ESTONIAN INFORMATION TECHNOLOGY COLLEGE

Tarvo Reinpalu

OPTIMIZING UITABLEVIEW IN THE PAIRBY IOS APPLICATION

Diploma Thesis

IT SYSTEMS DEVELOPMENT CURRICULUM

Supervisor: Gary Planthaber

Consultant: Toomas Lepikult

Tallinn 2017

Author's Declaration of Originality

I declare that this thesis is an original report of my research, has been written by me and has not been submitted for any previous degree. The experimental work is almost entirely my own work; the collaborative contributions have been indicated clearly and acknowledged. Due references have been provided on all supporting literatures and resources.

Author: Tarvo Reinpalu .....

(signature and date)

The thesis adheres to all specified requirements

Supervisor: Gary Planthaber .....

(signature and date)

Contents

Introduction

Mobile applications must have highly responsive user interfaces to maintain their brand's reputation and keep users engaged. [24] [1] Of all participants in a study by HP, 55% said that the application alone is to blame for bad performance, as opposed to other plausible causes like their possibly underpowered device or background processes, and 37% stated that errors and freezes made them think less of a company's brand. [24] A study by Apigee revealed that 18% of survey participants are likely to delete an application after it freezes for more than 5 seconds. [1] The percentage of Apigee survey participants who would delete such an application increased to 38% when the freeze time exceeded 30 seconds. [1] The Apigee survey also revealed that two of five most common reasons for bad reviews were related to user interface responsiveness: freezes were the reason for leaving a bad review for 76% of the survey participants and slow responsiveness for 59% of the participants. [1]

An expectation of having a highly responsive user interface has been around as long as user interfaces. [19] Jakob Nielsen, a web usability consultant who holds a Ph.D in human–computer interaction [14], has written that even the slightest delay (anything above 0.1 seconds) is not considered instantaneous by users, and anything above 1 second results in the user's flow of thought being interrupted and focus being lost. [19] Any delay above 10 seconds results in the user's attention being completely lost and, without any feedback about progress, they may not return their attention to the application. [19]

The studies cited in the previous paragraphs show that users expect a responsive user interface and if the application fails to provide it they might delete the application, leave a bad review, or both.

Pairby, a mobile dating service of which the author is a co-founder, has applications on both

Android and iOS. Neither application is an exception to the users' previously described expectation of having a responsive user interface. This thesis' subject is directly derived from the author's interest in solving user interface responsiveness and low frame rate issues in the Pairby iOS application's vertically scrollable views, which use the iOS UITableView component.

The following subsections describe the problem and declare the goal of the thesis in greater detail, how the thesis is structured, and how the problem was approached.

Description of the problem

Pairby's iOS application contains a multitude of different vertically scrolling views, which can contain a large number of subviews with no defined limit. One of the vertically scrolling views is the message list view, which this thesis uses as for reference. The message list view uses a UITableView to render a vertical list of views (called cells) that display messages, which can contain either only text or one or multiple visual media items.

The Pairby iOS application's first prototype encountered user interface responsiveness and low frame rate issues in the message list view. The prototype project was discarded, in part, because of those performance issues. Exploring ways to improve upon the two encountered issues in the message list view, and other similar scrollable views, was something that needed to be done before building the new project.

At the time of writing this thesis, the development of the new iOS application for Pairby had begun, but a solution to the user interface responsiveness and low frame rate issues in the vertically scrollable views had not yet been found.

Goal of the thesis

The goal of the thesis was to explore, analyze, and document ways to improve the Pairby iOS application's vertically scrolling views' user interface responsiveness and frame rate. More specifically, to find ways to improve the UITableView component's performance in

the aforementioned performance aspects when rendering complex views that contain either text or one or multiple visual media items.

The optimizations were expected to have a positive impact on user interface responsiveness, frame rate, or both and be applicable to the Pairby iOS application's vertically scrollable views, including the message list view. All the optimizations were expected to be documented as independently of each other as possible, to allow for their use as separate resources.

Structure of the thesis

Reaching the goal of this thesis required a strictly structured approach. The following paragraphs describe the order of operations chosen to approach the problem.

Specifications - outlines a set of specifications describing the content which message objects could contain, and how the message views should look.

Measuring performance - explains the techniques used to measure different user interface responsiveness and device performance-related metrics, and explains in detail how the test environment was created, on what device the tests were run, and how the metrics were analyzed.

Comparison points - describes how two different vertically scrollable views, to be used as reference points, were created utilizing both the specification, and the set of sample message objects produced.

Performance optimizations - focuses on exploring and analyzing different optimization methods. All performance optimizations were built into the same project, with the starting point being the worst case scenario, and resulting performance was benchmarked after each optimization.

Findings - compares the combined result of all optimizations with positive impacts on user interface performance against the two reference points and analyzes the results.

1 Specifications

The test project was designed to broadly mimic the user interface and data layer of the production Pairby mobile applications message list view. The following subsections describe the requirements for the test data and user interface, as well as how the test data was generated.

1.1 Test data requirements

The test data contained 500 message type objects. The data was stored in JSON format and saved to a file, which could then be loaded into the iOS test project using the XCode assets interface.

1.1.1 Message type

The type definition for each message object is shown below, in Table 1.

Table 1

Message type definition

Property	Type	Required	Description
messageId	Integer	true	A unique identifier generated from a sequence starting at 0 with an increment of 1.
direction	String	true	Can be either "in" or "out", representing the direction of the message relative to the current user.
message	String	false	Text message's text. Not present in media messages.
mediaItems	String[]	false	Media message's media URLs. Not present in text messages.

1.1.2 Text message object

Text messages are messages that contain only text. The content chosen for the generated text messages was tightly tied to the way they were expected to be rendered in the user interface. To generate a set of message views that have a variable height and width, with the goal of simulating real-life use cases, the number of lines in the text and the number of characters per line had to be taken into consideration.

The width of a text message view in the user interface is dependant on the length of the longest line of text in the text message object's content. That dependency was taken into consideration when generating text message content. For the purposes of simulating real-life scenarios, text messages of three different widths were generated.

The height of a text message view in the user interface is dependent on the number of lines in the text message object's content. That dependency was also taken into consideration when generating text message content. To broadly simulate real-life scenarios, text messages of three different heights were generated. The number of lines was controlled using line break characters.

Combining the two dependencies and user interface constraints mentioned in the previous two paragraphs resulted in three text message contents being created. Each of the three text message contents were expected to have different heights and widths.

1.1.3 Media message object

Media messages are messages that contain only media items. In this thesis' project, media messages can contain up to three visual media items as opposed to the maximum of 6 visual media items or 1 audio file in the production Pairby applications. The lower number and variety of media items was chosen to simplify and speed up the building of the thesis project.

The visual media items, or images, can either be static (in JPG format) or animated (in GIF format). To simulate real-world conditions, all media needed to be loaded from Pairby servers. Loading media from the Pairby servers also raised the requirement of running all

the tests on the same network to guarantee similar environmental conditions for all test cases.

1.2 Test data generation

Test data for the project was generated in a separate script, which was written in JavaScript and executed in a Node.JS environment. The output of the script was a JSON file that contained an array of message type objects. The following subsections describe in detail how the message generation script worked.

1.2.1 Message object generation

The number of messages to be generated was configurable, and was set to 500 for the purposes of this project.

The type (text or media) of each message was decided using a semi-random probability technique: a random number from 0 to 1 was generated and compared against a configurable parameter called "media message probability". If the generated number was less than or equal to the media message probability parameter, the message type was set to be media message and therefore its mediaItems value was then defined. If the generated number was larger than the media message probability parameter, then the message was set to be of text type and its message value was then defined. The test data set was generated with the media message probability set to 0.25 (25%).

Each messageId value was set to the sequence number in the running loop (which started at zero and ended at the number of generated messages minus 1). For the 500 messages generated for the purpose of this test, the values were in the range of 0 to 499 (inclusive).

Each message's direction value was also defined through a random number generator and was set to have an equal chance of either being outbound or inbound. This process was similar to the way the message type was chosen: a random number between 0 and 1 was generated and it was compared against 0.5, if was less than or equal to 0.5, direction value was set to "out", otherwise it was set to "in". This random number generation technique

meant in theory that half of the messages would be outbound and the other half inbound, but not in a pre-determined order.

1.2.2 Text message content generation

As defined in section 1.1.2, text messages of three different heights and widths exist. The text for each different size of text message was static (pre-defined). Selecting one of the three different contents (which would determine the size of the message view) was random and each size had an equal chance of being selected. A random whole number between 1 and 3 (inclusive) was generated. If the generated number was 1, then the smallest width and height message content was added to the message object. If the generated number was 2, then the medium size message content was added to the message object. If the number was 3, then the tallest and widest message content was added to the message object.

1.2.3 Media message content generation

As defined in section 1.1.3, media messages can contain up to three visual media items (images) which were expected to be loaded from the Pairby servers, through the Internet. The constraint of containing either one, two or three images limited the number of different media message view types to three. The three media types were media message views that contain either one, two or three images.

The set of sample images consisted of 100 static images of JPG format and 10 animated images of GIF format. The data generator script uploaded the images to the Pairby servers and then returned an array of URLs that point to the image files on the Pairby server.

