Free University of Bozen - Bolzano

Faculty of Computer Science

Master Thesis

Adjusting the Layout to the Interface Orientation in NetSnips

Author:
Johannes Erschbamer

Supervisor:
Prof. Johann Gamper

A thesis submitted in partial fulfillment of the requirements for the degree of Master in Computer Science

in the

Centre for Information and Database Systems Engineering (IDSE)

September 2013

Abstract

Developing for mobile devices poses new challenges for software design and development. One such challenge is related to the fact that mobile devices can be held in different orientations. This leads to two very common user interface idioms: portrait and landscape mode. Users expect that the scarce screen of mobile devices is optimally used and that the user interface adapts *immediately* to the actual device orientation.

This thesis proposes ARITA, a novel *repositioning algorithm* for a mobile application called NetSnips. NetSnips allows to create rectangular snippets from arbitrary web pages. These snippets can be positioned on multiple virtual pages. On device rotation, snippets might exceed the new screen limits and therefore only be partially visible or not visible at all. The ARITA algorithm automatically repositions these snippets in a way that the original snippet layout is preserved as much as possible.

This work is a first attempt to address the complex problem of automatically repositioning rectangles (i.e., snippets) in a context where *no objectively best solution* exists. ARITA works in 3 steps: First, a sweep line based algorithm identifies the largest free spaces between the snippets. Second, a heuristic is used to place the snippets that need to be repositioned in the free rectangular spaces. Third, a so-called layout graph is build which represents the snippets and their relative position. This graph is used for balancing the margins between fixed snippets, movable snippets and the screen bounds.

We have evaluated the runtime of ARITA using snippet layouts of different complexity with a varying number of snippets. The experiments show that ARITA provides good performance results in realistic layout situations for NetSnips. In order to evaluate the quality of the layout produced by ARITA, we performed a user experience evaluation with participants of the NetSnips user base. Although the judgment of the automatically generated snippet layout depends strongly on habits and preferences, the results show that the participants of our study generally validate the layout of ARITA from satisfiable to very good.

Acknowledgements

Many people helped me to successfully complete this thesis. First and foremost, I would like to thank my supervisor Professor **Johann Gamper**, for his valuable advices and support. He was always available for clarifications and explanations in his friendly manner.

A big thank goes to my parents **Edith** and **Martin**, who made my studies possible and supported me in my entire study period. Moreover I would like to thank my siblings **Franziska** and **Andreas** and my girlfriend **Judith** for always having a friendly ear and being there when I needed them.

I thank all my **fellow students**, for the great time at UniBZ with inspiring coffee talks and funny moments throughout the entire study period.

Finally I would like to send a *"Thank you!"* around the world to all **NetSnips users** that participated in the user experience evaluation. You're a great user base!

Johannes

Contents

Abstract **i**

Acknowledgements **ii**

Contents **iii**

List of Figures **v**

List of Tables **vii**

1 Introduction **1**

1.1 NetSnips . . . 2

1.2 Contributions . . . 3

1.3 Thesis Outline . . . 4

2 Related Work **6**

2.1 Two-Dimensional Packing Problems . . . 6

2.2 Constraint-Based Layout Algorithms . . . 7

2.3 Mobile Computing and Power Saving . . . 8

3 Problem Description **10**

3.1 Positioning Snippets in NetSnips . . . 10

3.2 Requirements for the Repositioning of Snippets . . . 11

4 Adjusting Rectangles on Interface Turn Algorithm **13**

4.1 Symbols and Abbreviations . . . 13

4.2 Identification of Adjustable Rectangles . . . 14

4.3 Determining Free Rectangular Spaces . . . 15

4.3.1 Processing of Active Spaces . . . 16

4.3.2 Instantiation of New Active Spaces . . . 17

4.3.3 Algorithm for Computing Free Spaces . . . 18

4.4 Placing Rectangles in Free Spaces . . . 19

4.5 The Layout Graph . . . 21

4.5.1 Definition . . . 22

4.5.2 Constructing the Layout Graph . . . 23

4.6 Balancing the Layout Graph . . . 25

4.6.1 Extracting Variable Paths . . . 26

4.6.2 Balancing Variable Paths 27
4.7 ARITA 29
4.8 Complexity Analysis 30
4.9 Adjusting Multiple Pages 31

5 Evaluation **32**
5.1 Runtime Experiments 32
5.1.1 Setup 32
5.1.2 Results 33
5.2 User Experience Evaluation 36
5.2.1 Setup 36
5.2.2 Results 38
5.2.3 Discussion and Lessons Learned 40

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

A Evaluation Questionary **46**

B Evaluation Cases **48**

List of Figures

1.1 NetSnips with arbitrarily sized snippets for 6 different web pages (left). Users can freely define the size and content of web snippets on any web page (right). 2

1.2 The user defined snippet layout in portrait interface orientation (left) and the correspondingly generated layout for landscape interface orientation (right). 3

4.1 The user defined layout in portrait mode and in landscape mode. In landscape some snippets are completely or partially hidden (highlighted in red). 15

4.2 All 7 partially overlapping free spaces (translucent colors) in respect to the fixed rectangles 1 and 2 in landscape interface orientation. 16

4.3 The *sweep line* and *negative line list* at x = 20. 17

4.4 The *sweep line* at x = 350 and *negative line list* for x = 351 in scope of rectangle 2. 18

4.5 The temporary landscape rectangle layout. Remaining *free spaces* are highlighted with translucent colors. 21

4.6 The layout graph with all nodes and edges for the rectangle layout in Figure 4.5. 23

4.7 The layout graph after initializing all nodes; movable nodes are bold. . . . 24

4.8 The layout graph after setting all west and east edges to all nodes. Variable edges are labeled and highlighted in orange. 25

4.9 The list of all horizontal *variable paths* in the layout graph with labeled edges *before* layout balancing. 27

4.10 The layout graph and the resulting rectangle layout after completing the horizontal balancing process. The path balancing was performed in the following order: *red*, *orange*, *yellow*. 29

4.11 The resulting rectangle layout in landscape interface orientation completing ARITA. 30

5.1 CPU time versus number of rectangles for ARITA with 3 different settings for adjustable rectangles. 33

5.2 CPU Time versus Number of Adjustable Rectangles for ARITA. 34

5.3 CPU time versus number of adjustable rectangles partitioned in the two main parts of ARITA. 35

5.4 CPU time versus rectangle size for ARITA. 35

5.5 CPU time versus number of pages for ARITA. 36

5.6 Average criteria rating on 20 cases. 38

5.7 Case 16. Higher ratings for *expected*; lower ratings for *overall satisfaction* (most probably due to unnecessarily high overlapping). 39
5.8 Case 12. High ratings for all criteria. 39
5.9 Case 1. Diverse ratings for *equal distribution*. 40
5.10 Average criteria rating for all cases. 40

