed our conclusion that more advanced filtering was desired.

### C.3.3 Investigating Kalman filters

When investigating possible filters, we found various references towards (variations of) Kalman filters. [17, 23] A Kalman filter basically approximates the current state of the system given a set of measurements, a model of the system, and various parameters describing the noise. [22] Nice properties of this filter are that it guarantees to return the value that is the best possible estimate of the true value—assuming the model is not simplified and the parameters are correct—and that it can fuse data given by multiple sensors into one vector representing the current state.

Existing models and Kalman filter implementations for orientation determination exist, but these often assume that a gyroscope is available, or multiple acceleration sensors are available to approximate the angular rate. [17, 23] The version of the iPhone for which this application was developed does not have a gyroscope and only one acceleration sensor, so directly implementing the solutions presented in those papers is not possible. We attempted to adapt these filters by estimating the value of the gyroscope based on other sensor data, but this did not lead to any good results; this may be due to our limited understanding of creating and configuring Kalman filters. We likely used an incorrect model or invalid parameters, or maybe Kalman filtering is not applicable at all due to the missing sensor.

Eventually, we decided that we were not comfortable enough with Kalman filtering to implement it within a reasonable timeframe. The effort was not wasted however, we learnt at least two ideas that were useful for implementing our final filter:

- defining the underlying model to which state updates adhere and respecting this model in the filter, and
- using *quaternions* to represent orientation rather than the combination of up- and North vector.

## C.4 Accelerometer calibration

During testing of our browser as well as other augmented reality browsers, we observed that the virtual horizon was never straight (about $1^o$ or $2^o$ off) when the real horizon was straight relative to the device. The sensor that is used for determining the pitch and roll of the device is the accelerometer, which measures the direction of the Earth's field of gravity. This can be done reasonably precisely, assuming the user is holding the device steady. (This is unlike the magnetometer, which is affected by interference from all kinds of sources.) Therefore we think the accelerometer has enough precision to be calibrated.

The iPhone 3GS contains an ST Microelectronics LIS331DLx 3-axis accelerometer [2]. According to the data sheet of this sensor, it itself is factory-calibrated [5]. Therefore, we believe the reason that the virtual horizon is not straight is that the accelerometer is not put perfectly straight in the iPhone. We think this is even to be expected.

Therefore we attempted to find a method for calibrating the accelerometer in relation to the device. Since we could not find much information about this subject we tried to develop our own method. The method would need to work for all axes, since—unlike a simple bubble level application which uses only two—we make use of all three axes for our application. Also, the method needed to be easy for the user to perform and therefore would not need to require any level surfaces or perfect right-angled corners.

We tried to do this by making use of the fact that a certain bias in an axis will become apparent when making two measurements: one in any orientation, and the other when rotated exactly $180^o$ orthogonal to the axis. After some calculations in MATLAB and careful thinking we concluded that it is not possible to find a calibration method that satisfies the above criteria. For all axes to be calibrated we would need at least three measurements. These measurements cannot be all done in the same plane, requiring a right-angled corner for the user to hold the device against. Even if we would ignore this criterium, we didn't know how to easily solve the three matrix equations that result from the three measurements.

In the end we decided that the effort required does not outweigh the benefits, so we put the idea of calibrating the accelerometer in the ice box.

## C.5 Representing rotations as quaternions

A quaternion is a hypercomplex number $w + xi + yj + zk$. Quaternions can, amongst other things, be used for representing and manipulating 3D rotation and orientation in an elegant way. Their exact mathematical properties are not easy to explain, and are outside the scope of this report. Fortunately, many ready-made quaternion functions and libraries can be found, so only a basic understanding of quaternions is necessary.

A quaternion used to represent a rotation in 3D can be best interpreted as a representation of a rotation axis, and an angle of rotation around this axis. These values are scaled and then stored in the quaternion. A rotation quaternion is always of unit length. That is, the sum of the squared coordinates must equal 1.

Due to their encoding, some important mathematical properties hold, and quaternion algebra can be used to manipulate the quaternions. For example, the quaternion can be applied to a point, much like a transformation matrix, and it can be combined with other quaternions to get a new quaternion representing the combined rotation.

It is important to note that due to the axis-angle based encoding, a quaternion $q$ represents exactly the same rotation as $-q$ (both the axis and angle are inverted, resulting in the same rotation). This property makes some quaternion operations, such as averaging multiple quaternions, much less trivial. The algorithm must first ensure that the quaternions 'point in the same direction', before averaging, but there may be no way to do that for some combinations of three or more quaternions. [15]

The reason for using quaternions is that they have no singularities: two quaternions that represent similar rotations have similar elements (or the first quaternion's elements are similar to the negation of the second quaternion's elements, as described above). This property is required for smooth interpolation between two rotations. [13]

### C.5.1 Estimating angular velocity

For a quaternion $q$, the derivative $\frac{dq}{dt}$ is not very meaningful, and is not linear when $q$ is changing due to a constant angular velocity. Instead, the angular velocity can be determined and used. For a rotation quaternion $q$ and a angular velocity $\omega$ (expressed as a quaternion with $w = 0$ and $x$, $y$, and $z$ representing the rotations about each axis; this is *not* a rotation quaternion), the following expression holds: $\frac{dq}{dt} = \frac{1}{2}q\omega$. [12] Rewriting that gives $\omega = 2q^{-1}\frac{dq}{dt}$.

By estimating $\frac{dq}{dt} \simeq \frac{q_n - q_{n-1}}{t_n - t_{n-1}}$ and using the formula above, an estimate for the angular velocity can be computed from a series of samples.

### C.5.2 Spherical moving average

A moving average filter is a commonly used filter for scalar values, however if rotation quaternions are filtered, simply taking the moving average for each component is not correct. Because for any rotation quaternion, its negation represents the same rotation, simply averaging two similar quaternions that are expressed differently may lead to incorrect results. Additionally, the spherical distance between two points is not equal to the linear distance, which may cause the spherical average to differ slightly from the (normalized) linear average.

For computing the spherical weighted average rotation quaternion, we use an algorithm by Samuel R. Buss and Jay P. Fillmore. [15] In short, this iterative algorithm computes an initial estimate of the spherical average by assuming it is in the direction of the axis with the largest average absolute value. It negates quaternions that make an angle of more than 90° with the estimate, and approximates the weighted average by computing the average of the spherical distance to each point, and rotating the estimate in that direction. If the estimate changes less than a threshold value, the approximated spherical average is returned.

# Appendix D

# Evaluation

Looking back at the project, we are very pleased with almost every aspect of its process and results. Naturally, during the project there were many unforeseen problems, interesting experiences, and new ideas. This appendix evaluates how this project went, by discussing our planning, the difficulty of the project, and our learning experiences.

Before this project, there already was a working Android implementation and a protocol definition that our browser was to support. Although both of these were of low quality, they did clearly define the scope of our project and its functionality. We therefore found it easy to determine the user stories that define the application's functionality.

We started this project with about three weeks (about ten business days due to holidays) of orientation ('sprint zero'). Although this seemed like a long time initially, it helped us to research various things. We stuck to the three weeks pretty closely, and thought of more things to research because more time was available than was strictly necessary to write the action plan and research report. The time we spent doing this additional research proved to be very useful later on, as we had already identified (and sometimes had solutions to) most of the problems that have occured, such as orientation determination and sensor filtering.

Because the goal of the project was clear, and our research indicated how easy or hard various parts of the application would be to implement, we could estimate the time required for implementing each of the user stories reasonably well. Consequentially, the schedule for the first implementation sprint was spot on. Pivotal Tracker helped us to keep track of the tasks that were to be done, which worked very well for us: we always had a clear idea of what was done and what was not.