The number of visual media items to include with a specific media message was a random number between 1 and 3 (inclusive). The image URL, or URLs, that point to the images were chosen randomly and duplicates were not prevented. The lack of duplicate prevention meant that there could be media messages that contained the same media item URL multiple times. Even though this would not be possible in the production Pairby application, there was no direct reason to prevent it from happening in the thesis project as it should still work as expected.

1.3 User interface requirements

As stated in the introduction paragraph of section 1, the user interface was designed to mimic the Pairby applications' message list view. The following subsections describe in detail what the message views in the messaging list view were expected to look like.

1.3.1 Common design elements

Both the text and media message views were expected to have rounded corners with a radius of 15 points and a border of gray colour (`#ECEFF1`) with a width of 1 point.

1.3.2 Text message view

The text message view was expected to handle line breaks, which means it should expand vertically to accommodate multiple lines of text. The background color for inbound and outbound messages was expected to be different: teal (`#4DB6AC`) for outbound, gray (`#ECEFF1`) for inbound.

1.3.3 Media message view

The media message view stylings are dependant on the number of images they contain, as is the case in the production Pairby applications. The following paragraphs define the layout and size of the three possible view types, which were media message views containing either one, two or three media items.

1 media item A single media item message should display the image with a width and height of 180 points.

2 media items A two media item message should display the images side by side, with the image widths and heights being 90 points. The gap between the media items should be 10 points.

3 media items A three media item message should display the images side by side, with the image widths and heights being 90 points. The gap between the media items should be

10 points.

Figure 1, shown below, displays the three different media message view types and their size, both in points and in relation to each other.

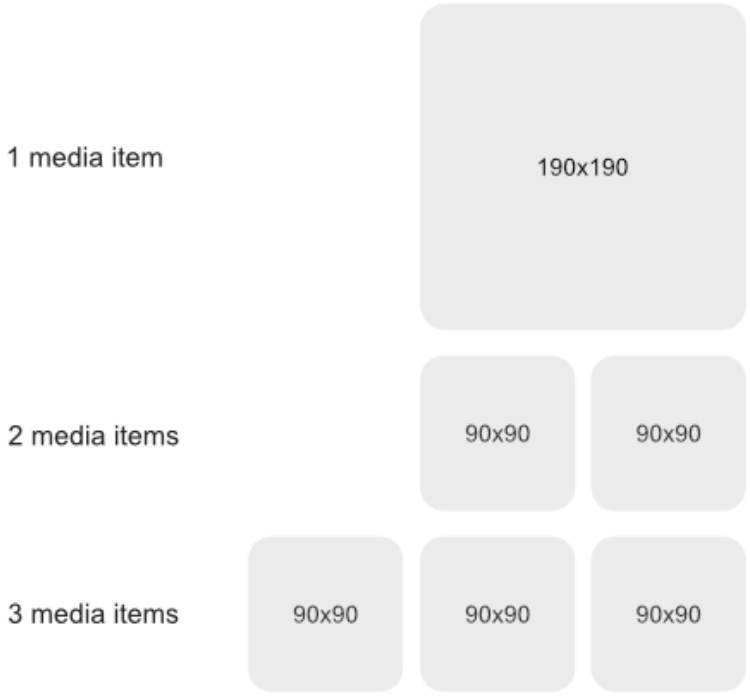

Figure 1 User interface mockup showing the three different media message view types and their expected sizes in points

2 Measuring performance

To gather quantifiable information, and not base comparison points on visual judgement, there needed to be a way to run reproducible tests and gather numeric statistics from each test. The gathering of quantifiable information was achieved in two different parts: the creation of a repeatable test case (programmatic scrolling that simulates human interaction) and using different software to retrieve and save information about the user interface related metrics, like frame rate, and the test device's resource usage, like average CPU usage.

The following subsections describe, in depth, how the repeatable test case was created and works, how the user interface related metrics were measured, what device was used as the test device and how the device resource usage was monitored, and what steps were taken to make the gathered data more easily analyzable.

2.1 Repeatable test case

The repeatable test case's purpose was to simulate human interaction through scrolling and do the exact same way every time the test is executed. The test consisted of scrolling from the top of the list view to the bottom five times, with the scroll speed continuously increasing.

Achieving the continuously increasing speed was done using the UITableView function `setContentOffset(contentOffset, animated: true)`, along with a repeating Timer. The Timer fired every 0.4 seconds and forced the view to scroll, starting from the top, towards the bottom. The scrolling was purposefully not linear: starting from 100 points, every time the Timer fired and the view was scrolled, the scroll offset value increased by 30 points. This means

that the first time the Timer fired, the `setContentOffset(contentOffset, animated: true)` was executed with a 100 point increase in the vertical content offset. The next time the Timer executed, the increase was 130 points and after that, 160 points.

To achieve the scrolling to bottom five times expectation of the test, the view's vertical offset was instantaneously reset to the 0 (causing the UITableView to start at the very top), if the inner UITableView reached the bottom and an internal timer recording the count of resets had not reached 4. If the internal timer recording the count of resets (number of scrolls to the top) reached 5, the test execution was stopped and the test was considered over.

2.2 Measuring test case performance

Five metrics were monitored to get an overview of the performance and efficiency of the UITableView. The measured metrics are both user interface and device resource usage related. The user interface related metrics are the test duration and frame rate. Device resource usage related metrics are average processor, graphics processor and memory usages.

Measuring of the duration of the test case and the frame rate of the user interface was done programmatically in the thesis project. Measuring the frame rates was handled by a custom implementation of the FPSCounter [12] library. Measuring the test duration was done by recording the system absolute time, using the `CACurrentMediaTime()` function, at the start of the test and comparing it against the system absolute time at the end of the test.

In addition to measuring frame rate, a custom metric called "frame rate stability" was introduced. The simplified definition of frame rate stability is "percentage of time spent within 20% of the average frame rate". In the code, the frame rate stability was calculated as the percentage of measured frame rate points that were within 20% of the average frame rate. Because the actual measurement technique for this metric is an estimation of its theoretical definition, its accuracy is not guaranteed to directly match its theoretical definition. For the purposes of this thesis project, the slight variation in accuracy is considered acceptable and the metric will still provide a beneficial metric value for analyzing the overall responsiveness of the user interface.

The device resource usage related metrics (average CPU, GPU and RAM usage) were measured using a third-party software called GameBench [11]. The software required very little setup and offered very few configuration options. The test device monitoring process in GameBench was started at the same time as the repeatable scrolling test, described in subsection 2.1, and stopped when the test had finished, manually. After stopping the test device monitoring process, GameBench uploaded the test data to their servers and the data became available and exportable on their website. The test data contained, among other device related information and metrics, the average CPU, GPU and RAM usage.

2.3 Analyzing gathered metrics

All the recorded metrics, mentioned in the subsections of this section, resulted in a total set of 6 different measurement points. The resulting list of measurement points is displayed below, in listing 1.

- Duration (seconds);
- Average frame rate;
- Frame rate stability;
- Average processor usage;
- Average graphics processor usage;
- Average memory usage.

Listing 1 List of measurement points recorded for every test case

2.4 Test device

All tests were run on an iPod Touch 5G, as it is a good example of a low-powered device by today's standards. The 5th generation iPod touch has a retina display (resolution 1136x640) [3] while having a relatively underpowered [27] processor (dual core Apple A5 underclocked to 800 MHz) [16]. Having a high resolution display with an underpowered processor means that the processor may not be able to handle complex rendering tasks that well, as the number of pixels it is required to control is too large for it to handle, among other tasks. [18] The high-resolution display and underpowered processor combination is

expected to have the effect of making the user interface responsiveness and device resource usage related metrics change more noticeably after every optimization.

The input lag of the test device was taken into consideration when making assumptions and calculations regarding the user interface response times. No data could be found regarding the input latency of the touch screen of the test device, which was the iPod Touch 5G. As online sources stated that the iPod Touch 5G has the same screen as the iPhone 5 [26] and that the iPhone 5's screen was measured to have a input latency of 55 milliseconds [17], it was assumed, for the purposes of this thesis, that the test device's screen input latency is 55 milliseconds.

3 Third party libraries used

Creating every component of this project from scratch would have been too large of a task to cover in this thesis, therefore third-party solutions were used to simplify some tasks. The affected tasks were for example JSON parsing, image loading and caching, detailed logging and in-code constraint creation.

The following subsections will both list all the used third-party solutions and describe their purpose in the project.

3.1 JASON

JASON is a faster JSON deserializer written in Swift. [7] The sole purpose for this library in the test project was to turn the test data from JSON form into Swift objects. Even though deserialization of the JSON data could have been done using Swift's built in capabilities, JASON allowed the same goal to be achieved with much shorter and more readable code.

3.2 SnapKit

SnapKit is a DSL to make Auto Layout easy on both iOS and OS X. [21] SnapKit was used extensively in the test project to create constraints in the code, rather than in the storyboard. Creating layout constraints in the code had many benefits over creating them in the storyboards. For example, creating programmatic constraints removed the system overhead brought in by the processing of storyboards. SnapKit also gave the engineer very precise control over each constraint and provided an overview of all constraints that exist and when they will be created. Storyboards would have only had the benefit of providing the engineer with a graphical interface in which to create those constraints.