A.1 Home page of www.netsnipsapp.com on September 5, 2013. 46
A.2 The study description for participants. 47

B.1 Case 1 48
B.2 Case 2 48
B.3 Case 3 49
B.4 Case 4 49
B.5 Case 5 49
B.6 Case 6 50
B.7 Case 7 50
B.8 Case 8 50
B.9 Case 9 51
B.10 Case 10 51
B.11 Case 11 51
B.12 Case 12 52
B.13 Case 13 52
B.14 Case 14 52
B.15 Case 15 53
B.16 Case 16 53
B.17 Case 17 53
B.18 Case 18 54
B.19 Case 19 54
B.20 Case 20 54

List of Tables

4.1 Symbols used in this chapter. 14

5.1 The parameters under investigation and their default values. 32

List of Algorithms

1 ComputeFreeSpaces 19
2 PlaceRectangles 21
3 SetWestAndEastNeighbors 24
4 WestNeighbors 26
5 HorizontalVariablePaths 27
6 ARITA 30

Chapter 1

Introduction

Applications for mobile devices offer undreamt-of possibilities how users interact with computers. The combination of user specific data, mobile internet and various real-time sensor inputs allows to build a new generation of software that is personalized, location-aware, interconnected and adapts to the current user's context.

However, developing for mobile devices poses additional challenges in software design and development. Already in the early years of mobile computing Forman and Zahorjan [1] (1994) identified various new challenges and stated that *"the shift toward mobility and wireless communication is testing the abilities of designers to adapt the abilities of traditional system structures"*. Applications for mobile devices are constraint to smaller displays, energy consumption, slower processors and limited memory. Hence resources may be scarce, network connections may be slow or error-prone and user interfaces must be very flexible, touchable and focus on the most essential parts to solve a specific problem.

In contrast to stationary computers, mobile devices may be held in different orientations. Two user interface idioms are very common, namely *portrait* mode (higher as broad) and *landscape* mode (broader as high). On most mobile devices acceleration and motion sensors detect device orientation changes automatically and request applications to adapt their user interface accordingly. Users expect that the scarce screen is used optimally and that the user interface adapts *immediately* to the current device orientation. This is especially important because on touch-based mobile devices the display serves as input and output device simultaneously.

This thesis presents a *repositioning algorithm* for NetSnips, a mobile application that allows creating rectangular snippets from arbitrary web pages. Section 1.1 describes the purpose, spreading and background of NetSnips and section 1.2 details the contributions of this thesis. Finally section 1.3 outlines the organization of this document.

1.1 NetSnips

NetSnips is an utility application that is freely available on the Apple AppStore[1] for iPhone, iPad and iPod Touch. NetSnips allows defining specific areas on *arbitrary web pages*, which are *updated automatically* in regular time intervals. This way, users can create so-called web snippets that visualize a snapshot of the latest content on the selected web pages. NetSnips may be used for very different purposes, including tracking the latest stock changes, monitoring the weather forecast or keeping track of daily comic strips.

Users can position these rectangular web snippets on multiple virtual pages in order to create their own personalized "wall of information". It is also possible to open the full web page with a single tap or force a manual snippet reloading by double-tapping on a snippet.

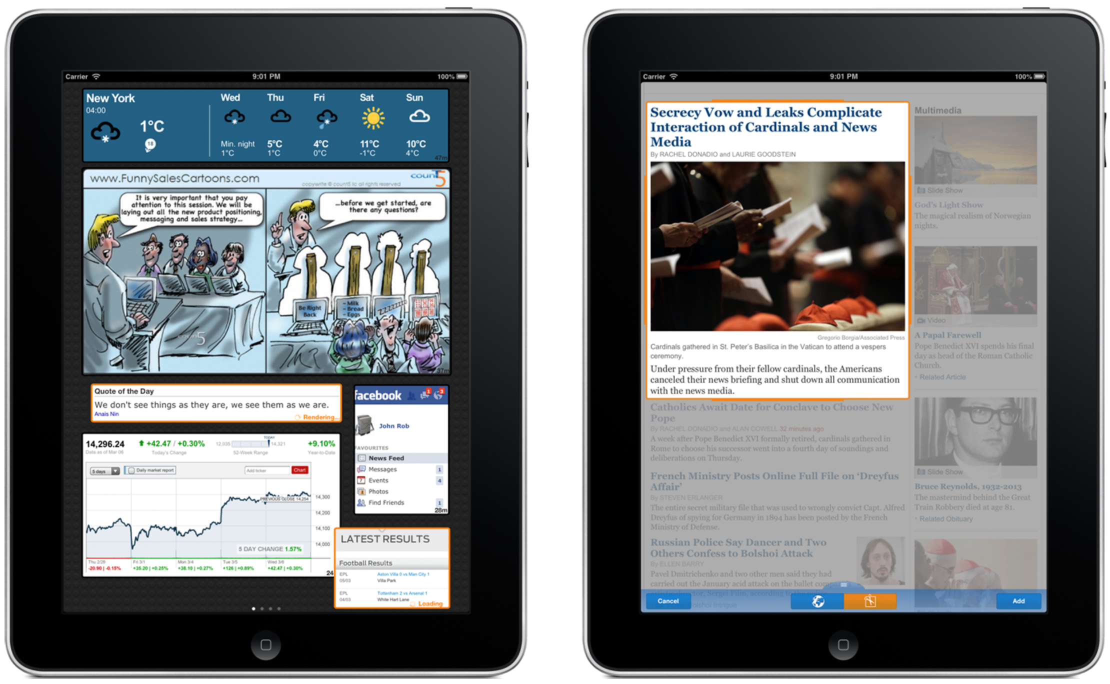

New York
Wed
Thu
Fri
Sat
Sun
www.FunnySalesCartoons.com
Quote of the Day
facebook
LATEST RESULTS
Secrecy Vow and Leaks Complicate Interaction of Cardinals and News Media
Multimedia
Cancel
Add

FIGURE 1.1: NetSnips with arbitrarily sized snippets for 6 different web pages (left). Users can freely define the size and content of web snippets on any web page (right).

NetSnips is completely localized in English, French, German, Italian, Russian and Simplified Chinese. The first version was published on June 29 in 2012 on the App Store. Until today (September 2013) it was downloaded more than 60.000 times all over the world. The countries with the most downloads are the United States (29%), China (15%), France (9%), Japan (6%) and Italy (5%). NetSnips received generally high user ratings and good press reviews for its *"unique multitasking web browser experience"*[2] and for its *"novel way to track updates for favorite web pages"*[3].