We managed our project using Scrum. Working in short iterations and not making a final and hugely detailed design at the start of the project suited us well. Although we feel the project went well, partially due to Scrum, we do not feel we used Scrum's full potential. First, it happened to be hard to schedule daily stand up meetings with the product owners to discuss the progress. The initial plan was to do this at least a few times per week, but unfortunately this never worked out. Second, the transitions between sprint 2 and 3 and sprint 3 and the final sprint were a little fuzzy: we were already working on the next sprint's functionality or we were still working on functionality of the previous sprint. Although we had a taste of Scrum, we would have liked to see more of it.

Applying test driven development worked well at first, when we knew best what had to be done and how. Further along in the project, we often neglected to test new methods we implemented immediately, and implemented tests in the last few weeks instead. On the one hand, this was a bad thing because bugs were sometimes found later than desired. On the other hand, we wrote a lot of code that was later removed because we implemented it differently. If we would have written unit tests for each of these attempts, we would likely not have had enough time left to implement filtering that works as well as it currently does.

This project had many difficult aspects. Most difficulties were mathematical difficulties: rendering the virtual world and determining orientation required a significant amount of linear algebra, and sensor

filtering required both signal processing and probability analysis. We studied various websites and papers when working on sensor filtering; especially about the workings and usage of quaternions.

Although the mathematical side was the most challenging aspect, there were some challenges to properly designing the software architecture as well. For example, defining the interfaces of the range of filter classes was challenging. We also had some trouble designing the asset loading infrastructure, because it was tough to determine whether the asset, the asset manager, the feature or overlay, or the main controller was responsible for controlling the loading of data.

Finally, we found that the working environment at Finalist was a very pleasant one. Although working full-time left us with much less free time than we were used to, we never felt that the days were too long; on the contrary, we were often surprised that it was already time to go home.

In this project, we gained experience with working in a company, we improved our software design skills by working together and discussing about possible solutions to design issues, we refreshed and extended our mathematical skills, and finally practiced documenting and reporting the things we have done.

# Glossary

Note: the definitions below are not generally accurate and should only be used in the scope of this project.

**accelerometer** Measures the acceleration imposed on it. In particular, it can determine the direction of Earth's gravitational pull.

**Android** An operating system for mobile devices made by Google.

**App Store** Part of the Apple iTunes Store, the only official place where iPhone applications can be distributed to users.

**ARena** An project done by Finalist that uses augmented reality for educational purposes.

**asset** Data used by a *feature* or *overlay*, such as an image.

**augmented reality** The reality created by an *augmented reality browser* by augmenting the real world with a *dimension*.

**augmented reality browser** Software that runs on an advanced mobile phone that allows the user to browse through an *augmented reality*. This is done by projecting a *dimension* onto a live camera image, depending on the device's *spatial state*.

**Betelgeuse Designer** Web-based application that can be used to make and publish a *dimension*.

**DDF** Dimension Description File, file format describing a *dimension*.

**dimension** The virtual world and its contents, often bounded to a certain area surrounding a particular location. The most important elements of a dimension are *features* and *overlays*.

**ECEF** Earth-Centered, Earth-Fixed, a global Cartesian coordinate system in metric units whose origin is in the center of Earth.

**ENU** East, North, Up, a local Cartesian coordinate system in metric units whose origin is at a particular location on Earth.

**feature** An object in the virtual world with a three-dimensional location (in practice, a *WGS84* location). May use *assets*. Examples include: labels that mark points of interest in a city, photos taken at a certain location.

**Gamaray** An *augmented reality browser* for the *Android* platform.

**gcov** Part of the GNU Compiler Collection, a code coverage tool.

**GDDF** Gamaray Dimension Description File, see *DDF*.

**GitHub** Web-based hosting service and community for both commercial and open source projects that use the Git version control system.

**GH-Unit** Unit testing framework for *Objective-C*.

**GPL** GNU General Public License, a free 'copyleft' license for software and other kinds of works.

**gyroscope** Measures angular velocity imposed on it.

**iBetelgeuse** Pronounced 'iBeetlejuice', the *augmented reality browser* for the *iPhone* platform that is the subject of this report.

**iPhone** Product line of advanced mobile phones manufactured and sold by Apple.

**iPhone 3GS** Third generation model of the *iPhone* product line.

**iPhone 4** Fourth generation model of the *iPhone* product line. Unlike the *iPhone 3GS*, it has a *gyroscope*.

**Layar** Leading commercial *augmented reality browser*.

**magnetometer** Measures the magnetic flux density. In particular, it can determine the direction of Earth's magnetic North.

**Objective-C** Object-oriented extension to the C programming language with a dynamic runtime. The de-facto language for developing *iPhone* applications.

**overlay** An object displayed on screen with a two-dimensional location (a screen coordinate). May use *assets*. Examples include: the logo of the author of the dimension, a label indicating the score of a game.

**Pivotal Tracker** Web-based project management tool that uses *Scrum*-principles.

**QR code** Quick Response code, a two-dimensional barcode that can contain various kinds of data, such as URLs.

**quaternion** Number system that extends the system of complex numbers. Can be used to represent three-dimensional rotations.

**Scrum** Framework for agile software development and project management.

**spatial state** The location and orientation of an object in relation to the real world.

**WGS84** World Geodetic System 84, standard reference ellipsoid for Earth that is currently being used by GPS.

**Xcode** The development environment provided by Apple to develop *iPhone* applications.

# Bibliography


[1] Agile software development. http://en.wikipedia.org/wiki/Agile_software_development. Retrieved June 29, 2010.

[2] Apple iPhone 3GS 16 GB - What's Inside. http://www2.electronicproducts.com/Apple_iPhone_3GS_16GB-whatsinside_text-82.aspx. Retrieved June 29, 2010.

[3] Dimension description files. http://finalist.github.com/iBetelgeuse/#ddf. Retrieved June 30, 2010.

[4] Geldautomaten zoeken (Netherlands). http://site.layar.com/geldautomaten-zoeken-netherlands/. Retrieved May 6, 2010.

[5] LIS3LV02DL datasheet. http://www.datasheetarchive.com/LIS3LV02DL-datasheet.html. Retrieved June 29, 2010.

[6] Other Earth based coordinate systems. http://en.wikipedia.org/wiki/Geodetic_system#Other_Earth_based_coordinate_systems. Retrieved June 30, 2010.

[7] Scrum. http://en.wikipedia.org/wiki/Scrum_(development). Retrieved June 29, 2010.

[8] Spaceinvaders 3D (International). http://site.layar.com/spaceinvaders-international/. Retrieved May 6, 2010.

[9] W3C workshop: Augmented reality on the web. http://www.w3.org/2010/06/w3car/report.html. Retrieved July 4, 2010.

[10] ZBar. http://zbar.sourceforge.net/. Retrieved July 2, 2010.

[11] Apple, Inc. UIAcceleration class reference. http://developer.apple.com/iphone/library/documentation/uikit/reference/UIAcceleration_Class/Reference/UIAcceleration.html. Retrieved July 5, 2010.

[12] Martin Baker. Quaternion calculus. http://www.euclideanspace.com/maths/differential/other/quaternioncalculus/index.htm. Retrieved June 29, 2010.

[13] Martin Baker. Quaternion interpolation (SLERP). http://www.euclideanspace.com/maths/algebra/realNormedAlgebra/quaternions/slerp/index.htm. Retrieved June 29, 2010.

[14] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development. http://www.agilemanifesto.org/. Retrieved June 29, 2010.

[15] Samuel R. Buss and Jay P. Fillmore. Spherical averages and applications to spherical splines and interpolation. *ACM Transactions on Graphics*, 20:95–126, 2001.