3.3 PINRemoteImage

PINRemoteImage, also known as PINRemoteImageManager, is an image downloading, processing and caching manager. [22] PINRemoteImage was used for the exact three things brought out in the library's description: it downloaded the images, allowed them to be processed, and then stored safely in a disk cache. The PINRemoteImage library made the image downloading, processing and caching tasks require very little work to be done in the thesis project's own code. Downloading images would have been trivial with the Swift's built in tools as well, but processing and caching would have been more difficult. The ability to process the images is especially useful as it allows the images to be scaled down if necessary, which in turn makes the image smaller in dimensions and the file smaller in size. The library's cache manager is also powerful and extremely useful as it provided the option to cache different, processed, versions of the same image, including the original one. The different versions were distinguished by a string key passed to the process function.

3.4 XCGLogger

XCGLogger is a debug log framework for use in Swift projects. [29] XCGLogger improved Swift's built-in logging mechanism by adding a lot of useful, yet optional, information about every log statement. For example, XCGLogger's log statements displayed the thread name on which the logging function was executed on, without forcing the engineer to explicitly print out the thread name. XCGLogger also printed out function names, file names and line numbers in a similar manner to the thread names printing. The library's uses and benefits in the test project are the same as the examples brought out previously (such as the identification of threads and functions).

3.5 FLAnimatedImageView

FLAnimatedImage is a performant animated GIF engine for iOS. [10] FLAnimatedImageView does one thing that the iOS's built in UIImageView was not able to do: play animated images (GIFs). FLAnimatedImageView was very important to both the test project and the actual Pairby iOS application due to the fact it can display animated images.

3.6 FPSCounter

FPSCounter is a small library to measure the frame rate of an iOS Application. [12] The FPSCounter library was used for the purpose of measuring frame rates in the test project. The data gathered by FPSCounter was used to analyze the user interface responsiveness of each optimization method (and/or test case).

4 Comparison points

To accurately compare performance changes in the thesis project, a set of reference points were needed and a starting point was needed to relatively compare changes to the metrics listed in listing 1 after every optimization. Two reference points were created for this purpose, one called "best case scenario" and the other called "worst case scenario". Both the reference points used the same data set, which was specified in detail in subsection 1.2.

The following subsections describe, in detail, how the two reference points were created and what their purposes were.

4.1 Best case scenario

The goal of the best case scenario was to display the best possible performance. To achieve the best possible performance, simplest possible subviews were used. To create the simplest possible subviews, images were omitted from he equation, and built-in UITableViewCell class instances were used, displaying only plain text.

To remove media messages from the equation, all media messages were replaced by text messages at runtime. The text messages that replaced them contained the text "Media message with ID [N]", where N was the messageId value of the message object. Then they were rendered as text message views.

Running the performance tests yielded the results shown in Figure 2.

0.1 - Best case scenario	#1	#2	Average
Duration (sec)	66.1	66.8	66.5
FPS (avg)	60.0	60.0	60.0
FPS stability (%)	100.00%	100.00%	100.00%
Average CPU usage (%)	15.05%	17.16%	16.11%
Average GPU usage (%)	23.36%	24.36%	23.86%
Average memory usage	28.0	34.0	31.0

Figure 2 Performance test results for the best case scenario

As presumed before running the tests, the best case scenario test case displayed the best possible performance. The test suite was run twice and the averages of the two test cases were taken as final comparable results, as shown in Figure 2. The average frame rate was exactly 60 frames per second, which is the iOS golden standard [9] as well as the upper limit [28]. Both the CPU and GPU usage remained fairly low, with the averages being 16% and 24% respectively. The low usages meant that neither of the processors were stressed while still providing the user with the smoothest possible experience.

4.2 Worst case scenario

The goal of the worst case scenario was to display the worst possible performance. To achieve the worst possible performance, no known optimizations were applied and no common performance boosting patterns (for example reuse of cells) were used. All messages were displayed as specified in section 1.

Displaying of cells according to the user interface specifications, listed in section 1, required the creation of two different view classes, which both inherited from the UITableViewCell class. One of the new created classes was for displaying text messages and the other for displaying all three types (as specified in section 1.3.3) of media messages.

Running the performance tests yielded the results shown in Figure 3.

0.2 - Worst case scenario	#1	#2	Average
Duration (sec)	105.9	105.0	105.5
FPS (avg)	25.9	26.1	26.0
FPS stability (%)	24.60%	22.60%	23.60%
Average CPU usage (%)	70.00%	68.82%	69.41%
Average GPU usage (%)	65.78%	65.75%	65.77%
Average memory usage	25.0	31.0	28.0

Figure 3 Performance test results for the worst case scenario

The test results displayed relatively worse performance compared to the best case scenario (subsection 4.1), as expected. The average frame rate had dropped to 26, compared to the 60 measured in the best case scenario tests. Both the processor and the graphics processor were noticeably more stressed with the average usages being 69% and 66% respectively, compared to the best case scenario results of 16% and 24%, respectively. Unexpectedly, the average memory usage for the application decreased by 3 megabytes, even though images, which use more memory both on disk and in random access memory [13], were used in the tests.

5 Performance optimizations

This section focuses on the practical part of the thesis, which was the creation, testing and benchmarking of different optimizations. All optimizations that had a positive effect of improving either average frame rate or frame rate stability or both were kept in the test project. This was expected to ultimately lead to the creation of a project that would have all the found optimizations with positive effects on performance.

Each of the following subsections in this section describes an optimization. Each subsection is also divided into three parts: theoretical analysis of why and how the optimization should work, benchmarking of the automated test results specified in section 2 after applying the optimization, and code examples.

5.1 Reuse of cells

Cell reuse was chosen as the first optimization due to the fact it is one of the more widely known ones. That capability is built into the UITableView component by Apple and has been there since iOS 2.0. [15]

5.1.1 Theoretical benefits

Reusing of cell views means that most times, when a new message enters the view, instead of creating a new instance of the cell view class, an existing object created from that class is used. This reuse of cell views is a performance enhancement because it eliminates the overhead of cell creation. [2]

This enhancement required fundamental changes to be made to the existing cell classes. Views needed to have the ability to re-adjust themselves during runtime in order match their

design to the type of message they were meant to display. Previously, views only needed to define their layout state once, upon initialization, and would no longer have to readjust themselves.

5.1.2 Performance changes

Running the performance tests yielded the results shown in Figure 4.

1 - Reusing cells	#1	#2	Average	Change (vs 0.2)
Duration (sec)	92.9	92.4	92.7	-12.8
FPS (avg)	34.8	35.0	34.9	8.9
FPS stability (%)	27.00%	25.00%	26.00%	2.40%
Average CPU usage (%)	57.63%	55.63%	56.63%	-12.78%
Average GPU usage (%)	62.83%	62.80%	62.82%	-2.95%
Average memory usage	31.0	31.0	31.0	3.0

Figure 4 Performance test results after applying cell reuse optimizations

The optimization had a positive impact on the performance of the thesis project, compared to the worst case scenario, which was the starting point. Average frame rate increased by 8.9 frames per second, while both the processor's and graphic processor's average usage decreased by 12.78 and 2.95 percent respectively. The duration of the test case also decreased from 105.5 seconds to 92.7 seconds, displaying a decrease of 12.8 seconds. The decrease in duration was directly tied to the average frame rate, as all the necessary frames were rendered faster. Average memory usage did show an increase of 3 megabytes, from 28 to 31 megabytes, but this change was small enough to classify it as minor and label it as "did not change".

5.1.3 Code examples

Apple has already built support for cell reuse into the UITableView component, making its implementation easy. Each of the different reusable cell classes have to be registered to the instance of the UITableView class they are meant to be reused in using the `register(_:forCellReuseIdentifier:)` function. [5] After registering the cell classes, they can be requested from the UITableView instance by using the `dequeueReusableCell(withIdentifier:)` function. [4]

Listing 2 shows an example on how to register cell classes to be reused in a specific UITableView instance.

```
  self._tableView.register(ReusingCells_TextCell.self, forCellReuseIdentifier:
↪   "ReusingCells_TextCell")
  self._tableView.register(ReusingCells_MediaCell.self, forCellReuseIdentifier:
↪   "ReusingCells_MediaCell")
```

Listing 2 Registering cells to be reused on a specific UITableView instance

Listing 3 shows an example on how to query reusable cells in the same UITableView instance.

```
  func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let message = self._messages[indexPath.row]
    let cellIdentifier = (message.mediaItems == nil) ? "ReusingCells_TextCell" :
↪   "ReusingCells_MediaCell"
    let cell = self._tableView.dequeueReusableCell(withIdentifier: cellIdentifier) as! ReusableCell
    cell.updateFromMessage(message: message)
    return cell
  }
```

Listing 3 Using recycled cells in the test project

5.2 Making static cell layouts

The idea behind this possible performance optimization was to use different view classes for each of the eight different message types (see listing 4), instead of the two mutating view classes used currently. The current two view classes, text message view and media message view, adjust their layouts according to the type of message they are going to display.

- Inbound text message view;
- Outbound text message view;
- Inbound media item view (1 media item);
- Outbound media item view (1 media items);
- Inbound media item view (2 media items);
- Outbound media item view (2 media items);
- Inbound media item view (3 media items);
- Outbound media item view (3 media items).