[1]http://appstore.com/netsnips
[2]http://appchronicles.com/netsnips-a-truly-unique-multitasking-web-browser-experience/
[3]http://appadvice.com/appnn/2013/03/todays-apps-gone-free-hackycat-mountain-bike-cycling-computer-envelope-hd-and-more

Users position their web snippets with absolute values in the cartesian coordinate system of a page. A device rotation to landscape mode leads to flipped dimensions for all pages, i.e., the page width becomes the page height and vice versa. Snippets that are positioned beyond the new page width and height limits will no longer be visible or partially cut off. Therefore the current version of NetSnip is limited to portrait interface mode only. In order to remove this limitation, this thesis presents a repositioning algorithm for rectangular web snippets on interface orientation changes.

1.2 Contributions

The research project presented in this thesis proposes ARITA, a *novel repositioning algorithm* that adapts the position of rectangular web snippets on changes of the user interface orientation. Figure 1.2 shows the *generated landscape snippet layout* (right) based on the *user defined portrait snippet layout* (left).

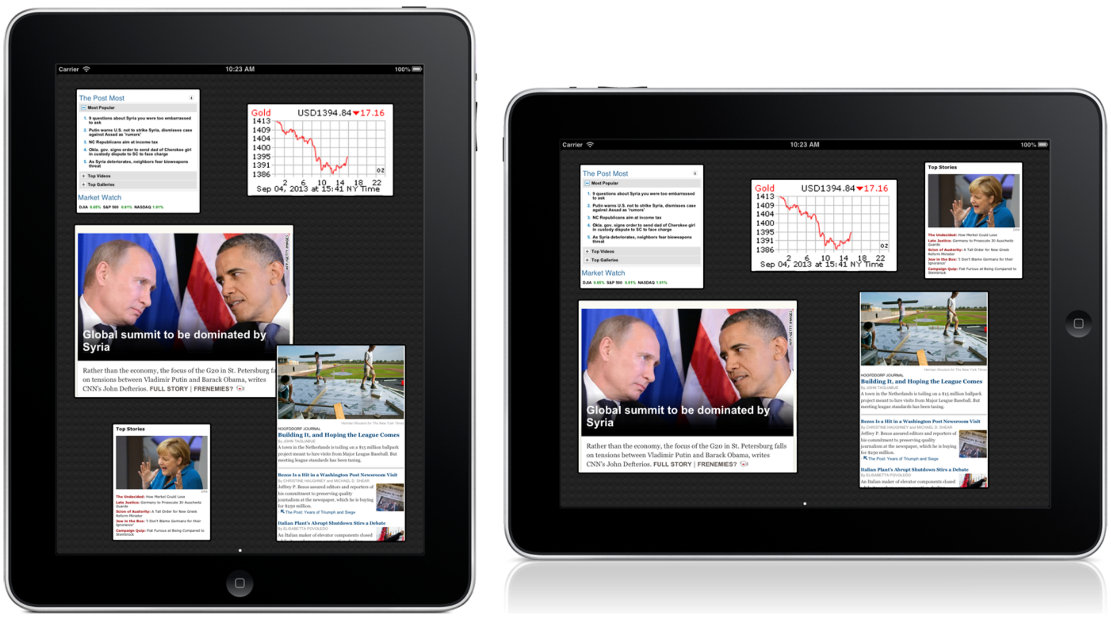


FIGURE 1.2: The user defined snippet layout in portrait interface orientation (left) and the correspondingly generated layout for landscape interface orientation (right).

The contributions of this thesis can be summarized as follows:

1. **Strategy decision**: This work is a first attempt to address the complex problem of repositioning rectangles where *no objectively best solution* exists. The judgment of the automatically generated rectangle layout depends strongly on user preferences, habits and general usage of the application. Therefore, we explored various strategies that address this problem in different ways. For instance, we experimented with solutions that reposition *all snippets* on device rotation following a specific pattern and others that try to *preserve the original user defined layout* as much as possible. After various experiments we decided to follow the latter strategy and specified our exact requirements for the repositioning algorithm.

2. **Conceptual work**: The repositioning concept was refined by selecting the foundations and data structures for the solution (e.g., sweep line technique and graph based storage of rectangle layouts). We have elaborated a sweep line based algorithm that identifies the *largest free spaces between rectangles* in a given container rectangle. We have specified a *heuristic* for placing rectangles in a set of free space candidates. Finally, we have elaborated an algorithm that builds a *graph for describing a rectangle layout* and a strategy for *balancing the horizontal and vertical margins* between fixed/variable rectangles and screen bounds.

3. **Implementation in a released mobile application**: The conceptual work resulted in the implementation of ARITA, a novel algorithm for repositioning rectangles. We have implemented ARITA in a released mobile application called NetSnips. A new version of NetSnips will benefit from the work of this thesis and will be available on the Apple App Store shortly.

4. **Runtime evaluation**: We have evaluated the runtime of ARITA for different rectangle layouts with various experiments on real devices. The experiments show that ARITA provides good performance results in realistic layout situations for NetSnips.

5. **User experience evaluation**: In order to evaluate the quality of the layout produced by ARITA, we performed a user experience evaluation with participants of the NetSnips user base. The results confirm our assumption that the assessment of automatically generated layouts depends strongly on personal preferences. Our results show also that, in general, the participants of your study validate the layout results of ARITA from satisfiable to very good.

1.3 Thesis Outline

The rest of this thesis is organized as follows:

- Chapter 2 gives an overview of related problems and algorithms. First it describes the characteristics and solutions to two-dimensional packing problems and constraint-based layout algorithms. Following that we discuss mobile computing and power saving in general.

- Chapter 3 describes the problem that was addressed in this thesis. It begins with a general discussion how the problem of repositioning snippets on interface orientation changes could be solved. It further states the requirements and constraints that we have defined for the repositioning algorithm.

- Chapter 4 describes the repositioning algorithm ARITA in detail. First it defines symbols and abbreviations that will be used to describe the concepts in this chapter. It proceeds by explaining which criteria are used for judging if a rectangle should be repositioned. Further it shows how the free spaces between rectangles are identified and which criteria is used by ARITA for to find a new location for a rectangle. It describes how we build the layout graph (a graph for describing rectangle layouts) and how we use this graph for distributing rectangles with equal margins to other rectangles and screen bounds. Finally it states how all this sub algorithms are used by ARITA, discusses briefly the algorithmic complexity of the presented solution and explains how computed layouts could be cached in a database to reduce CPU load and save battery.

- Chapter 5 evaluates the runtime of ARITA with several experiments that stress different layout situations. Moreover it evaluates the layout results with a user experience study, including the results from a questionary that was answered by NetSnips users.
- Chapter 6 summarizes the presented work in this thesis and gives hints to future improvements and extensions of ARITA.
- Appendix A and B show the questionary that was used for the user experience evaluation, including 20 different user generated snippet layouts in portrait screen orientation and their automatically generated landscape mode counterparts.