[16] Joseph J. LaViola. Double exponential smoothing: an alternative to Kalman filter-based predictive tracking. In *EGVE '03: Proceedings of the workshop on Virtual environments 2003*, pages 199–206, New York, NY, USA, 2003. ACM.

[17] Joa Luis Marins, Xiaoping Yun, Eric R. Bachmann, Robert McGhee, and Michael J. Zyda. An extended Kalman filter for quaternion-based orientation estimation using MARG sensors. In *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, pages 2003–2011, 2001.

[18] Oliver Montenbruck and Eberhard Gill. State interpolation for on-board navigation systems. *Aerospace Science and Technology*, 5(3):209 – 220, 2001.

[19] Dennis Stevense and Peter van der Tak. Opdrachtomschrijving.

[20] Dennis Stevense and Peter van der Tak. Plan van aanpak.

[21] Dennis Stevense and Peter van der Tak. Research report.

[22] Greg Welch and Gary Bishop. An introduction to the Kalman filter. Technical report, Chapel Hill, NC, USA, 1995.

[23] Yun Xiaoping, E.R. Bachmann, and R.B. McGhee. A simplified quaternion-based algorithm for orientation estimation from earth gravity and magnetic field measurements. *Instrumentation and Measurement, IEEE Transactions on*, 57(3):638 –650, march 2008.

# Opdrachtomschrijving

Dennis Stevense (1358448) en Peter van der Tak (1358464)
Bachelorproject (IN3405)
TU Delft

15 april 2010

**Bedrijf** Finalist IT Group[1]

**Locatie** Rotterdam

**Contactpersoon** Okke van 't Verlaat (okke@finalist.com)

**Mentoren** Okke van 't Verlaat (okke@finalist.com) en Jacques Bouman (jacques@finalist.com)

**Periode** 26 april t/m 9 juli

# 1 Inleiding

## 1.1 Bedrijf

"Finalist IT Group verleent de IT diensten consultancy, projecten en applicatiebeheer. Wij zijn vooral gespecialiseerd in open source en open standaarden. Finalist werd in 1994 opgericht. In Nederland werkt Finalist met 70 collega's verdeeld over kantoren in Rotterdam, Amsterdam en Eindhoven. Finalist is tevens gevestigd in New York (V.S.) en Beijing (China)."[2]

## 1.2 Motivatie

Gamaray[3] is een open source augmented reality browser, gemaakt in Java, die geschikt is voor het Android-platform. Door concurrentie is de ontwikkeling van de commerciële versie van deze browser gestopt en is er een open source project[4] uit voortgekomen.

Met behulp van Gamaray heeft Finalist een pilotproject voor stichting Kennisnet uitgevoerd. In deze pilot is software ontwikkeld waarmee docenten in een Google Maps-omgeving routes door een stad kunnen definiëren. Op bepaalde locaties langs de route kan een vraag, notitie of afbeelding in de ruimte worden geplaatst. Leerlingen kunnen met behulp van de Gamaray browser op een Android-telefoon deze route lopen en onderweg de vragen beantwoorden, notities lezen en afbeeldingen bekijken.

Zowel Finalist als Kennisnet hecht belang aan ondersteuning voor andere mobiele platformen, waarvan het iPhone-platform de belangrijkste is.

# 2 Opdracht

## 2.1 Omschrijving

Het doel van de stageopdracht is om een prototype AR browser voor het iPhone-platform te ontwerpen en te ontwikkelen die aansluit op de hierboven beschreven pilot. Deze browser zal communiceren met

[1] http://www.finalist.com/
[2] http://www.finalist.nl/over_finalist/missie_en_kernwaarden
[3] http://www.gamaray.com/
[4] http://sourceforge.net/projects/opengamaray/

de bestaande backend door middel van dezelfde API die nu door Gamaray geïmplementeerd wordt. Om het ontwikkelproces te versnellen kan de broncode van de Gamaray AR browser gebruikt worden als 'inspiratie'. Finalist is voornemens om de nieuwe browser—net als Gamaray—als open source software beschikbaar te stellen.

## 2.2 Inrichting

De opdracht zal met de Scrum-methode uitgevoerd worden:

1. Een 0-sprint van twee weken, die overeenkomt met de oriëntatiefase van het project. In deze fase wordt de Scrum product backlog opgesteld; anders dan in Scrum zijn de stagiaires hiervoor verantwoordelijk. Daarnaast wordt in deze fase een plan van aanpak en een oriëntatieverslag gemaakt, en wordt de ontwikkelomgeving opgezet.
2. Drie sprints van elk twee weken, wat overeenkomt met de ontwikkelfase van het project. In de deze fase wordt het prototype geïmplementeerd en getest.
3. Een eindsprint van twee weken, die overeenkomt met de afrondingsfase van het project. In deze fase worden 'puntjes op de i gezet' en een eindverslag en -presentatie gemaakt.

De Scrum-methode zal worden toegepast zoals die gebruikelijk is bij Finalist. Okke en Jacques zullen de rollen aannemen van Scrummaster en Product Owner. Dagelijks zal een standup meeting worden gehouden met alle teamleden waarin de voortgang van het project wordt besproken.

Voor versiebeheer zal gebruik gemaakt worden van Subversion. Voor zover mogelijk zal gebruik gemaakt worden van de Hudson server voor continuous integration.

## 2.3 Deliverables

De volgende producten moeten worden afgeleverd:

1. Een plan van aanpak volgens de eisen van de TU.
2. Een oriëntatieverslag volgens de eisen van de TU en met de volgende inhoud:
   (a) onderzoek naar bestaande augmented reality browsers,
   (b) onderzoek naar de API en implementatie van Gamaray,
   (c) onderzoek naar de theorie en algoritmen die nodig is bij het implementeren van AR, en
   (d) onderzoek naar de software en hardware van het iPhone-platform.
3. Een prototype van de AR browser voor het iPhone-platform dat moet voldoen aan de volgende eisen:
   (a) de browser moet een looproute kunnen ophalen en tonen, inclusief de bijbehorende objecten zoals vragen en afbeeldingen,
   (b) de browser moet informatie, zoals de locatie van de gebruiker en antwoorden op vragen, kunnen terugsturen naar de server,
   (c) de code moet voldoende gedocumenteerd zijn,
   (d) bijzondere aspecten van de software moet gedocumenteerd zijn op de interne wiki van Finalist, en
   (e) de code moet getest zijn met unit tests, waarbij gestreefd moet worden naar een zo groot mogelijke test coverage.
4. De product backlog.
5. Een eindverslag volgens de eisen van de TU.

   Finalist heeft aangegeven dat het protocol dat de AR browser en de backend gebruiken om met elkaar te communiceren voor verbetering vatbaar is. Eén ding dat in het eindverslag moet komen is een aanbeveling voor een verbeterde standaard voor deze communicatie.

# Plan van aanpak

Dennis Stevense (1358448) en Peter van der Tak (1358464)
Bachelorproject (IN3405)
TU Delft

7 mei 2010

## 1 Inleiding

Dit document vormt een plan van aanpak voor het bachelorproject van Dennis Stevense en Peter van der Tak bij Finalist IT Group. Omdat Dennis reeds contact had met Finalist en er op korte termijn een stageopdracht nodig was voor het bachelorproject is bekeken wat er mogelijk is bij Finalist. Zij hebben de wens om een bestaande augmented reality browser voor Android ook op het iPhone-platform beschikbaar te maken. Aangezien er binnen Finalist geen mensen zijn die ervaring hebben met iPhone-ontwikkeling was dit een voor beide partijen geschikte stageopdracht.

Dit plan van aanpak legt vast welk probleem opgelost moet worden en op welke manier dit moet gebeuren.