Listing 4 List of necessary cell views to make static cell layouts possible

5.2.1 Theoretical benefits

The theoretical performance enhancing effect of this optimization was expected to be caused by the elimination of the overhead of creating and disposing of both subviews and layout constraints. This optimization complements the previous one, as they both work towards the same goal of removing unnecessary view creations, mutations and deletions. While the previous optimization only recycled the larger parent (cell) views, this change was expected to also optimize all of the inner views and constraints.

5.2.2 Performance changes

Running the performance tests yielded the results shown in Figure 5.

2 - Static views	#1	#2	Average	Change (vs 1)
Duration (sec)	88.9	91.6	90.3	-2.4
FPS (avg)	40.7	40.4	40.6	5.7
FPS stability (%)	33.30%	31.50%	32.40%	6.40%
Average CPU usage (%)	58.38%	59.71%	59.05%	2.41%
Average GPU usage (%)	65.18%	63.58%	64.38%	1.57%
Average memory usage	32.0	33.0	32.5	1.5

Figure 5 Performance test results after making cell layouts static

The optimization had a strong effect on the average frame rate which increased by 5.7 frames per second. The frame rate stability saw an increase of 6.4 percent. The changes in the average processor, graphics processor and memory usage were so slight, they can be

classified as staying the same. Overall, this optimization increased the frame rate by 16% and frame rate stability by 6.4% without affecting the resource usage.

5.2.3 Code examples

The optimization of making static cell layouts also affected the code that handled the reuse of cells. After the creation of the 8 different cell views (listed in listing 4), all of the created view classes had to be registered to the UITableView instance before they could be reused. The higher number of cell view classes also made the logic of querying for reusable cells slightly more difficult.

Listing 5 shows the code used in the thesis project for registering the 8 different cell views to a UITableView instance.

```
  self._tableView.register(StaticCells_TextCell_In.self, forCellReuseIdentifier:
↪   "StaticCells_TextCell_In")
  self._tableView.register(StaticCells_TextCell_Out.self, forCellReuseIdentifier:
↪   "StaticCells_TextCell_Out")
  self._tableView.register(StaticCells_MediaCell_1_In.self, forCellReuseIdentifier:
↪   "StaticCells_MediaCell_1_In")
  self._tableView.register(StaticCells_MediaCell_1_Out.self, forCellReuseIdentifier:
↪   "StaticCells_MediaCell_1_Out")
  self._tableView.register(StaticCells_MediaCell_2_In.self, forCellReuseIdentifier:
↪   "StaticCells_MediaCell_2_In")
  self._tableView.register(StaticCells_MediaCell_2_Out.self, forCellReuseIdentifier:
↪   "StaticCells_MediaCell_2_Out")
  self._tableView.register(StaticCells_MediaCell_3_In.self, forCellReuseIdentifier:
↪   "StaticCells_MediaCell_3_In")
  self._tableView.register(StaticCells_MediaCell_3_Out.self, forCellReuseIdentifier:
↪   "StaticCells_MediaCell_3_Out")
```

Listing 5 Registering cell views with a static layout to a UITableView instance

5.3 Manual cell height calculations

By default, the cell height calculations are done automatically by the UITableView instance's underlying logic. According to literature on this topic, these automatic calculations are often slow and it's sensible to override them with manual calculations that are more

efficient. [20] [6] [23]

5.3.1 Theoretical benefits

Calculating the cells heights manually using efficient calculations was expected to improve the performance when scrolling at high speeds or when the whole UITableView is refreshed using the `reloadData()` function, which causes all cell heights to be calculated again [25].

When the calculations are done manually, it is trivial to optimize the calculations for the type of cells the application should be able to render. For example, media cells have a fixed height and calculations were not necessary in that case, as the pre-defined constant could be returned instead (after adding the defined vertical margins).

5.3.2 Performance changes

Running the performance tests yielded the results shown in Figure 6.

3 - Manual height calculations	#1	#2	Average	Change (vs 2)
Duration (sec)	88.0	90.0	89.0	-1.3
FPS (avg)	45.8	43.8	44.8	4.3
FPS stability (%)	41.50%	38.30%	39.90%	7.50%
Average CPU usage (%)	60.50%	57.12%	58.81%	-0.23%
Average GPU usage (%)	69.36%	66.52%	67.94%	3.56%
Average memory usage	32.0	33.0	32.5	0.0

Figure 6 Performance test results after migrating to manual cell height calculations

The manual cell height calculations increased the average frame rate by 4.3 frames per second and the frame rate stability by 7.5 percent, while keeping the resource usage nearly the same. This optimization was considered a success as there was a 10.6% increase in the average frame rate without affecting the resource usage.

5.3.3 Code examples

To override the cell height calculations, the function `tableView(_:heightForRowAt:)` from the UITableViewDelegate protocol has to be overwritten. An example of overriding the function, but not changing the default behavior of the UITableView is shown in listing 6.

```
func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat {
  return UITableViewAutomaticDimension
}
```

Listing 6 Function for overwriting cell height calculations from the UITableViewDelegate protocol

Calculating the height of the media message cells is trivial as the media message views have a fixed height and fixed vertical margins. Listing 7 shows how the height of the media message cells was calculated in the thesis project.

```
let message = self._messages[indexPath.row]
if let mediaItems = message.mediaItems {
  let baseHeight = (mediaItems.count == 1) ? Pairby.MessageUI.HEIGHT_MEDIA_ONE :
↪  Pairby.MessageUI.HEIGHT_MEDIA_MULTIPLE
  let verticalMarginsSum = (2 * Pairby.MessageUI.MARGIN_VERTICAL_WRAP)
  return (baseHeight + verticalMarginsSum)
}
```

Listing 7 Calculating media message views' height manually

Calculating the height of the text message cells is more complex, as the number of lines in the message content can vary. Listing 8 shows how the height of the text message view is calculated, using the message text and pre-defined vertical and horizontal margins.

```
let maximumWidth = (self._tableView.frame.size.width - (2 * Pairby.MessageUI.MARGIN_HORIZONTAL) -
↪  Pairby.MessageUI.MARGIN_HORIZONTAL_WRAP_DYNAMIC -
↪  Pairby.MessageUI.MARGIN_HORIZONTAL_WRAP_FIXED)
let verticalMarginsSum = (2 * Pairby.MessageUI.MARGIN_VERTICAL + 2 *
↪  Pairby.MessageUI.MARGIN_VERTICAL_WRAP)
let baseHeight = ceil(message.message!.heightWithConstrainedWidth(width: maximumWidth, font:
↪  Pairby.MessageUI.TEXT_FONT))
return (baseHeight + verticalMarginsSum)
```

Listing 8 Calculating text cell's height manually

5.4 Caching cell heights

The previous optimization of using efficient manual cell height calculations resulted in a noticeable increase in performance. Reducing the number of calculations altogether should therefore also result in a boost in performance. Reducing the number of calculations was expected to be possible by caching the calculation results in memory and only doing the calculations if no cached value could be found.

5.4.1 Theoretical benefits

Caching the calculation results was expected to reduce the number of calculations, and through that strain the processor less, when the user scrolls the same content multiple times. This optimization would not provide any performance benefits the first time the user opens or scrolls the list view, as the in-memory cache of the calculated cell heights would be empty. When the user reaches content they have already seen, the calculation should theoretically not be run and the cached value used instead.

5.4.2 Performance changes

Running the performance tests yielded the results shown in Figure 7.

4 - Cell height caching	#1	#2	Average	Change (vs 3)
Duration (sec)	87.2	88.0	87.6	-1.4
FPS (avg)	49.8	47.5	48.7	3.9
FPS stability (%)	53.10%	40.60%	46.85%	6.95%
Average CPU usage (%)	55.18%	58.62%	56.90%	-1.91%
Average GPU usage (%)	70.14%	68.81%	69.48%	1.54%
Average memory usage	32.0	34.0	33.0	0.5

Figure 7 Performance test results after the implementation of caching manually calculated cell heights

As expected, the performance did improve. The average frame rate increased by 3.9 frames per second and the frame rate stability by 6.95%. The average processor, graphics processor and memory usages, similarly to previous tests, remained nearly the same. Average device resource usages stayed nearly the same. The implementation of caching manually calculated cell heights improved the user interface responsiveness while not affecting the resource

usages.