Chapter 2

Related Work

The work in this thesis is related to different fields, including two-dimensional packing problems (section 2.1), constraint-based layout algorithms (section 2.2) and mobile computing and power saving in general (section 2.3).

2.1 Two-Dimensional Packing Problems

Two-dimensional packing problems are highly relevant for many applications. In many industries a set of rectangular items or goods have to be placed in another bigger rectangular container by minimizing waste or unused space. For instance, in the wood, steel and glass industry exact solutions to those problems are important, when arbitrarily sized customer orders must be cut out from rectangular pallets by minimizing the material waste. In the newspaper business it is sometimes important to maximize the amount of advertisements and minimize the unused space on a page.

Generally speaking, all these problems are variants or special cases of the well-known and extensively studied knapsack problem, which was first described in 1896 by the British mathematician Mathews [2]. It describes the problem of selecting items, each of them with a mass and a value, such that the total value is maximized and the total weight is below or equal to a defined limit. Two common generalizations of the problem are the *bounded* and the *unbounded knapsack problem.* The former problem specifies for every item an upper bound on the number of times an item can be selected. The latter problem defines no constraint on that property. The *unbounded knapsack problem* was proven to be *NP-complete* by Lueker [3] in 1975.

The previously described packing problems in industry can be formalized with the *two-dimensional knapsack packing problem* (2DKP). We assume a set of finite rectangles R = $\{\mathrm{r_1, r_2, \ldots, r_n}\}$, each of them with width $\mathrm{w_i}$, height $\mathrm{h_i}$ and value $\mathrm{v_i}$, which should be placed in a larger container rectangle C with width W and height H such that the total v is maximized and the solution is feasible. A solution for this kind of problem is called feasible if no rectangle overlaps with another and if no rectangle exceeds the bounds of C. Because of the strong need for efficient solutions for these kind of problems, many heuristic approaches and genetic approaches have been presented in the last years. Heuristic approaches that solve the two- and three dimensional knapsack problem efficiently have been studied extensively by Egeblad and Pisinger [4] [5], including variants where items can be rotated. The genetic approach of Bortfeldt and Winter [6] analyzes completely defined packing plans in order to solve unresolved packing problems and adapts its approach by evaluating the results. The authors have shown in experiments with large instances (up to 1000 pieces) that their genetic approach is competitive with the heuristics proposed in recent years.

The mentioned methods are not directly applicable for our problem, since we do not want to minimize the free space or create a layout with the most coherent free space area. Instead, we are searching for a solution that places rectangles equally distributed in the available space of the container. In contrast to 2DKP problems, our problem allows overlapping of rectangles and exceeding the container bounds (if necessary). In other words, in our scenario rectangles can be placed in 2.5 dimensional space, since we are also distinguishing rectangles by their depth level. Most importantly, all the mentioned approaches produce layouts which stand for their own, i.e., they are not related in any way with another rectangle layout. However, as Chapter 3 describes in detail, for our needs the relation between the layout in the portrait and landscape interface orientation is crucial.

In 1983, Chazelle [7] originated an algorithm that implements the rectangle *bottom left placement* heuristic in O(n^2) time. A rectangle layout is said to have the bottom left property if every rectangle is placed as far south and west as possible. The authors state that the algorithm places rectangles in bottom left layouts optimally by identifying in each iteration the most south and left placement point without introducing rectangle overlapping. Healy et al. [8] extended this, by elaborating an efficient algorithm that places rectangles in bottom left manner, even in rectangle layouts that may not respect the *bottom left* property. For the problem presented in this thesis, this requirement is fundamental, since users can arrange their snippets freely with no layout constraints (even with overlapping). Even thought we got some inspiration due to this research work, ARITA is not based on this algorithm for the following reasons. First, the algorithm of Healy et al. [8] is not able to handle rectangle overlapping in the *original* layout. Secondly, we want to exploit free space "gaps" in the layout in order to distribute rectangles more equally, rather than placing rectangles in one specific corner of the container (e.g., *bottom left*). Moreover, ARITA takes also the original position of the rectangles into account and tries to rearrange each rectangle in the nearest and most suitable currently available free space area (refer to Section 4.4 for details).

2.2 Constraint-Based Layout Algorithms

For our needs we do not want to change the user defined rectangle layout completely on an interface rotation by placing rectangles with one of the previously described algorithms space-optimally (without margins). Foremost, the automatically generated landscape snippet layout should be familiar, clear and visually appealing to the user. Moreover, our problem has no clear optimal solution, since it depends strongly on the users preferences. Therefore, we studied various constraint-based layout solutions [9] and machine learning approaches [10], which adapt the user interface based on previous user interaction.

Constraint-based approaches are common in automatic user interface adaption, since they allow great flexibility and real-time repositioning of user interface elements. For instance, Kustanowitz and Shneiderman [11] elaborated a real-time algorithm for displaying and auto-aligning photos for events, consisting of a primary region with the main event photo and surrounding secondary regions for other related photos. They introduce a number of constraints that define layout guidelines for photo libraries, which must be respected by the auto-layout algorithm. Even though relaxing some of those constraints could lead to more photos in the layout, the authors state that this would lead also to an unbalanced view and lower visual clarity.

Moghaddam et al. [12] propose a system that, based on an initial user defined photo layout, generates a new layout for related photos. In situations with many retrieved images, this may lead to undesired image overlapping. Therefore the same authors elaborated an optimization process [13] that minimizes overlapping by adjusting sizes and positions and at the same time tries to preserve the impression that related images are close to each other.

Our presented solution does not fall in the strict definition of a constraint-base layout algorithm, but we use constraints implicitly, for instance the decision for fixing rectangles that stay in screen bounds or the intention to distributed repositioned rectangles equally in the available free space (refer to Chapter 3 for details).

2.3 Mobile Computing and Power Saving

Since the problem presented in this dissertation targets an implementation for mobile devices, we are not only interested in highly efficient algorithms for user interface adaption, but also for saving battery power. Longer running algorithms prevent the CPU from going in power saving idle states, thus reducing the battery life of the device. Carroll and Heiser [14] showed that in smartphones the CPU is one of the components that consumes most energy (1st place in *high load* scenarios, 2nd place in *suspended state* and 4th place in *idle case*). Burr et al. [15] state that the potential for power savings in software is greater than the potential for savings in hardware, althought the software power savings are more difficult to achieve.

Already in 1996 Udani and Smith [16] claimed that, in contrast to memory size and CPU power, the improvements in battery capacity will not be significant in the next years. Comparing the battery capacities for mobile devices from the nineties with actual ones, proves that these predictions were accurate and still valid. For this reasons energy efficient algorithms must be used in our implementation for carefully utilizing the scarce battery capacity of mobile devices.