Hoofdstuk 2 geeft achtergrondinformatie over Finalist en de aanleiding van de opdracht. Hoofdstuk 3 omschrijft de opdracht in detail en wat de op te leveren producten en eisen zijn. De manier waarop de opdracht wordt uitgevoerd is beschreven in hoofdstuk 4. Hoofdstuk 5 gaat in op een aantal praktische zaken rond het project en tot slot beschouwt hoofdstuk 6 kort de kwaliteitswaarborging.

## 2 Achtergrond

Dit hoofdstuk zal kort toelichten wat Finalist doet en wat de aanleiding is van de opdracht.

### 2.1 Bedrijf

Finalist IT Group[1] is een IT-bedrijf dat werd opgericht in 1994 en is gespecialiseerd in open source en open standaarden. Er werken in totaal 70 mensen bij Finalist verdeeld over kantoren in Rotterdam, Amsterdam en Eindhoven. Daarnaast zijn er kantoren in New York en Beijing. Finalist doet consultancy, applicatiebeheer en voert maatwerk projecten uit.

### 2.2 Aanleiding

Finalist heeft een pilotproject voor stichting Kennisnet uitgevoerd. In deze pilot is software ontwikkeld waarmee docenten in een Google Maps-omgeving routes door een stad kunnen definiëren. Op bepaalde locaties langs de route kan een vraag, notitie of afbeelding in de ruimte worden geplaatst. Leerlingen kunnen met behulp van een augmented reality browser op een smartphone deze route lopen en onderweg de vragen beantwoorden, notities lezen en afbeeldingen bekijken.

Als augmented reality browser wordt Gamaray gebruikt. Gamaray[2] is een open source augmented reality browser, gemaakt in Java, die geschikt is voor het Android-platform. Door concurrentie is de ontwikkeling van de commerciële versie van deze browser gestopt en is er een open source project[3] uit voortgekomen.

[1] http://www.finalist.nl/
[2] http://www.gamaray.com/
[3] http://sourceforge.net/projects/opengamaray/

# 3 Opdrachtomschrijving

In dit hoofdstuk wordt beschreven wat het probleem is, welke oplossing wij daarvoor zullen implementeren, waaraan het eindproduct moet voldoen en welke randvoorwaarden er zijn.

## 3.1 Probleemstelling

Finalist heeft de Gamaray browser gebruikt voor het realiseren van de pilot zoals beschreven in de inleiding; deze browser is alleen op Android smartphones te gebruiken. Zowel Finalist als stichting Kennisnet hechten belang aan ondersteuning van andere platformen, waarvan het iPhone-platform de belangrijkste is. Er is op dit moment geen augmented reality (AR) browser beschikbaar voor de iPhone die compatibel is met de Gamaray backend.

## 3.2 Doelstelling

Het doel van de stageopdracht is om een (prototype) AR browser voor het iPhone-platform te ontwerpen en te ontwikkelen die aansluit op de pilot die is uitgevoerd in opdracht van stichting Kennisnet. Deze browser zal communiceren met de bestaande backend door middel van dezelfde API die nu door Gamaray geïmplementeerd wordt. Om het ontwikkelproces te versnellen kan de broncode van de Gamaray AR browser gebruikt worden als 'inspiratie'. Finalist is voornemens om de nieuwe browser—net als Gamaray—als open source software beschikbaar te stellen.

## 3.3 Deliverables

Nu beschrijven we de op te leveren producten en de eisen en beperkingen die daaraan verbonden zijn:

1. Een plan van aanpak volgens de eisen van de TU.
2. Een oriëntatieverslag volgens de eisen van de TU en met de volgende inhoud:
   (a) onderzoek naar bestaande augmented reality browsers,
   (b) onderzoek naar de API en implementatie van Gamaray,
   (c) onderzoek naar de theorie en algoritmen die nodig is bij het implementeren van AR, en
   (d) onderzoek naar de software en hardware van het iPhone-platform.
3. De product backlog. (Zie paragraaf 4.1.)
4. Een prototype van de AR browser voor het iPhone-platform dat moet voldoen aan de volgende eisen:
   (a) de browser moet vragen, notities en afbeeldingen kunnen ophalen van de bestaande backend en tonen over een live camerabeeld,
   (b) de browser moet informatie, zoals de locatie van de gebruiker en antwoorden op vragen, kunnen terugsturen naar de bestaande backend,
   (c) de code moet voldoende gedocumenteerd zijn,
   (d) bijzondere aspecten van de software moeten gedocumenteerd zijn op de interne wiki van Finalist, en
   (e) de code moet getest zijn met unit tests, waarbij gestreefd moet worden naar een zo groot mogelijke test coverage.
5. Een eindverslag volgens de eisen van de TU.

   Finalist heeft aangegeven dat het protocol dat de AR browser en de backend gebruiken om met elkaar te communiceren voor verbetering vatbaar is. Eén ding dat in het eindverslag moet komen is een aanbeveling voor een verbeterde standaard voor deze communicatie.

# 4 Aanpak

Dit hoofdstuk beschrijft op welke manier de opdracht uitgevoerd moet worden.

## 4.1 Methodiek

De opdracht zal met de Scrum-methode uitgevoerd worden. Deze methodiek wordt uitvoerig door Finalist toegepast. Eén van de voordelen van Scrum is dat er met hogere zekerheid meermalen een werkend product wordt opgevelerd. Daarnaast is het een mogelijkheid voor ons om deze methodiek te ervaren.

De periode wordt verdeeld in een vijftal sprints van ieder twee weken. (Zie ook paragraaf 4.4.) Okke en Jacques zullen samen de rollen aannemen van zowel Scrummaster als Product Owner. Tijdens de ontwikkelfase zal dagelijks zal een standup meeting worden gehouden met alle teamleden waarin de voortgang van het project wordt besproken.

Verder zal worden gedaan aan test-driven development om een zo hoog mogelijke coverage te bereiken.

## 4.2 Werkzaamheden

**bepalen van eisen** Het bepalen van de eisen zal hoofdzakelijk worden gedaan tijdens de 0-sprint. Door het maken van de Scrum product backlog worden namelijk functionele en niet-functionele eisen bepaald.

**ontwerpen** Hoewel er geen uitgebreide ontwerpdocumenten zullen worden opgeleverd is er tijdens de oriëntatiefase en de ontwikkelsprints ruimte voor het specificeren en ontwerpen van de te ontwikkelen functionaliteit.

**testen** Omdat er gebruikt zal worden gemaakt test-driven development vindt er continu testing plaats tijdens de ontwikkelingsfase.

## 4.3 Technieken

**documenten** Documenten zullen worden gemaakt met LaTeX voor makkelijke samenwerking en revisering.

**ontwikkeling** Ontwikkeling zal plaatsvinden met Xcode, de IDE die gratis door Apple beschikbaar wordt gesteld voor onder andere iPhone-ontwikkeling.

**testen** Bij Xcode wordt een unit testing framework geleverd. Daarnaast kan *gcov* van GCC gebruikt worden voor coverage reporting.

**versiebeheer** Voor versiebeheer zal gebruik worden gemaakt van Git. Omdat het product open source is zal code op een publieke repository worden gezet.

## 4.4 Planning

De looptijd van het project is 26 april tot en met 9 juli 2010. Deze periode van 10 weken (excl. feestdagen) wordt als volgt ingedeeld:

**26 april—12 mei** De 0-sprint; de oriëntatiefase van het project. Aan het einde van deze periode moet het plan van aanpak, het oriëntatieverslag en een eerste iteratie van het product backlog af zijn. Ook moet de ontwikkelomgeving e.d. klaar zijn voor gebruik.

**14 mei—28 mei, 31 mei—11 juni, 14 juni—25 juni** Drie ontwikkelsprints. Aan het einde van elke sprint moet er een werkend product zijn.