5.4.3 Code examples

The cache was a Dictionary, in which the key was the cell's row index and the value was the height as a CGFloat. Listing 9 shows an example of the whole cycle of calculating the cell heights manually, storing and using the cached value. The manual calculations have been omitted from this listing, they can be found in subsection 5.3.

```
    private var _heightCache: [Int: CGFloat] = [:]

    func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat {
        if let height = self._heightCache[indexPath.row] { return height }
        let message = self._messages[indexPath.row]

        if let mediaItems = message.mediaItems {
            let Height = (mediaItems.count == 1) ? Pairby.MessageUI.HEIGHT_MEDIA_ONE :
↪   Pairby.MessageUI.HEIGHT_MEDIA_MULTIPLE
            let Margins = (2 * Pairby.MessageUI.MARGIN_VERTICAL_WRAP)
            self._heightCache[indexPath.row] = (Height + Margins)
            return self._heightCache[indexPath.row]!
        }

        let MaxWidth = (self._tableView.frame.size.width - (2 * Pairby.MessageUI.MARGIN_HORIZONTAL) -
↪   Pairby.MessageUI.MARGIN_HORIZONTAL_WRAP_DYNAMIC -
↪   Pairby.MessageUI.MARGIN_HORIZONTAL_WRAP_FIXED)
        let Margins = (2 * Pairby.MessageUI.MARGIN_VERTICAL + 2 *
↪   Pairby.MessageUI.MARGIN_VERTICAL_WRAP)
        let Height = ceil(message.message!.heightWithConstrainedWidth(width: MaxWidth, font:
↪   Pairby.MessageUI.TEXT_FONT))
        self._heightCache[indexPath.row] = (Height + Margins)
        return self._heightCache[indexPath.row]!
    }
```

Listing 9 Using a Dictionary to store and use manually calculated cell heights for caching purposes

5.5 Optimizing image sizes

As described in the subsection 1.2, the test data contained about 25% of media messages, each of which contained 1 to 3 images (static, animated or both). As media messages

account for a large portion of the total number of messages in the test data set, it was decided to explore what effect resizing the images to their correct size before displaying them in the UI would have.

PINRemoteImage, a library described in detail in subsection 3.3, provided the functionality of image processing and caching of processed images. That functionality was only available for non-animated images. It was decided to not optimize the animated images optimized and let the FLAnimatedImageView (see subsection 3.5) handle them as they are.

5.5.1 Theoretical benefits

The purpose of pre-processing static images is to make the images the correct size before they reach the user interface. The removed need of up- or downscaling of static images, once they have reached the user interface, was expected to reduce strain on the GPU while improving the average frames per second and frame rate stability.

5.5.2 Performance changes

Running the performance tests yielded the results shown in Figure 8.

5 - Image optimizations	#1	#2	Average	Change (vs 4)
Duration (sec)	86.8	88.4	87.6	0.0
FPS (avg)	51.1	49.5	50.3	1.7
FPS stability (%)	80.70%	74.00%	77.35%	30.50%
Average CPU usage (%)	60.92%	62.21%	61.57%	4.67%
Average GPU usage (%)	70.55%	68.09%	69.32%	-0.15%
Average memory usage	76.0	80.0	78.0	45.0

Figure 8 Performance test results after after implementing resizing of images

The improvement of 1.7 frames per second in the average frame rate was not noticeable. However, massive improvements were recorded in the frame rate stability parameter, as the frame rate stability increased by 30.5%. The average processor and graphics processor usages did not display a noticeable change. The memory usage, however, changed dramatically. The average memory usage increased by 136%, from 33 megabytes to 78 megabytes, compared to the previous test. This increase is explainable by the fact that the image processing uses a lot of memory and the application had to start a lot of image processing

processes when the scroll speed was nearing its maximum.

5.5.3 Code examples

Listing 10 shows how the image was resized to a given size, using iOS' built-in classes. The custom function accepted the original UIImage object and the expected size as the parameters.

```
  func processImage(image: UIImage, size: CGFloat) -> UIImage {
    let scaledSize = (size * UIScreen.main.scale)
    let imageRect = CGRect(x: 0, y: 0, width: scaledSize, height: scaledSize)

    UIGraphicsBeginImageContext(imageRect.size)

    let sizeMultiplier: CGFloat = (scaledSize / image.size.width)

    var drawRect = CGRect(x: 0, y: 0, width: image.size.width * sizeMultiplier, height:
↪   image.size.height * sizeMultiplier)
    if (drawRect.maxX > imageRect.maxX) { drawRect.origin.x -= (drawRect.maxX - imageRect.maxX) / 2
↪   }
    if (drawRect.maxY > imageRect.maxY) { drawRect.origin.y -= (drawRect.maxY - imageRect.maxY) / 2
↪   }

    image.draw(in: drawRect)

    let processedImage = UIGraphicsGetImageFromCurrentImageContext()
    UIGraphicsEndImageContext()

    return processedImage!
  }
```

Listing 10 Resizing an UIImage to a specified size

5.6 Rasterization of cell layers

Rasterization in the iOS view rendering context means turning a view into a cacheable bitmap that can be used in later rendering more efficiently. Rasterization of views is a built in feature of the CALayer class. Every view has a CALayer instance, accessible through the view object's `layer` property. That property is inherited from the UIView base class, that all views extend.

5.6.1 Theoretical benefits

All views, except the ones containing animated images, are static once rendered and will not change in the user interface. Having a static view is one of the requirements to reap benefits from layer rasterization. Rasterization process consumes more resources and time than regular rendering when it's first triggered. After the view has been drawn once, the view is handled much more efficiently by the rendering engine due to the fact that the graphics processor can use the cached bitmap instead of re-drawing the view. [8]

5.6.2 Performance changes

Running the performance tests yielded the results shown in Figure 9.

6.1 - Rasterizing cell layers	#1	#2	Average	Change (vs 5)
Duration (sec)	87.6	87.2	87.4	-0.2
FPS (avg)	47.0	48.0	47.5	-2.8
FPS stability (%)	43.10%	52.20%	47.65%	-29.70%
Average CPU usage (%)	60.30%	62.20%	61.25%	-0.31%
Average GPU usage (%)	63.25%	64.30%	63.78%	-5.55%
Average memory usage	73.0	77.0	75.0	-3.0

Figure 9 Performance test results after enabling rasterization of the views

The rasterization of cell layers did not result in expected performance improvements in any of the user interface related metrics. The average frame rate decreased by 2.8 frames per second and the frame rate stability went down by nearly 30%.

Analyzing and explaining the unexpected decrease in performance is done in subsection 5.7.

5.6.3 Code examples

Enabling the rasterization of cell views required the addition of two lines of code. Listing 11 shows the added two lines, where the first one enabled rasterization and the second one forced the rasterization mechanism to use scaling to compensate for the use of retina screens on some devices.

```
self.view.layer.shouldRasterize = true
self.view.layer.rasterizationScale = UIScreen.main.scale
```

Listing 11 Rasterizing views

5.7 Avoiding unnecessary offscreen rendering

Offscreen rendering means drawing the view into a new buffer (bitmap cache) which is offscreen (not on the screen) before drawing that buffer on the screen. The conventional way of drawing views is to draw them subview by subview directly onto the screen. [8]

Offscreen rendering consumes more resources and is slower than conventional rendering, but it can be beneficial to performance as the resulting bitmap can be cached and reused. The caching and reusing will only improve performance when the view does not change often (remains the same for at least a second). The ideal case is be a view that never changes (is immutable). If a view were to change more than a few times per second, for example a view containing an animated image (GIF), then the drawing to bitmap has to be done every time the animated image's frame changes and none of the cached buffers could be reused. [8]

Layer rasterization is one example of manually forcing offscreen rendering of a view. Offscreen rendering can also be triggered automatically by Core Animation. Offscreen rendering would happen, for example, when a mask is directly or indirectly applied to a layer. Offscreen rendering will in turn, as stated earlier, put unnecessary burden on the GPU. [8]

To benefit from the rasterization of layers user interface responsiveness wise, all automatic direct or indirect causes of offscreen rendering should be avoided. The automatic causes of offscreen rendering are theoretically why the previous optimization, in subsection 5.6, did not work: some, currently unknown, indirect causes of offscreen rendering were invalidating the rasterization cached bitmap. Indirect causes of offscreen rendering are, for example, applying shadows and corner radiuses to a layer. [8] Applying corner radiuses is currently done for all message views to create a chat bubble effect.

5.7.1 Theoretical benefits

The results of avoiding unnecessary offscreen rendering were expected to be reduced device resource usage and an increase in both frame rate and frame rate stability. The theoretical causes of the improvement are the same as described previous optimization, in subsection 5.6. This optimization tried to address the theoretical issue of the cached bitmap being invalidated due to offscreen rendering being triggered by automatic and non-direct causes. [8]

5.7.2 Performance changes

Running the performance tests yielded the results shown in Figure 10.

6.2 - Avoiding offscreen rendering	#1	#2	Average	Change (vs 5)
Duration (sec)	87.6	87.2	87.4	-0.2
FPS (avg)	53.5	53.2	53.4	3.1
FPS stability (%)	87.20%	87.00%	87.10%	9.75%
Average CPU usage (%)	56.91%	58.79%	57.85%	-3.72%
Average GPU usage (%)	40.31%	40.03%	40.17%	-29.15%
Average memory usage	101.0	102.0	101.5	23.5

Figure 10 Performance test results after removing unnecessary offscreen rendering

Avoiding unnecessary offscreen rendering proved to be beneficial to performance and graphics processor usage. The average frame rate increased by 3.1 frames per second and the frame rate stability increased by nearly 10%. While the average processor usage did not change noticeably (-3%), the average graphics processor usage displayed a large decrease of 29%. The average memory usage, however, increased by a whopping 30% (from 78 megabytes to 101.5 megabytes).