Even though the processing speed of mobile devices, like smartphones and tablets, improved significantly in the last years, the CPU power of these devices is not yet at the level of a typical desktop computer. Roberts-Hoffman and Hegde [17] showed in 2009 that even low-end desktop CPUs (Intel Atom 330) are approximately 3 times faster than high-end smartphone processors (ARM Cortex-A8), although the mobile processors showed significantly lower power consumption. This fact was also taken into consideration when choosing the techniques for ARITA.

Chapter 3

Problem Description

In this chapter, we detail the problem that we address in this thesis. Section 3.1 analyzes the snippet placing behavior in the current version of NetSnips. Section 3.2 states the requirements that a repositioning algorithm shall respect in order to fulfill our needs.

3.1 Positioning Snippets in NetSnips

In NetSnips users can define web snippets of almost any size. Starting from very small to very big web snippets that fill out the complete display. In contrast to many other applications that give users the possibility to define arbitrarily sized content, in NetSnips users can position these snippets completely freely and pixel accurately on the user interface on one of the available pages. Snippets are not forced to "dock" on a specific raster that partitions the user interface. Snippets are also allowed to (partially) overlap each other. User may do so, because they find the hidden snippet content less important or visually more appealing to have specific overlapping.

User reviews show that they really appreciate the high level of customization and the high flexibility of creating and arranging web snippets. This is confirmed by anonymized usage statistics, that highlight the fact that users spend a considerable amount of time with personalizing and positioning their web snippets.

The current version (2.1.1) of NetSnips does only support portrait screen orientation on all devices. However one of the most popular feature requests is the support for landscape screen orientation. Especially on tablets, like the iPad, supporting all interface orientation is extremely important. Users expect that applications adapt to their current context of use, which may be highly varying on mobile devices. Mobile devices may be used in portrait mode while reading longer texts and whenever one-hand use is preferred (for instance while walking). Landscape mode is generally favorited for reading in more relaxed situations, for instance while laying on the bed, examining wide tables or charts and whenever both hands can be used to interact with the touch screen device. Moreover due to the fact that mobile devices are generally equipped with smaller displays, depending on the content, users could find it more comfortable to rotate the device in their preferred screen orientation.

The high flexibility for creating and positioning web snippets in NetSnips, makes it difficult to support both screen orientations. Depending on the position and size of the web snippets, some of them may be (partially) out of screen bounds in one of the two interface idioms. The simplest solution to this problem would be to introduce a scrolling view, in order that users can scroll to the snippets that are currently (partially) hidden. However, we believe that this would lead to a very bad user experience, since it is no longer possible to see the full content at a glance. This claim is supported by the study of Sanchez and Branaghan [18], where user experiments showed with a strong evidence that the need to scroll content on mobile devices influences reasoning performance and task completion times negatively. Allowing user to rotate the device in landscape screen orientation in order to minimize or prevent scrolling completely, led to faster information reception and lower task completion times. Therefore we decided against the trivial scrolling solution and defined our requirements on a snippet repositioning system.

3.2 Requirements for the Repositioning of Snippets

We define the following *non-functional requirements* for the repositioning algorithm:

1. **Fast adaption to the current interface idiom**: Rotating a mobile device is a very common action. Therefore, users expect that the user interface adapts almost immediately to the current interface idiom (portrait or landscape). The repositioning system shall find a solution that adapts the position of all snippets for the current interface idiom in a reasonable amount of time, typically not more than 1000ms.

2. **Respect for the originally user defined layout**: As stated previously, users position snippets in a very specific and individual way in one interface mode. Therefore the system shall respect the user defined snippet layout when generating a similar layout for the other interface idiom, rather than repositioning all snippets. More specifically the system shall not unnecessarily move snippets to another position if they are not partially or completely hidden in the current interface idiom. The motivation behind this requirement is that users might have had good reasons when positioning two snippets next to each other. Snippets may be related in some way, for instance, users might like to compare multiple weather snippets from different locations. It is assumed that the system is not able to identify and validate the importance of the content of a particular snippet and the reasoning behind the user defined layout. Therefore, the change to the layout shall stick to the original layout as much as possible.

3. **Move snippets to the nearest free space**: If the interface orientation changes, the system shall reposition snippets in the closest free space relative to the original snippet position. Depending on the original layout, this could enhance the visual

impression that the new layout did not change fundamentally. This requirement assumes that the layout offers multiple candidate positions for the current snippet, which lead to no overlapping with other snippets and/or overflowing the screen bounds.

4. **Minimize overlapping and overflow**: This requirement describes the opposite situation in comparison to the above stated requirement. The system shall set snippets, which cannot be placed without overlapping or overflowing, to a position that *minimizes* overflowing and/or overlapping.

5. **Distribute snippets equally**: If snippets must be moved to a new position, the system shall place them in a way that the vertical and horizontal margins to the screen bounds and to other snippets are equally distributed.

Our requirements for the repositioning system respect the interface design guidelines for mobile devices of Gong and Tarasewich [19]. More specifically, we focus especially on *"Support Internal Locus of Control"* by minimizing the automatically layout changes on the user defined layout. Our decisions are also influenced by guideline *"Design for speed and recovery"* and *"Allow for personalization"*.

According to our knowledge a positioning system for rectangles that fulfills the above stated requirements has not yet been addressed in literature.

Chapter 4

Adjusting Rectangles on Interface Turn Algorithm

In this chapter we describe the **A**djusting **R**ectangles on **I**nterface **T**urn **A**lgorithm (ARITA). Section 4.1 lists all symbols and abbreviations used in this chapter. Section 4.2 describes the criteria for identifying rectangles that should be repositioned. Section 4.3 and section 4.4 focus on discovering the free space in the rotated interface and finding a new rough position for rectangles that should be rearranged, respectively. Section 4.5 and section 4.6 discuss how these rough rectangle positions could be fine-tuned, in order to archive a more harmonious visual result. Section 4.7 describes the complete ARITA, which performs the described steps in this chapter in the right order. Finally section 4.8 discusses the algorithmic complexity of ARITA and section 4.9 explains how computed layouts could be cached in a database to reduce CPU load and save battery.