**28 juni—9 juli** De eindsprint; de afrondingsfase van het project. Hierin is ruimte voor het polijsten van het product en het maken van een eindverslag en -presentatie. Aan het einde van deze periode moeten alle deliverables af zijn.

# 5 Projectinrichting

Dit hoofdstuk beschrijft welke partijen betrokken zijn bij het project, waar en wanneer het project wordt uitgevoerd, welke middelen beschikbaar zijn voor het uitvoeren van het project en hoe de communicatie met de opdrachtgever en begeleiders verloopt.

## 5.1 Betrokkenen

Naast de stagiaires—Dennis Stevense en Peter van der Tak—zijn de begeleiders binnen Finalist en de TU Delft betrokken bij het project. Okke van 't Verlaat[4] en Jacques Bouman[5] zullen de begeleiding binnen Finalist verzorgen en daarnaast als opdrachtgever fungeren. Hans-Gerhard Gross[6] zal verantwoordelijk zijn voor de begeleiding vanuit de TU Delft.

## 5.2 Locatie

De stageopdracht zal worden uitgevoerd bij Finalist IT Group in het kantoor in Rotterdam, waar de stagiaires fulltime aan het project zullen werken.

## 5.3 Communicatie

In de oriëntatiefase is er informeel contact met de opdrachtgever. De voortgang van de oriëntatiefase wordt via een weblog gecommuniceerd naar de begeleider bij de TU Delft en de opdrachtgever.

Na de oriëntatiefase zal er dagelijks contact zijn met de opdrachtgever—in de Scrum stand up meetings—waarin de bezigheden worden besproken en eventuele problemen worden aangekaart.

## 5.4 Faciliteiten

Finalist stelt een MacBook beschikbaar voor Peter van der Tak om de applicatie op te ontwikkelen. Dennis Stevense zal zijn eigen MacBook Pro gebruiken. Daarnaast zorgt Finalist voor een publieke Git repository en een iPhone en benodigdheden (zoals een Apple iPhone Developer account) om de applicatie te kunnen testen.

De TU Delft stelt een weblog[7] ter beschikking die kan worden gebruikt voor het bijhouden van de voortgang in de orientatiefase.

# 6 Kwaliteitswaarborging

In dit hoofdstuk lichten we kort toe hoe het eerder genoemde bepaalde kwaliteiten waarborgt.

**bruikbaarheid** Door het tussentijds opleveren van werkende productiteraties kan de bruikbaarheid gedurende het project worden gecontroleerd.

**betrouwbaarheid** Door het uitgebreide testen van de applicatie door de toepassing van test-driven development wordt de betrouwbaarheid van de applicatie verhoogd.

**onderhoudbaarheid** De onderhoudbaarheid wordt verhoogd door de vereiste documentatie in de code en de aanwezigheid van tests.

**proces** De Scrum-methodiek zorgt voor een iteratief proces met kortere doorlooptijden waardoor hopelijk effectiever kan worden gewerkt.

[4] okke@finalist.com
[5] jacques@finalist.com
[6] h.g.gross@tudelft.nl
[7] http://ar.weblog.tudelft.nl/

# Research Report

Dennis Stevense (1358448) en Peter van der Tak (1358464)
Bachelorproject (IN3405)
TU Delft

May 14, 2010

## Contents



# 1 Introduction

This report is part of the Bachelor project by Dennis Stevense and Peter van der Tak for Finalist IT Group, and describes the results of our research on augmented reality browsers and the iPhone. This research is meant as orientation on augmented reality browsers and the iPhone, with the goal of preparing ourselves for the implementation of our own augmented reality browser on the iPhone.

The structure of this report is as follows: section 2 introduces the term 'augmented reality' and explains its features and concepts. Section 3 shows a survey of existing augmented reality browsers. The next section, section 4, describes how the current Gamaray browser works, focusing on the aspects that will be important when implementating our own browser. Section 5 briefly discusses the desired use of

Gamaray by Finalist. Section 6 addresses the theory that is necessary for implementing our own augmented reality browser. Finally, section 7 describes how the iPhone platform affects the implementation of our augmented reality browser.

# 2 Augmented reality

This section describes the fundamental concepts of augmented reality and augmented reality browsers, and how these browsers can be applied.

In augmented reality, the real world is augmented with artificial data by means of an augmented reality device, in other words, these devices combine actual reality and virtual reality. Augmented reality devices include smartphones, which are the kind of devices that we will be focusing on in this project, but also include other devices such as projectors or head mounted displays.

In this project, we will be focusing on applications for smartphones that implement augmented reality. Such applications typically use the camera to capture the real world, and the GPS, compass, and accelerometer to determine the current position and orientation. The application captures camera pictures in real time and draws markers on top of the picture to indicate points of interest; depending on the application, these markers are simple 2D icons, or complex 3D models. The user can now look 'through' the smartphone to see the augmented world, and move the smartphone to look around. In most browser, the user can view detailed information on a particular point of interest by selecting it on the screen or by physically moving towards it.

Augmented reality browsers usually communicate with a backend to fetch the points of interest. The application determines the current position and sends a query to a backend to determine which points of interest are in the neighborhood. The backend sends the resulting points of interest to the browser, which can then project these on the screen.

Augmented reality applications for smartphones can be applied for both practical and entertaining purposes. Existing applications range from finding ATMs [7] to projecting alien space ships in the sky [15].

# 3 Mobile augmented reality browsers

This section will take a look at existing implementations of augmented reality browsers. The most well known augmented reality browsers for smartphones are Layar and Wikitude, which we will consider first. Then we will take a short look at some other mobile browsers available.

## 3.1 Layar

Layar [10] is arguably the most popular mobile augmented reality browser. It was first released on June 19th, 2009 by SPRXmobile, a company founded in the Netherlands. Layar claims to be the first of its kind [14]. The Apple iPhone 3GS and all devices running the Google Android operating system with a camera, GPS and compass are supported.

Layar relies on layers made by Layar itself as well as by third parties. These layers are reviewed and published solely by Layar so that they appear within the browser. A layer is a static or dynamic set of points of interest that are loaded from a webservice. For example, there could be a layer containing the locations of post officies or a layer containing geotagged photos from Flickr.

After starting the application, the user can find and pick a layer. The application then displays the phone's camera image and determines the device's location and orientation. Using this information, points of interest within a certain radius are fetched from a server and displayed on top of the camera image. They are also shown in a radar-type view. See figure 1 on the following page. When the device's location changes significantly, new points of interest are retrieved.

Layers can have settings, for example for the user to be able to limit the returned points of interest to a certain keyword. Points of interest can be represented by customizable icons or simple 3D models. In addition, points of interest can have audio assigned to them.


Figure 1: Layar [4]

## 3.2 Wikitude

Wikitude [16] calls itself the “No. 1 Augmented Reality Browser” [18] and is a product of Mobilizy, a company located in Austria. The first news and videos about Wikitude [17] are dated October 2008, so apparently that makes Wikitude, not Layar, the first mobile augmented reality browser around.

The full name of the browser is Wikitude World Browser. As of version 4, Wikitude works almost exactly the same as Layar; layers are referred to as worlds. See figure 2 on the next page. One exception is that users can view more than one world at a time. The points of interest of the worlds are displayed concurrently.

Prior to version 4, Wikitude only had a single world of points of interest. These locations were supplied and maintained by the community, in a similar way to Wikipedia. Locations can be entered individually, or imported using KML or ARML. KML is an XML-based language for describing geographic data and is the format used by Google Earth. ARML is an extension to KML and is an initiative to create a standard format for augmented reality browsers [12]. Currently it is not clearly specified and only supported by Wikitude.