5.7.3 Code examples

To work around using CALayer's corner radius when drawing text cell views, the background with a corner radius was drawn manually. The manual drawing of the rounded cornered background was done by creating a custom wrapper view for the text that was able draw the rounded corner background efficiently. Listing 12 shows the code for the text message view class with the manually drawn background that avoids automatically triggering offscreen rendering. The listing also shows that the background color is customizable.

```
class NoOffscreenRendering_TextCell_LabelWrapView: UIView {
  private var _fillColor: UIColor?

  override init(frame: CGRect) {
    super.init(frame: frame)
    self.backgroundColor = Pairby.Colors.GRAY_BG
  }

  convenience init() {
    self.init(frame: CGRect.zero)
  }

  required init(coder aDecoder: NSCoder) {
    fatalError("This class does not support NSCoding")
  }

  func fillWith(color: UIColor) {
    self._fillColor = color
    self.setNeedsDisplay()
  }

  override func draw(_ rect: CGRect) {
    UIBezierPath(roundedRect: rect, cornerRadius: 15.0).addClip()
    self._fillColor?.setFill()
    UIBezierPath(rect: rect).fill()
  }
}
```

Listing 12 Custom view for displaying text messages with rounded corners that avoid automatically triggering offscreen rendering

5.8 AsyncDisplayKit

AsyncDisplayKit is an iOS framework built on top of UIKit that keeps even the most complex user interfaces smooth and responsive. [9] The library's goal was to make even the most complex views have a 60 frames per second rendering in iOS applications by moving the rendering off the main thread.

To test AsyncDisplayKit's base performance, a replica of the best case scenario view was built, identical in user interface design to the one built with UIKit (see subsection 4.1). As

was the case with the UIKit's best case performance, all media messages were replaced with text messages at runtime. Instead of UIKit classes, AsyncDisplayKit's classes and methodologies were used.

5.8.1 Theoretical benefits

The performance test results were expected to be similar to that of the UIKit's best case scenario (see subsection 4.1). The user interface responsiveness metrics, namely the average frame rate and frame rate stability, were expected to be the same as the values measured in best case scenario test and the device resource usages were expected to be similar to the values measured in the best case scenario test.

5.8.2 Performance changes

Running the performance tests yielded the results shown in Figure 11.

AsyncDisplayKit	#1	#2	Average	Change (vs 0.1)
Duration (sec)	80.8	79.8	80.3	13.9
FPS (avg)	59.0	58.0	58.5	-1.5
FPS stability (%)	98.00%	97.00%	97.50%	-2.50%
Average CPU usage (%)	42.21%	43.44%	42.83%	26.72%
Average GPU usage (%)	25.31%	26.12%	25.72%	1.86%
Average memory usage	29.0	29.0	29.0	-2.0

Figure 11 Performance test results using AsyncDisplayKit's classes and methodologies

Not only was the performance noticeably worse compared to the UIKit's best case scenario, the time it took to open the view initially was unacceptable (more than 10 seconds to load all the AsyncDisplayKit's classes and views). UIKit's best case scenario view opened instantly, with no main thread holdups.

With respect to scrolling performance, the scrollable view built using AsyncDisplayKit performed much worse than the UITableView built in the best case scenario section. The average frame rate did not meet the 60 frames per second expectation. The frame rate stability was also not a solid 100%. The actual recorded values were 58.5 frames per second and 97.5% respectively. CPU usage was 26% higher than that of UIKit's best case scenario.

The recorded average graphics processor and memory usages did not change notably.

Due to the test results being so poor, it was decided not to continue with using AsyncDisplayKit for any further tests.

5.8.3 Code examples

Listing 13 shows the code for the creation and configuration of cells, used in the AsyncDisplayKit's equivalent of UITableView, called ASTableNode.

```
  func tableNode(_ tableNode: ASTableNode, nodeBlockForRowAt indexPath: IndexPath) ->
↪   ASCellNodeBlock {
    let message = self._messages[indexPath.row]
    return {
      let cell = ASTextCellNode()
      cell.text = (message.message != nil) ? message.message! : "Media message with ID
↪   [\(message.id)]"
      return cell
    }
  }
```

Listing 13 Creation and use of AsyncDisplayKit's cell views

6 Findings

The final result, or fully optimized project, was derived frm applying all the optimizations with performance enhancing results to the worst case scenario (see subsection 4.2). Listing 14 shows the list of all applied optimizations.

- Reuse of cells (subsection 5.1);
- Making static cell layouts (subsection 5.2);
- Manual height calculations (subsection 5.3);
- Caching of cell heights (subsection 5.4);
- Optimizing image sizes (subsection 5.5);
- Avoiding unnecessary off-screen rendering (subsection 5.7).

Listing 14 List of optimizations applied to worst case scenario to create the fully optimized project

The following subsections will benchmark the final result against the best and worst case scenarios.

6.1 Performance relative to worst case scenario

To better benchmark the final result against the worst case scenario, a table comparing the worst case scenario results from subsection 4.2 against the results from subsection 5.7 was created and is displayed below, as Figure 12.

	Worst case	End result	Change	Change (%)
Duration (sec)	105.5	87.4	-18.1	-17.12%
FPS (avg)	26.0	53.4	27.4	105.19%
FPS stability (%)	23.60%	87.10%	63.50%	269.07%
Average CPU usage (%)	69.41%	57.85%	-11.56%	-16.65%
Average GPU usage (%)	65.77%	40.17%	-25.60%	-38.92%
Average memory usage	28.0	101.5	73.5	262.50%

Figure 12 Worst case scenario results benchmarked against the fully optimized project's test results

The duration of the test decreased from 105.5 seconds to 87.4, which is a 17% decrease. The decrease is explainable by the increased user interface performance (better frame rate and frame rate stability), as the processors could finish rendering all the necessary frames faster.

The average frame rate increased from 26.0 to 53.4, which is a tremendous 105% increase, effectively doubling the number of frames each second. The cap and goal of 60 frames per second was not achieved.

The frame rate stability improved more than 3.6 times, from the starting value of 23.6% to 87.1%. This custom measurement point effectively showed that not only did the average frame rate increase, but also the time spent within 20 percent of that average also increased by a large margin.

Average processor and graphics processor usages both decreased noticeably. Average processor usage decreased from 69.4% to 57.8%, which is a 11% decrease (comparative decrease of 16%). Graphics processor's average usage decreased from 65.7% to 40.1% (comparative decrease of 38.9%).

Average memory usage increased by 2.6 times, from an average of 28 megabytes to 101.5 megabytes. The main reason for this increase was image processing. The biggest increase in average memory usage can be traced to subsection 5.5, where image optimizations were introduced as an optimization. The mentioned optimizations increased average frame rate

and frame rate stability while sacrificing memory usage. The increase in memory usage is temporary, while the processing of images is first done. After processing the images, the bitmaps will be cached and memory usage will decrease.

To conclude the comparison between the worst case scenario and the final result, it can be said that the optimizations proved beneficial to the performance of the user interface. The average frame rate more than doubled and the frame rate stability almost quadrupled. Average usages of both the central and the graphics processor decreased. Memory usage did increase by a large margin, but as the increase is expected to be temporary and will not affect the application's memory usage for the whole duration of the application's life, its importance can be disregarded.

6.2 Performance relative to best case scenario

To benchmark the final result against the goal, which was the best case scenario (see subsection 4.1), a table comparing the results from both the fully optimized project and the best case scenario was created. The created table, which also contains comparative percentages, is displayed below, as Figure 13.

	Best case	End result	Change	Change (%)
Duration (sec)	66.5	87.4	21.0	31.53%
FPS (avg)	60.0	53.4	-6.7	-11.08%
FPS stability (%)	100.00%	87.10%	-12.90%	-12.90%
Average CPU usage (%)	16.11%	57.85%	41.75%	259.21%
Average GPU usage (%)	23.86%	40.17%	16.31%	68.36%
Average memory usage	31.0	101.5	70.5	227.42%

Figure 13 Best case scenario results benchmarked against the end result's test results

The test duration of the final result was 21 seconds longer than that of the best case scenario's. As stated in subsection 6.1, this can be explained by the changes in performance. When the necessary frames are rendered faster, the duration will be less.

The best case scenario's average frame rate confirmed that the cap is 60 frames per second and it is possible to achieve it with views that have little complexity. The fully optimized project's results did not display the same, highest possible, average frame rate. Instead, the

tests recorded the average frame rate to be 53.4 frames per second.

The frame rate stability for the best case scenario showed that a stability of 100% is possible. The final result did not manage to reach the 100 percent. Instead, the tests recorded a result of 87.1%. As explained previously, in subsection 2.2, the frame rate stability showed that 87.1% of the time was spent within 20 percent of the average frame rate, which was 53.4 frames per second.

Given the measured average frame rate of the fully optimized project, which was 53.4 frames per second, and the assumption that the screen input lag is around 55 milliseconds (see subsection 2.4), the user interface response time can be calculated to be an average of 73.7 milliseconds. As stated in the introduction, a user tolerance's upper limit, before they assume the response to not be instantaneous, is 100 milliseconds. The calculated user interface response value of 73.7 milliseconds fits well within the envelope of instantaneous response times and the measured frame rate stability value of 87.1% also guarantees that the response time stays in the instantaneous range for the majority of the time.

Benchmarking the recorded best case scenario's average central processor and graphics processor usages against the respective recorded values from the fully optimized project displayed large increases in both. The recorded average CPU and GPU showed an increase from 16% to 57.8% and 23.8% to 40% respectively. The increase in average processors' usages can be explained by the best case scenario having views with very little complexity, as described in subsection 4.1. Handling views with more complexity, especially images, is expected to use more resources.