4.1 Symbols and Abbreviations

We describe a rectangle in the coordinate system of the container rectangle by specifying its *frame*, which is composed of:

- the left-upper based *origin*, specified in absolute x/y coordinates and abbreviated with *origin (x/y)*
- the *size*, specified in absolute width/height values and abbreviated with *size (width/ height)*

Sometimes, especially in pseudo code samples, we are also abbreviating the specification of the frame by using the notation *frame (x, y, width, height)*. We refer to specific parts of the frame with the *dot notation*. For instance, we use *frame.width* for referring to the width value or *frame.center* for the center point of the frame. Moreover we use the functions *minX(frame), maxX(frame), minY(frame)* and *maxY(frame)* for referring to the minimal/maximal x and y coordinates of a frame. For instance, *minX(frame)* = *frame.x* and *maxX(frame)* = *frame.x* + *frame.width.*

Table 4.1 shows all symbols used in this chapter. We will refer to them in the following sections.

Symbol	Description
C	Container rectangle
$\mathrm{R_{all}}$	Set of all rectangles positioned in C
$\mathrm{R_{adj}}$	Set of all adjustable rectangles $\subset \mathrm{R_{all}}$
$\mathrm{R_{fix}}$	Set of all fixed rectangles, i.e. $\mathrm{R_{all} \setminus R_{adj}}$
$\mathrm{FS_{fix}}$	Set of all free spaces constraint to the rectangles in $\mathrm{R_{fix}}$

Table 4.1: Symbols used in this chapter.

4.2 Identification of Adjustable Rectangles

In the current and following sections, we will perform ARITA step-by-step on the example layout illustrated in Figure 4.1. The initial layout in portrait mode (outlined with a green border), is specified by the user and composed of 6 rectangles of different sizes and proportions. We assume that the device will be rotated to landscape screen orientation (outlined with a blue border). After the interface rotation some rectangles are positioned completely out of screen bounds (rectangles 5, 6) or partially (rectangles 3, 4). The hidden snippet areas in landscape are highlighted in translucent red.

We describe the available area for placing all rectangles (i.e., dimension of the user interface) with a container rectangle C. Moreover each web snippet is represented by a rectangle. On device rotation, the orientation of the user interface and therefore also of C changes, i.e. the previous container width becomes the height and vice versa.

After device rotation, ARITA identifies all rectangles that are positioned partially or completely out of the bounds of C. The algorithm aggregates all rectangles in $\mathrm{R_{all}}$ that are *adjustable.*

Definition 4.1 (**No Overflow Criteria**). A rectangle r satisfies the *no overflow criteria* iff *r.width* $\leq$ *C.width* and *r.height* $\leq$ *C.height*.

Definition 4.2 (**Adjustable Rectangle**). Let p be the *bottom right corner* point of rectangle r. We call r *adjustable* iff it fulfills the no overflow criteria and either or both of the following conditions are true: (i) p.x $>$ C.width (ii) p.y $>$ C.height.

This first step of ARITA results in a set of *adjustable rectangles* $\mathrm{R_{adj}}$, i.e., rectangles that are positioned (partially) out of the *container rectangle* and fulfill the *no-overflow criteria.* We consider these rectangles for repositioning. If this set is empty, ARITA does not perform any changes and terminates immediately. In our example illustrated in Figure 4.1 we have $\mathrm{R_{adj}} = \{\mathrm{r_3, r_4, r_5, r_6}\}$.

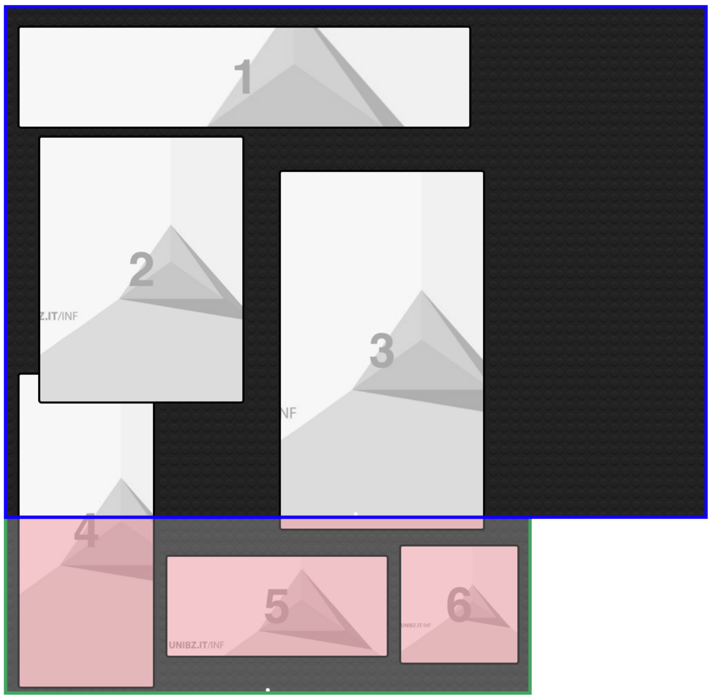


FIGURE 4.1: The user defined layout in portrait mode and in landscape mode. In landscape some snippets are completely or partially hidden (highlighted in red).

The reason for excluding these very wide and/or high rectangles, which do not fulfill the *no-overflow criteria*, is founded in the general idea of the algorithm. As described in chapter 3, ARITA does only intervene in the user specified rectangle layout if it is absolutely needed and content is hidden. We do not perform any actions on rectangles that overflow screen bounds *on every position* in C. This is true for rectangles that do not fulfill the *no overflow criteria* and therefore no processing time should be wasted for finding another (possible inferior) position for these rectangles. It is assumed that only the user can valuate the different importance of the visualized content and decide *which part* of the snippet is less critical. All rectangles that are *not adjustable* are part of $\mathrm{R}_{\mathrm{fix}}$ and considered as *fixed*, i.e., their position should not be changed.

4.3 Determining Free Rectangular Spaces

In order to find a suitable position for every rectangle in $\mathrm{R}_{\mathrm{adj}}$, we identify all *free spaces* that exist in C constraint to the rectangles in $\mathrm{R}_{\mathrm{fix}}$, that is all rectangles in $\mathrm{R}_{\mathrm{all}} \setminus \mathrm{R}_{\mathrm{adj}}$.

A *free space* is a rectangular area in the coordinate system of C, which is not intersected by any rectangle $\in R$.

Definition 4.3 (Free Space). Let $R = \{r_1, r_2, \ldots, r_n\}$ be a set of rectangles. A *maximal free space* f is a rectangle, which does not intersect with any r $\in$ R, i.e., $\forall$ $\mathrm{r}_{\mathrm{i}} \in$ R (f $\cap$ r_{i} = $\emptyset$) and $\nexists$ f’: f’ $\geq$ f.