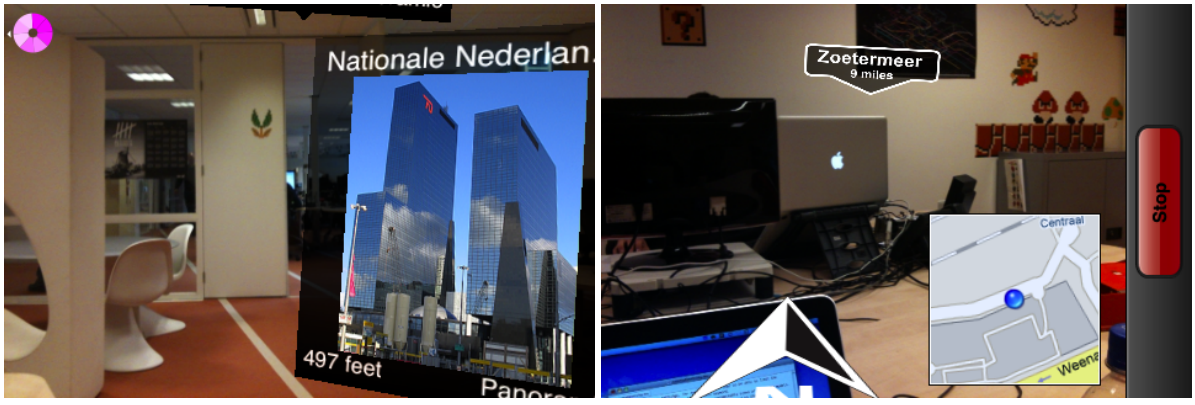
## 3.3 Other browsers

Other mobile browsers we were able to find (for iPhone) are acrossair [1] and Heads Up Navigator [8]. See figure 3 on the following page.

Heads Up Navigator only displays locations defined by the user in the application using a simple label.

The acrossair browser uses a concept similar to layers or worlds. For some layers, it displays a full thumbnail instead of a simple icon. Instead of displaying points of interest that are farther away in the (virtual) distance, it stacks points of interest on top of each other towards the sky. In this way, points of interest in the same horizontal viewing direction do not occlude each other.


Figure 2: Wikitude [12]




(a) acrossair (b) Heads Up Navigator

Figure 3: Other browsers

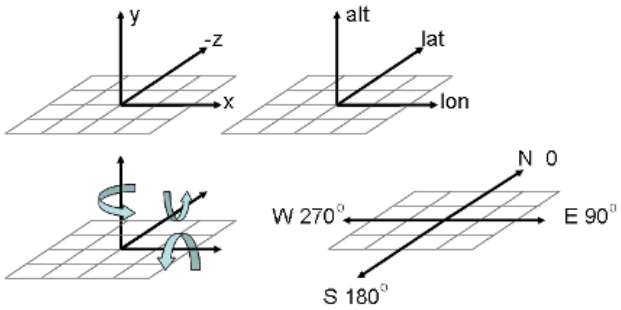



Figure 4: The Gamaray coordinate system [6]

# 4 Gamaray

This section describes the Gamaray augmented reality browser, the Android browser that we will be porting to the iPhone platform. This section first describes the important aspects of the browser's implementation and then shows the protocol that is used to communicate with the backend.

## 4.1 Mobile augmented reality browser

In Gamaray, the virtual world is called a *dimension*. The dimension is described in a Gamaray Dimension Description File (GDDF) [6], which contains information about the objects that can be displayed in the browser. Objects can be either features—text, images, or 3d models displayed on a specified location in the 3D world—or overlays—text or images that are displayed on a fixed location on the screen. In addition to the camera image, projected features, and overlays, Gamaray shows a radar which displays the location of nearby features, and some general information, such as the distance to the nearest feature.

For features, Gamaray determines their relative position to the camera and then projects them on top of the camera image. The location of features on the screen is continually updated based on updated position and orientation information. If indicated, a feature is plotted on the radar as well as projected on the camera image.

Feature locations are defined by their geographical coordinates, combined with a possible offset in meters defined in the Gamaray coordinate system. This coordinate system is displayed in figure 4. Additionally, it is possible to change the anchor position, eg. an anchor position of 'TC' on an image specifies that the top center point of the image is used as its reference point: the top center point of the image is located on the specified geographical location.

Gamaray supports PNG and JPEG images. The images are defined as assets, which contain a URL to the asset, and the asset type (PNG, JPG, or GAMA3D—a custom 3D model format).

Overlays are displayed independently of the position and orientation of the smartphone. They are simply displayed on the specified location on the screen.

The user can press features and overlays, after which a website is visited, the dimension is refreshed, or the dimension is changed by loading a GDDF from a possibly (different) URL.

## 4.2 Backend specification

Gamaray uses XML based GDDFs to store data about the dimension. When the application is started, it receives an URL for such a GDDF. The browser then sends a HTTP POST request containing the position, orientation, and other relevant data. The HTTP server replies with a GDDF, which may be dynamically created from the POST data.

The GDDF file contains, amongst other things, the conditions for refreshing the data. The data is refreshed based on the time since the last update and based on the distance travelled since the last

update. The GDDF defines the maximum distance travelled and the maximum time period after which the GDDF is refreshed. Additionally, the GDDF can be refreshed based on an event attached to features or overlays, and it can be manually refreshed by the user. The GDDF contains the URL that should be used for refreshing the data, this URL may be different from the original URL, it may for example include a session ID to keep track of the user.

# 5 ARena

This section describes ARena [9], the system developed by Finalist for Stichting Kennisnet that uses the Gamaray augmented reality browser. The main goal of our project is to allow the ARena backend to be used in combination with the iPhone platform.

ARena allows teachers to define questionnaires in a web interface based on Google maps. This is done by placing information, questions, and images on the map. These objects will be visible in the Gamaray browser. Each of these objects has a location and a radius within which it is visible. Information and images are displayed in the virtual world, and questions are multiple choice questions that can be answered by selecting one of the options.

When the teacher has finished the questionnaire, the ARena web interface provides a series of QR-codes[1] each of which can be scanned using a smartphone by a team of students to participate in the questionnaire. This opens the Gamaray browser and loads the virtual world created by the teacher. The teacher can now keep track of the location and progress of students in real time.

Students walk through the questionnaire by looking for the nearest object in the virtual world on their smartphone and walking towards it. Once arrived at a question, Gamaray displays the question and possible answers. The team then selects the option they think is correct. Images and text are simply displayed in the virtual world, facing the viewer. Once done, the team walks towards the next object, until the tour is finished and all questions are answered.

# 6 Theory

In this section, we describe the theory that is necessary for implementing our augmented reality browser. We have identified the following areas that need investigating: dealing with geographic information, determining the orientation of the device, filtering sensor data and concepts of 3D projection.

## 6.1 Geography

Since Gamaray uses a Cartesian coordinate system whose origin is at the camera's position (see paragraph 4.1 on the preceding page), it converts any geographic coordinates (WGS84 reference coordinates) to this coordinate system. The current Gamaray browser performs this translation by computing the distance along the line of longitude[2] and the line of latitude separately and using these distances as the $-z$ and $x$ coordinates in the Gamaray coordinate system respectively. Due to the curvature of the Earth this linearisation is not accurate for large distances, but it is sufficiently accurate for Gamaray's purposes.

Instead of computing a local East, North, Up (ENU) coordinate system such as the one Gamaray uses, we will compute the Earth Centered Earth Fixed (ECEF) coordinates [13]. This means that we will convert the geographic coordinates to a Cartesian coordinate system centered in the earth's center of mass, in which the axes pass through the true North pole and the intersection of the prime meridian and the equator. The conversion from WGS84 coordinates to ECEF coordinates is quite trivial [5]. The advantage to this approach is that large distances can be more accurately measured and projected and altitudes can be represented more realistically. It is also a model that more closely corresponds to reality, which makes it easier to reason about the model.