The average memory usage increased from 31 to 101 megabytes. The large, yet temporary (see subsection 6.1), increase is caused by two optimizations described in subsection 5.5 and subsection 5.7. The two mentioned optimizations are image processing related.

Summary

The goal of the thesis was to explore and document different ways of improving the Pairby iOS application's scrollable views' user interface performance. In total, 8 different optimization methods were theorized, realized, benchmarked and analyzed. Of the 8 theorized optimizations, 6 had the expected performance enhancing effect, meaning the optimizations improved either the frame rate, frame rate stability or both. The remaining 2 of the 8 theorized optimizations did not provide the expected performance enhancing benefits, but are still included in this thesis, as the reasons for the optimization methods failure either became a basis for the next optimizations or otherwise made an introduction to the next logical and structural step in the optimization process.

Before the exploration of the optimization methods could begin, specifications to a user interface similar to that of Pairby applications' had to defined and a way to measure user interface responsiveness had to be found.

Section 1 described, in depth, how the user interface was designed to look like the Pairby applications' message list view, as well as defining some data layer requirements. The more important requirements for the user interface were: distinguishing between text and media messages, text message content varying in length and number of lines, media messages containing either one, two or three images, images in the media messages being either static, animated or both.

The question of how to measure performance was addressed in section 2. The section described how a repeatable test case was created and what, as well as how, the metrics representing user interface responsiveness were measured. The repeatable test case, simulating user interaction with the user interface (scrolling), was created programmatically. A total of

6 metrics were measured for each test case (see listing 1). User interface responsiveness was directly related to the average frame rate and the frame rate stability. The latter of which was defined in subsection 2.2 as "percentage of time spent within 20% of the average frame rate". Three metrics were related to the device resource usage, which were expected to give an overview of how the hardware usage reacts to different optimizations. The mentioned device resource usage measurement points were average central processor, graphics processor and memory usages. The sixth, currently unmentioned, measurement point was test duration, which, by itself, did not convey enough information to draw any conclusions from, but a correlation between the test duration and average frame rate and frame rate stability could later be identified.

The practical part of the thesis started by creating two reference points called "worst case scenario" and "best case scenario". The goal of the reference points was to represent the worst possible and the best possible user interface performance. The worst case scenario became the starting point for all optimizations while the 60 frame per second performance recorded in best case scenario view became the ultimate goal.

The process of exploring different optimizations was split into sections, each covering one specific optimization technique. Each of the sections had three subsections, in addition to the short introduction: theoretical benefits, performance changes and code examples. The three subsection structure allowed each technique to be explained in adequate depth in a similar manner across all sections.

As mentioned in the introductory paragraph of this section, a total of 8 optimization techniques were presented in the thesis. 6 of the optimizations improved user interface responsiveness and 2 of the optimizations did not. To draw conclusions from done work, the final result, which was the result of applying all performance enhancing optimizations to the worst case scenario, was benchmarked against the two previously described reference points.

Benchmarking the fully optimized project, or the final result, against the worst case scenario

showed huge improvements in performance. The average frame rate increased by 105% and ended up at a measured value of 53.4 frames per second. The frame rate stability increased by more than 3.6 times, from 23.6% to 87.1%. The average central processor and graphics processor usages decreased by 11.5% and 25.6% respectively. The respective values for the average CPU and GPU usages measured for the final result were 57.9% and 40.2%. The average memory usage showed an increase from 28.0 megabytes to 101.5 megabytes. That increase can be explained by the addition of image processing, which improved average frame rates and frame rate stability at the cost of temporarily using more memory (see subsection 6.1). When comparing the fully optimized project's test results to those of the worst case scenario's, all the performance related parameters increased by more than two times while the average CPU and GPU usage decreased and the memory usage showed a temporary increase.

The final result did not achieve the user interface performance numbers measured and set as a goal in the best case scenario tests (see subsection 4.1). The best case scenario showed that an average frame rate of 60 frames per second and a frame rate stability of 100% are the best possible performance numbers. The fully optimized test project's results showed an average frame rate of 53.4 frames per second and a frame rate stability of 87.1%. Both of the two user interface responsiveness related metrics came close to 90% of the ultimate goal.

Based on the average frame rate of the fully optimized project, which was 53.4 frames per second, and the response time of the test device's screen (see subsection 2.4), the average user interface response time could be calculated to be 73.7 milliseconds. As concluded in subsection 6.2, the calculated value of 73.7 milliseconds fits well within the envelope of instantaneous response times and the measured frame rate stability value of 87.1% also guarantees that the response time stays in the instantaneous range for the majority of the time.

The product of the thesis is a documentation of different optimization methods that can be applied to iOS applications' views that contain a UITableView, including the vertically

scrollable views inside the Pairby iOS application. The creation of this documentation was the goal stated in the introduction section and it was fulfilled.

Kokkuvõte

UITableView optimeerimine Pairby iOS mobiilirakenduses

Lõputöö eesmärgiks oli leida ja dokumenteerida erinevaid viise Pairby iOS rakenduses olevate vertikaalselt keritavate kasutajaliidese komponentide töökiiruse parendamiseks. Kokku teoretiseeriti, realiseeriti, mõõdeti, võrreldi ja analüüsiti kaheksat erinevat viisi töökiiruse parendamiseks. Nendest kaheksast optimeerimise viisist olid kuus oodatud, töökiirust parendava, mõjuga. Töökiirust parendava mõju all peetakse silmas kasutajaliidese kaadrisageduse, kaadrisageduse stabiilsuse või mõlema näitaja suurenemist. Teised kaks, kaheksast teoretiseeritud parendusviisist, ei omanud töökiirusele soovitud, töökiirust parendavat, mõju, kuid on lõputöös siiski detailselt kajastatud, kuna nende mitte-töötamise põhjused olid aluseks järgmistele parendusviisidele või muudele olulistele järeldustele.

Enne optimeerimisviiside otsimise alustamist loodi kogum spetsifikatsioonidest, mis määratlesid Pairby mobiilirakendustega sarnaneva kasutajaliidese kui ka andmekihi. Lisaks leiti viis kasutajaliidese töökiiruse mõõtmiseks.

Peatükk 1 kirjeldab kuidas Pairby mobiilirakenduste sõnumite vaatega sarnane kasutajaliides planeeriti. Lisaks on samas peatükis defineeritud mitmed andmekihi nõuded. Mõned olulisemad kasutajaliidese nõuded olid näiteks: teksti- ja meediasõnumite eristamine, tekstisõnumite sisuks oleva teksti varieerumine pikkuselt ja ridade arvult, meediasõnumite sisu (üks kuni kolm pilti), meediasõnumite sisuks olevate piltide tüübid (staatilised, animeeritud või segamini).

Töökiiruse mõõtmise viis defineeriti peatükis 2. Nimetatud peatükk kirjeldas reprodutseeritava testjuhi loomist ning milliseid mõõtepunkte vaadeldi, et saada ettekujutus kasutajaliidese töökiirusest. Reprodutseeritav testjuht, mille eesmärgiks oli simuleerida kasutaja interaktsiooni kasutajaliidesega (kerimist), loodi programmaatiliselt. Kokku vaadeldi kuut erinevat mõõtepunkti (vt nimekiri 1). Kasutajaliidese töökiirus on otseselt seotud kahe mõõtepunktiga, nimelt keskmise kaadrisageduse ja kaadrisageduse stabiilsusega. Kaadrisageduse stabiilsus on defineeritud alapeatükis 2.2 kui "protsent ajast, mil kaadrisagedus oli keskmisest kaadrisagedusest kuni 20% erinev". Kolm mõõtepunkti olid seotud seadme ressursikasutusega, millede oodatud eesmärgiks oli anda ülevaade töökiiruse parendusviiside mõjust seadme ressurside kasutusele. Nimetatud, seadme ressursikasutusega seonduvad mõõtepunktid, olid keskmised protsessori, graafikaprotsessori ja mälu kasutused. Kuues, veel nimetamata, mõõtepunkt oli testile kuluv aeg, mis iseseisvalt ei anna piisavalt informatsiooni, et järeldusi teha. Küll aga koos keskmise kaadrisageduse ja kaadrisageduse stabiilsusega võis märgata aja, kaadrisageduse ja kaadrisageduse stabiilsuse vahelist korrelatsiooni.

Lõputöö praktiline osa algas kahe võrdluspunkti loomisega. Loodud võrdluspunktidele anti nimed "halvim testjuht"ja "parim testjuht". Võrdluspunktide eesmärkideks oli leida ja mõõta halvim võimalik ja parim võimalik kasutajaliidese töökiirus. Halvim testjuht sai ka kõikide töökiiruse parendusviiside alguspunktiks. Maksimaalset kaadrisagedust (60 kaadri sekundis) näidanud parim testjuht võeti kasutusele ülimaks kasutajaliidese töökiiruse eesmärgiks.

Erinevate töökiiruse parenduste leidmine jagati alapeatükkidesse, kus iga alapeatükk kattis ühte spetsiifiliset töökiiruse parendusviisi. Iga alapeatükk koosnes omakorda kolmest osast, lisaks lühikese sissejuhatusele: teoreetiline kasu töökiirusele, muutused kasutajaliidese töökiiruses ja koodinäited. Nimetatud kolm alapeatüki osa võimaldasid igat töökiiruse parendust kirjeldada adekvaatse täpsusega ning igas parendusviisile keskenduvas alapeatükis sarnasel moel.