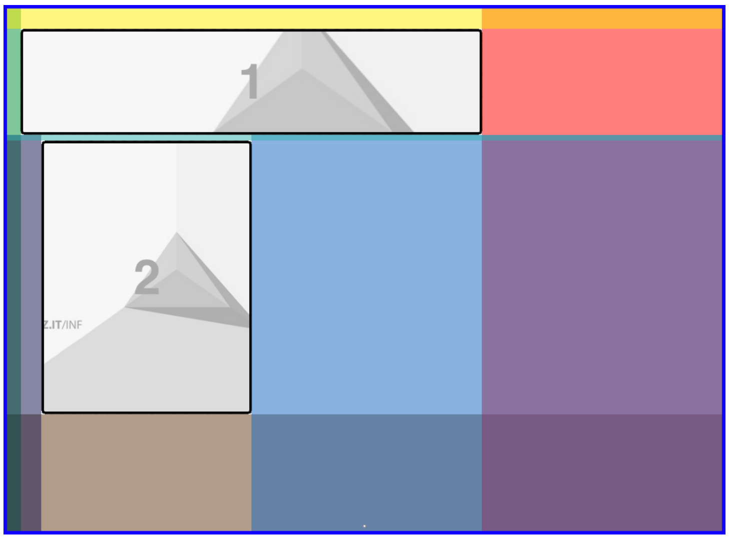


FIGURE 4.2: All 7 partially overlapping free spaces (translucent colors) in respect to the fixed rectangles 1 and 2 in landscape interface orientation.

We use the sweep line technique of Shamos and Hoey [20] in order to compute the set of free spaces FS_{fix} in C. The technique uses an imagined vertical line with height equal to C.height and sweeps from left to right over the container rectangle. Moreover we maintain two sets of free spaces. Free spaces that are no longer processed by the sweep line are closed and considered as *dead*. Free spaces that did not yet reach their maximum extend are considered as *active*. The sweep line stops at x values where special events occur. For our purposes this are all x values where a fixed rectangle is horizontally starting or ending. If such an event is triggered, we create new free spaces or split and close active ones accordingly.

Figure 4.2 shows all *free spaces* (translucent, colored areas) in respect to the fixed rectangles R_{fix} (rectangles 1 and 2). At this time, all other rectangles (3-6) are not yet present in the landscape rectangle layout, since their new position will be defined by the algorithms presented in the following sections. The free spaces in this example are *partially overlapping*, which is emphasized by depicting them with translucent colors.

The following two sub sections present the theory behind the two main steps of our tecnique: the *processing, closing and splitting* of active spaces and the *instantiation* of new active spaces.

4.3.1 Processing of Active Spaces

If the sweep line collides with a *horizontal start of a rectangle*, currently existing free spaces must be closed and split into sub spaces. Figure 4.3 illustrates the sweep line at x = 20 on the collision with rectangle 1. The initial free space (green) is closed and results in two new free spaces (yellow and blue) that are partially overlapping the initial free space. At this time, the new split spaces have width equal to 20, but for a better general view Figure 4.3 illustrates the final space sizes in dashed-yellow and dashed-blue, respectively.

In order define the size and position of resulting split spaces we keep a list of vertical lines that represent the vertical dimensions of rectangles at a given x value. In other words, if a rectangle r is colliding with the sweep line at x, then we create a new vertical line l that describes the y-position and height of that rectangle in C. We set $l.start = r.y$ and $l.length = r.height$. If rectangles are overlapping each other at a given x value, then the vertical lines are intersecting as well. Intersecting lines are combined into a simple line.

The vertical line list describes where rectangles are present. However, for splitting an active space into sub spaces, we need the information where *no rectangles* are present. Given the line list we can compute the opposite list, called *negative line list*. Given a x coordinate in C, the negative line list specifies at which vertical intervals *no rectangles* are present. Ultimately, we specify the frames of the new split spaces with the information contained in the negative line list. Figure 4.3 shows the negative line list (in orange; containing nl_1 and nl_2) at $\mathrm{x} = 20$.

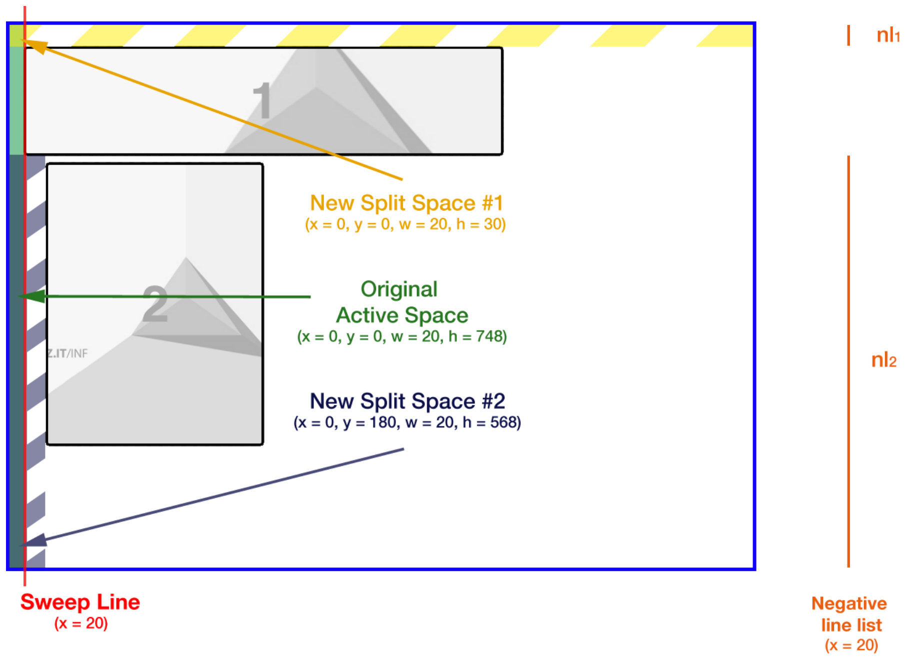


FIGURE 4.3: The *sweep line* and *negative line list* at $\mathrm{x} = 20$.

4.3.2 Instantiation of New Active Spaces

If the sweep line collides with a *horizontal end of a rectangle*, new free spaces must created. Figure 4.4 illustrates how a new active space is defined as soon as the sweep line reaches the end of rectangle 2 at $\mathrm{x} = 350$. In order to define the frame of the new active space, we are again calculating the *negative line list*. However, this time not at the current sweep line position, but one step ahead ($\mathrm{x} = 351$). Moreover we are restricting the calculation of the negative line list to the scope of the currently ending rectangle(s). Using the lines in the negative line list, we gather the information for creating the frames for the new free spaces.

Figure 4.4 depicts the negative line list (in orange; containing only nl_1), which was used to create the new active space (in blue). At this time the new active space has width equal to zero, but for a better general view the final space size is highlighted in dashed-blue.