Other than absolute altitudes for features, Gamaray supports relative altitudes. This means that the position of objects is relative to the device's altitude. Implementing this requires recomputation of object

[1] A two-dimensional barcode.

[2] The line of latitude or line of longitude is the arc on the globe with a constant latitude or longitude respectively.

position every time the GPS reference altitude changes. Additionally, text and image features face the observer, which require their orientation to be updated every time the smartphone is reoriented. However, these computations have to be performed in the current Gamaray browser as well, so using the ECEF coordinate system will require a comparable amount of computations.

## 6.2 Orientation

Determining the orientation of the device relative to Earth is more complicated than determining its location. To compute the orientation, two vectors are needed: the vector that points towards Earth's gravitational pull and the vector that points to true North.

The 'gravity-vector' is given by the accelerometer after normalizing the sensor values. When trying out Layar (see paragraph 3.1 on page 2), we noticed the (virtual) horizon wasn't always straight when compared to the real horizon. This is probably caused by the fact that the accelerometer is not always put perfectly straight into the smartphone. One solution to this problem would be to simply not show a virtual horizon or grid so that the user (probably) won't notice it. Another solution would be to allow calibration of the accelerometer to compensate.

Also, when looking around with the device from side-to-side, more than just gravity is included in the 'gravity-vector', making the horizon and all virtual objects wobble. One might be able to compensate for this using both the compass and the length of the gravity vector, which is greater when additional acceleration is imposed on the device. This could be used to smooth the compass readings at the same time.

The true North pole is in a different location from the magnetic North pole, so computing the heading of true North requires more than just reading the 3-axis magnetometer. First, you have to account for magnetic declination, which is the angle between true North and magnetic North. Second, there is magnetic inclination to account for, since the Earth's magnetic lines of force are not parallel to its surface. Both change over place and time. [11] Computing declination and inclination requires the use of a model; we found that WMM2010 [20] has a C implementation which we can use to compute these values.

In The Netherlands, the magnetic declination is less than $1^o$; in other locations around the world, declination is usually below $20^o$ (excluding exotic locations, such as the North pole). These declinations may be noticable, but are not large enough to be critical initially. Since correction for declination is not trivial, we will attempt to implement this in a later stage of this project.

Magnetic inclination can be safely ignored because we do not use the compass to determine the horizontal plane. In a later stage, we may be able to use the orientation of the horizontal plane computed from the compass to correct the horizontal plane measured from the accelerometer. Combining these measurements correctly may improve accuracy, but may be significantly harder than using the accelerometer measurements directly, which leaves the need for inclination correction for future work.

Although the iPhone supports a trueHeading method to compute the true heading to the North pole, this method does not produce expected results when looking up. Tilting the iPhone for about $45^o$ upwards suddenly flips the compass readings by $180^o$. This renders the use of trueHeading useless for our purpose.

## 6.3 Signal filtering

Since we need data from various sensors that are prone to noise and inaccuracies—such as the accelerometer, the compass and the GPS—we might need to apply some filtering to this data.

The Gamaray browser does not directly filter the accelerometer and compass data. Instead, it determines the new orientation of the device and applies a moving average filter on each component of the orthogonal orientation matrix. This does not guarantee the resulting matrix remains ortogonal, but apparently the developers decided to ignore this.

In our application we will more likely filter the sensor data before we perform any calculations on the data. We will do this because the noise in the compass and accelerometer data are most likely unequal, so that filtering them separately allows for better filter adjustment. We will have to determine what filtering methods to apply when we are implementing the sensor reading and tweak them in the

final application. Simple low-pass, median, or moving average filters will probably suffice. We also took a brief look at Kalman filters [19], but those are outside the scope of this report.

Gamaray does not apply any processing to the GPS data. It is probably not necessary to do this in our application either, since the geographic information is not used for realtime updating of the display, unlike—for example—in an application used for navigation.

When trying out Layar (see paragraph 3.1 on page 2), we noticed both jitter and lag in the movement of virtual objects when compared to the real world. This did not appear to be caused by limitation in graphics performance, but by bad (filtering of) sensor data.

To get a better insight in the values the sensors return, we built a test application that logs the $x$, $y$, and $z$ components of the accelerometer and compass whenever these values are updated. Our measurements show that the compass updates with about 33 Hz and the accelerometer with about 100 Hz. The sensors have a resolution of about 0.25 $\mu$T for the compass and about 0.018$g$ for the accelerometer, but are subject to random noise. The correlation between two samples measured in sequence when the device is moving nor rotating for the compass is above 0.7, and for the accelerometer is below 0.2—exact values differ slightly per axis. This shows that the compass noise of two subsequent samples is not independent, which may be important to realize when determining the appropriate filtering technique and properly adjusting it.

### 6.4 Projections

Due to the differences between the Android and iPhone platform, the method of projecting objects will differ from the method applied by the current Gamaray browser. Features and the camera can be positioned by using the coordinates in the ECEF coordinate system. The iPhone supports the assignment of homogeneous 3D transformation matrices to views, such as a perspective transformation. These matrices are then automatically applied to the view and its subviews when the view is rendered.

To compute the appropriate perspective transformation matrix, the camera's field of view is required. This can be computed from the width and height of the active image area and focal length of the camera [2]. For the iPhone the active image area is 3.58 mm by 2.69 mm and the focal length is 3.85 mm [3]. This leads to a horizontal field of view of about 50º and a vertical field of view of about 39º. These values correspond to our rough measurements (about 50º and 40º).

Since the iPhone camera sensor has an aspect ratio of 3:4, but the iPhone screen aspect ratio is 2:3. To display a full screen camera image, the captured image must be scaled to fit. To avoid deformation, the aspect ratio of this image must be kept equal, which results in part of it ending up outside the screen. Due to this scaling, the vertical field of view of the camera image is not equal to the vertical field of view of the image that is eventually displayed on the screen. Care must be taken to use the right value for the field of view.

## 7 iPhone platform

The augmented reality browser we are building is targeted at Apple's iPhone platform. This platform is essentially comprised of an Apple line of products running a common operating system. This section discusses the characteristics of the soft- and hardware in this platform and how these affect the implementation of an augmented reality browser.

All information given in this section can either be reviewed in the official Apple documentation or was determined experimentally.

### 7.1 Software

The common denominator of the iPhone platform is iPhone OS. It is the operating system that drives various mobile devices and acts as a platform for running all kinds of applications.

### 7.1.1 Overview

The iPhone OS is very similar to Mac OS X and runs a Mach microkernel in a BSD (UNIX-like) environment. Developers have access to the standard C library as well as other low-level operating system services.

On top of the low-level interfaces and libraries exist various general frameworks that can be used by developers. An important framework is Core Foundation, consisting of C-based interfaces that provide basic data structures such as strings, arrays and dictionaries, threading, sockets, etc. On top of this framework sits Foundation, which provides object oriented Objective-C wrappers to the features provided by Core Foundation and more. Another relevant framework is Core Location, which gives access to the user's current location and compass bearing.

In addition, there are a number of frameworks for graphics, audio and video. Most relevant are Quartz, Core Animation and OpenGL ES. Quartz is a C-based drawing engine that is used for 2D drawing. Core Animation provides hardware-backed animation and compositing services. Finally, OpenGL ES is a subset of OpenGL for embedded devices for drawing 2D and 3D content.

Most importantly, iPhone OS contains high-level frameworks that must be used in order to develop an iPhone application. This layer is often referred to as Cocoa Touch. Key framework in this layer is UIKit, which provides an Objective-C programming interface for implementing graphical and event-driven applications. It also has an interface for accessing the device's acceleration.