Nagu mainitud selle peatüki sissejuhatavas lõigus, on lõputöös esitatud kokku kaheksa teoreetilist töökiiruse parendamise viisi. Kaheksast lõputöös esitatud optimeerimisviisist kuus parendasid töökiirust ja kaks ei parendanud. Tehtud töö põhjal lõplike järelduste tegemiseks

loodi niinimetatud lõpptulemust esindav projekt, ehk lõpptulemus, mis saadi kõikide töökiirust parendavate optimeerimisviiside lisamisel halvima testjuhu projektile. Saadud lõpptulemust võrreldi nii parima kui ka halvima testjuhuga.

Lõpptulemust esindav projekt ehk lõpptulemus näitas võrreldes halvima testjuhuga suuri parendusi töökiiruses. Keskmine kaadrisagedus tõusis 105%, tõustes mõõdetud väärtuseni 53,4 kaadrit sekundis. Kaadrisageduse stabiilsus suurenes rohkem kui 3,6 korda, tõustes algväärtusest 23,6% lõppväärtuseni 87,1%. Keskmised protsessori ja graafikaprotsessori kasutused kahanesid vastavalt 11.5% ja 25.6%. Vastavad mõõdetud väärtused keskmise protsessori ja graafikaprotsessori kasutuse kohta olid lõpptulemuse puhul vastavalt 57.9% ja 40.2%. Keskmine mälukasutus kasvas 28,0 megabaidi pealt 101,5 megabaidi peale. Seda kasvu seletab lisandunud pilditöötluse võimekus, mis parendas keskmist kaadrisagedust ja kaadrisageduse stabiilsust, kuid nõudis seadmelt ajutiselt rohkem mälu (vt peatükk 6.1). Võrreldes lõpptulemuse mõõtetulemusi halvima testjuhu tulemustega, võib järeldada, et kõik mõõdetud töökiirusega seonduvate mõõtepunktide väärtused kasvasid rohkem kui kahekordselt ja seadme ressursikasutust esindavate mõõtepunktide mõõdetud väärtused kahanesid, kui välja arvata keskmine mälukasutus, mis näitas ajutist kasvu.

Lõpptulemus ei saavutanud samu töökiirust kirjeldavaid mõõtepunktide väärtusi, mis olid seatud eesmärgiks parima testjuhu alapeatükis (vt peatükk 4.1). Parima testjuhu mõõtetulemused näitasid, et parimaks kasutajaliidese töökiiruseks on keskmine kaadrisagedus 60 kaadrit sekundis ja kaadrisageduse stabiilsus 100%. Kõikide töökiiruse parendustega projekti puhul mõõdeti keskmiseks kaadrisageduseks 53,4 kaadrit sekundis ja kaadrisageduse stabiilsuseks 87,1%. Mõlemad kasutajaliidese töökiirusega seotud mõõtepunktid saavutasid seatud eesmärgist umbes 90%.

Põhinedes lõpptulemuse keskmisele kaadrisagedusele, milleks oli 53,4 kaadrit sekundis, ja testimiseks kasutatud seadme ekraani reageerimisajale (vt peatükk 2.4), on võimalik arvutada kasutajaliidese reageerimisajaks 73,7 millisekundit. Nagu järeldatud peatükis 6.2, on see arvutatud väärtus, 73,7 millisekundit, kindlalt selle aja piires, kus kasutaja peab kasutajaliidese reageerimist koheseks. Lisaks tähendab 87,1%-ne kaadrisageduse stabiilsus, et

enamuse ajast püsib rakenduse kasutajaliidese reageerimise kiirus eelpool mainitud kohese reageerimise aja piires.

Lõputöö tulemuseks on dokumentatsioon erinevatest kasutajaliidese töökiiruse parenduse viisidest, mida on võimalik rakendada iOS rakenduste vaadetele, mis sisaldavad UITableView komponenti. Need töökiiruse parendused on rakendatavad ka Pairby iOS rakenduse vertikaalselt keritavatele vaadetele. Nimetatud dokumentatsiooni loomine oli, nagu sissejuhatuses kirjeldatud, lõputöö eesmärgiks ja see sai täidetud.

List of Figures

References

[1] Apigee. Apigee survey: Users reveal top frustrations that lead to bad mobile app reviews. Online, 2012. https://apigee.com/about/press-release/apigee-survey-users-reveal-top-frustrations-lead-bad-mobile-app-reviews, 23.02.2017.

[2] Apple. Characteristics of cell objects, 2013. https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/TableView_iPhone/TableViewCells/TableViewCells.html, 23.03.2017.

[3] Apple. ipod touch (5th generation) - technical specifications, 2017. https://support.apple.com/kb/sp657, 18.03.2017.

[4] Apple. instance method: dequeuereusablecell(withidentifier:), 2017. https://developer.apple.com/reference/uikit/uitableview/1614891-dequeuereusablecell, 23.03.2017.

[5] Apple. instance method: register(_:forcellreuseidentifier:), 2017. https://developer.apple.com/reference/uikit/uitableview/1614888-register, 23.03.2017.

[6] Chukai. How to make dynamic table view cell height, 2017. http://chuckchu.com/how-to-make-dynamic-table-view-cell-height/, 24.02.2017.

[7] Damien. Jason, 2017. https://github.com/delba/JASON, 20.02.2017.

[8] Daniel Eggert. Moving pixels onto the screen, 2013. https://www.objc.io/issues/3-views/moving-pixels-onto-the-screen/, 26.02.2017.

[9] Facebook. Introducing asyncdisplaykit, 2017. https://code.facebook.com/posts/721586784561674/introducing-asyncdisplaykit-for-smooth-and-responsive-apps-on-ios/, 23.02.2017.

[10] Flipboard. Flanimatedimageview, 2017. https://github.com/Flipboard/FLAnimatedImage, 23.02.2017.

[11] GameBench. Gamebench: The performance standard for mobile gaming, 2017. https://www.gamebench.net/, 19.03.2017.

[12] Konoma GmbH. Fpscounter: A small library to measure the frame rate of an ios application, 2017. https://github.com/konoma/fps-counter, 19.03.2017.

[13] GreenNet. Understanding file sizes, 2016. https://www.greennet.org.uk/support/understanding-file-sizes, 13.05.2017.

[14] Nielsen Norman Group. Jakob nielsen, principal, 2017. https://www.nngroup.com/people/jakob-nielsen/, 11.05.2017.

[15] Paul Hudson. How to register a cell for uitableviewcell reuse, 2016. https://www.hackingwithswift.com/example-code/uikit/how-to-register-a-cell-for-uitableviewcell-reuse, 23.03.2017.

[16] Tanous Jim. 5th gen ipod touch benchmarks reveal slower a5 processor, 2012. https://www.macobserver.com/tmo/article/5th-gen-ipod-touch-benchmarks-reveal-slower-a5-processor, 18.03.2017.

[17] Joseph Keller. iphone 5 touchscreen latency measured, found to be 2.5x faster than closest android rival, 2013. http://www.imore.com/iphone-5-touchscreen-latency-measured-found-be-25-times-faster-closest-androic 18.05.2017.

[18] Andrew Ku. The complete ipad 3 review: Retina display, a5x, 4g lte, and camera, 2012. http://www.tomshardware.com/reviews/ipad-3-benchmark-retina-display,3157-3.html, 13.05.2017.

[19] Jakob Nielsen. *Usability Engineering*. AP Professional, 1993.

[20] Alexander Orlov. Perfect smooth scrolling in uitableviews, 2015. https://medium.com/ios-os-x-development/perfect-smooth-scrolling-in-uitableviews-fd609d5275a5#.30sea87ie, 24.02.2017.

[21] Robert Payne. Snapkit, 2017. https://github.com/SnapKit/SnapKit, 20.02.2017.

[22] Pinterest. Pinremoteimage, 2017. https://github.com/pinterest/PINRemoteImage, 20.02.2017.

[23] Andrea Prearo. Smooth scrolling in uitableview and uicollectionview, 2017. https://medium.com/capital-one-developers/smooth-scrolling-in-uitableview-and-uicollectionview-a012045d77f#.113swlhhz, 26.02.2017.

[24] Dimensional Research. Failing to meet mobile app user expectations: A mobile user survey. Online, 2015.

[25] Hamish Rickerby. Updating a uitableview without calling reloaddata, 2012. https://hamishrickerby.com/2012/07/06/

updating-a-uitableview-without-calling-reloaddata/, 15.05.2017.

[26] Will Shanklin. iphone 5 vs. ipod touch 5g, 2012. http://newatlas.com/iphone-5-vs-ipod-touch-5g/24488/, 18.05.2017.

[27] Qresolve Technical Team. iphone 5 vs. ipod touch 5g, 2012. http://blog.qresolve.com/blog/2012/10/17/iphone-5-vs-ipod-touch-5g/, 13.05.2017.

[28] Peter Tsoi. Advanced touch input on ios, 2015. https://developer.apple.com/videos/play/wwdc2015/233/, 13.05.2017.

[29] Dave Wood. Xcglogger, 2017. https://github.com/DaveWoodCom/XCGLogger, 20.02.2017.