### 7.1.2 Implementing augmented reality

The user's current location can be determined by the Core Location framework. The accuracy depends on the available hardware. The same framework provides the device's heading if the device has a digital compass. In combination with the device's acceleration, that is provided by UIKit, the orientation of the device can be determined.

Core Location also has a small number of basic data structures and methods for working with geographical information, such as calculating the distance between two WGS84 coordinates.

As of iPhone OS 3, there are no frameworks that allow direct access to the device's camera (assuming it has one). The UIKit framework does provide developers with a class that allows the user to take a picture. This class displays a live view of the camera's output. Therefore, the only way to facilitate an augmented reality browser is by (mis)using this class and displaying an additional layer of content on top of the live camera view. (This use is tolerated by Apple.)

For displaying content that is simple and 2D, it is not necessary to use OpenGL ES. Since the high-level view classes provided by UIKit are backed by Core Animation layers, they can easily be animated and have 3D transformations applied to them (e.g. when perspective is needed). Core Animation layers, in turn, are hardware-accelerated and should provide ample performance.

Unfortunately, the rendering architecture used by iPhone OS works with 32-bit floating point numbers. This means there are only about 7 significant decimal digits, giving us only just sufficient accuracy for working with numbers as high as the radius of the Earth ($6.378 \cdot 10^6$ m) and as low as the distance to a feature (several meters). Therefore, we may need to offset the origin of our coordinate system to a point close to the viewer if the precision is otherwise insufficient.

## 7.2 Hardware

Current devices part of the iPhone platform are the iPhone, the iPod touch and the iPad. Things they have in common are the iPhone OS, a multi-touch screen (and the lack of a keyboard) and an accelerometer that measures the device's acceleration in three axes. The acceleration can be used to determine the device's relative orientation to Earth's gravitational pull.

However, an augmented reality browser also needs to know how the device is oriented relative to the Earth's pole and therefore needs a digital compass. Only the most recent third generation model of the iPhone (the 3GS) and the recently introduced iPad have a compass.

In addition, only the iPhone has a camera and a GPS-unit for accurate location determination.

Therefore, an augmented reality browser for the iPhone platform requires an iPhone 3GS.

# 8 Conclusion

This report showed the findings of our research on augmented reality and the iPhone. It introduced augmented reality and augmented reality browsers, surveyed existing mobile browsers, described the implementation of the current Gamaray browser for the Android platform and its application in the ARena project, introduced the necessary theory for implementing an augmented reality browser and discussed how the iPhone platform affects our implementation.

Augmented reality is a combination of virtual reality and actual reality. One way to use augmented reality is to use a mobile augmented reality browser on a smartphone, which projects points of interest on top of the camera image in real-time. Besides a camera, this also requires an accelerometer and a compass to determine the orientation of the device. When points of interest have a geographic location, the user's current geopgrahic location must be determined using GPS.

Popular (commercial) implementations of augmented reality browsers are Layar and Wikitude. Most browsers work by allowing the user to select one or more 'worlds', each 'world' being a static or dynamic set of points of interest. Usually, the browser periodially fetches the points of interest nearest to the user's location from a webservice.

Gamaray consists of an augmented reality browser for Android and an associated webservice specification. We will build a similar browser for iPhone that uses the same webservice interface. A virtual world in Gamaray is called a 'dimension'. The dimension contains three kinds of points of interest: text, images and 3D models. In addition, it is possible to display overlays independently of the device's orientation. These objects are provided by a HTTP and XML based webservice and are refreshed when time or distance has changed significantly or on user action.

ARena allows teachers to define questionnaires in an augmented reality world for students. This is a concrete implementation of the Gamaray webservice, the implementation that we will work with. Questionnaires can contain text, images and questions. Questions are Gamaray overlays. Students can start the questionnaire by scanning a QR barcode (using an external application) that resolves to a URL recognized by the Gamaray browser. Teachers can track the progress and location of students.

In order to implement augmented reality, we need to be able to work with geographic coordinates, to interpret accelerometer and magnetometer readings, to determine the device's orientation from these readings and to use this information to project the virtual world onto a camera image of the real world. Luckily, most of these calculations do not require a high level of accuracy, or allow the accuracy to be improved later on.

Only the iPhone 3GS has the necessary hardware to use augmented reality on the iPhone platform. The iPhone OS that runs on it has all the necessary high-level frameworks for easily determining the user's location (Core Location), determining the device's orientation (part of UIKit) and using 3D transformations to project objects onto a live camera image (Core Animation).

Our findings suggest that there should be no technical issues or limitations for implementing a Gamaray augmented reality browser on the iPhone-platform.

# References


[1] Acrossair. `http://www.acrossair.com/`. Retrieved May 6, 2010.

[2] Angle of view. `http://en.wikipedia.org/wiki/Angle_of_view`. Retrieved May 6, 2010.

[3] Apple iphone 3gs 16gb - what's inside. `http://www2.electronicproducts.com/Apple_iPhone_3GS_16GB-whatsinside_text-82.aspx`. Retrieved May 6, 2010.

[4] Augmented reality soundwalk with layar. `http://www.soundwalk.com/blog/2009/08/17/augmented-reality-soundwalk-with-layar/`. Retrieved May 6, 2010.

[5] Datum transformation description. `http://www.satsleuth.com/GPS_ECEF_Datum_transformation.htm`. Retrieved May 6, 2010.

[6] Gamaray developers information. `http://www.gamaray.com/developers.html`. Retrieved May 6, 2010.

[7] Geldautomaten zoeken (Netherlands). http://site.layar.com/geldautomaten-zoeken-netherlands/. Retrieved May 6, 2010.

[8] Heads Up Navigator. http://itunes.apple.com/nl/app/heads-up-navigator-3d-augmented/id330367042. Retrieved May 6, 2010.

[9] Innovatie met Augmented Reality als leermiddel. http://www.finalist.nl/content/44449/innovatie_met_augmented_reality_als_leermiddel. Retrieved May 6, 2010.

[10] Layar. http://www.layar.com/. Retrieved May 6, 2010.

[11] Magnetic declination. http://earthsci.org/education/fieldsk/declin.htm. Retrieved May 6, 2010.

[12] Openarml. http://www.openarml.org/. Retrieved May 6, 2010.

[13] Other earth based coordinate systems. http://en.wikipedia.org/wiki/Geodetic_system#Other_Earth_based_coordinate_systems. Retrieved May 6, 2010.

[14] Press release: The first mobile augmented reality browser premiers in the Netherlands. http://site.layar.com/company/blog/press-release-the-first-mobile-augmented-reality-browser-premiers-in-the-netherland Retrieved May 6, 2010.

[15] Spaceinvaders 3D (International). http://site.layar.com/spaceinvaders-international/. Retrieved May 6, 2010.

[16] Wikitude. http://www.wikitude.org/. Retrieved May 6, 2010.

[17] Wikitude AR Travel Guide (Part 1). http://www.youtube.com/watch?v=8EA8xlicmT8. Retrieved May 6, 2010.

[18] Wikitude worlds – iphone 3g version released. http://www.wikitude.org/dewikitude-worlds-jetzt-fr-iphone-3g-verfgbarenwikitude-worlds-iphone-3g-version-re Retrieved May 6, 2010.

[19] Joã Luís Marins, Xiaoping Yun, Eric R. Bachmann, Robert McGhee, and Michael J. Zyda. An extended kalman filter for quaternion-based orientation estimation using marg sensors. pages 2003–2011, 2001.

[20] National Geophysical Data Center. The world magnetic model and associated software. http://www.ngdc.noaa.gov/geomag/WMM/soft.shtml. Retrieved May 12, 2010.