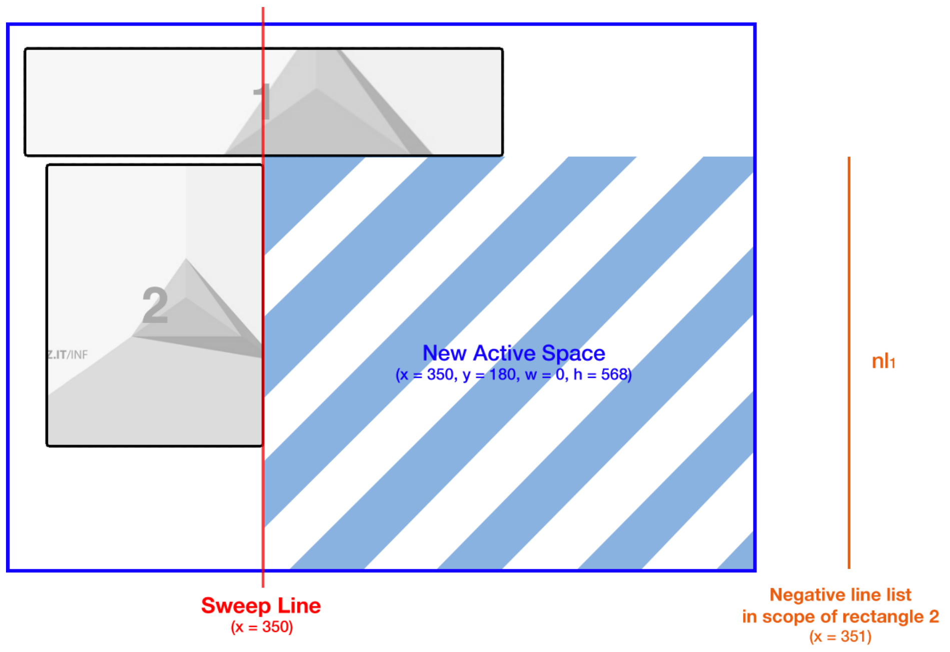


FIGURE 4.4: The *sweep line* at x = 350 and *negative line list* for x = 351 in scope of rectangle 2.

4.3.3 Algorithm for Computing Free Spaces

Algorithm 1 shows the pseudo-code for the COMPUTEFREESPACES algorithm, which implements the free space splitting and creating behavior described previously.

First, COMPUTEFREESPACES initializes two empty sets called *DeadSpaces* and *ActiveSpaces* (line 1). A free space is considered as *dead* as soon it is closed, i.e., the sweep line passed it and will not process it furthermore. A free space is called *active* if the sweep line will process it further on, when new rectangle starting- or ending events are arising. An initial free space is created with origin (0/0) and size (0/C.height) and added to the *activeSpaces* set (lines 2-3). All rectangle starting- and ending event points, as well as the container end point, are collected in a set (lines 4-5). The sweep line is implemented with a loop, that iterates through all event points in ascending sorted order. At every iteration all spaces that are currently in *ActiveSpaces* set are marked as unprocessed (line 7).

Every iteration passes through two main parts that were discussed previously. The *instantiation of new active spaces* (subsection 4.3.2) is implemented in lines 7-16 and the *processing, closing and splitting of active spaces* (subsection 4.3.1) is realized in lines 17-28. After processing all event points, all remaining active spaces are added to the *DeadSpaces* set which is returned as final result (lines 29-30).

```
Input: A container rectangle $C$; a set of rectangles $R$ that are positioned in $C$
Output: The set of free spaces in $C$ in respect to the rectangles in $R$
1  Initialize empty sets DeadSpaces, ActiveSpaces
2  Let $f$ be the initial free space with frame (0, 0, 0, $Y_{max}$)
3  Add $f$ to ActiveSpaces
4  Let $X = \{x_1, x_2, ..., x_k\}$ be the set of x-values of rectangle starting/ending points in $R$
5  Add C.width to $X$
6  foreach $x \in X$ in ascending sorted order do
7      Set all spaces in ActiveSpaces unprocessed
8      if at least one rectangle $\in R$ is horizontally ending at $x$ then
9          Let LS be the vertical line list returned by LINES(x+1, R)
10         Let NLS be the negative vertical lines list returned by NEGATIVELINES(LS, C)
11         Sort NLS by y-start coordinate
12         foreach $nl \in$ NLS do
13             Let $a$ be a new active space with frame (x, nl.start, 0, nl.length)
14             Flag $a$ as processed and add it to ActiveSpaces
15         end
16     end
17     repeat
18         Extract unprocessed space $s$ in ActiveSpaces
19         if $s$ found then
20             Set width of $s$ until $x$
21             if at least one rectangle $\in R$ is horizontally starting at $x$ then
22                 Let LS be the vertical line list returned by LINES(x, R)
23                 Compute SplitSpaces set for $s$ with SPLITSPACES(s, LS)
24                 Set spaces in SplitSpaces processed and add them to ActiveSpaces
25                 Remove $s$ from ActiveSpaces and add it to DeadSpaces
26             end
27         end
28     until all spaces in ActiveSpaces are processed
29     Add spaces in ActiveSpaces to DeadSpaces
30     return DeadSpaces
31 end
```

Algorithm 1: COMPUTEFREESPACES

The previously discussed concepts of vertical *lines* and *negative lines* for detecting the vertical covering situation of rectangles at x are implemented with the helper algorithms LINES and NEGATIVELINES, respectively. The algorithm SPLITSPACES uses the line list for splitting a given free space into sub spaces.

4.4 Placing Rectangles in Free Spaces

After identifying all free spaces, we need to select a free space container for all rectangles in R_{adj}. Every adjustable rectangle should be placed in a free space that fits the dimensions of the actual rectangle. If this is not the case (i.e., the rectangle is higher and/or broader as any available free space), we select the free space container that minimizes the overflow area. However often the opposite is the case and a rectangle could be placed in more than one free space container. For this reason we compute the set of free space candidates that can completely fit the actual rectangle. In order to decide which of the container candidates fits the actual rectangle best, we introduce the *container-rectangle difference*.

Definition 4.4 (Container-Rectangle Difference). The difference between free space container c and rectangle r is defined as:

$$\mathit{difference}(c, r) = sizeWeight * \mathit{sizeDiff}(c, r) + posWeight * \mathit{posDiff}(c, r)$$

where $sizeWeight$ and $posWeight$ may be any positive decimal and

$$\mathit{sizeDiff}(c, r) = |c.width - r.width| + |c.height - r.height|$$

$$\mathit{posDiff}(c, r) = |c.center - r.center|$$

The *container-rectangle difference* function takes into account the absolute size and position differences between a container candidate and a rectangle, in both cases with a weight parameter. For our experiments we have used *sizeWeight = 1* and *posWeight = 2* in order to place adjustable rectangles closer to their original position. In some rectangle layouts this emphasizes the visual impression that the new rectangle layout changed insignificantly compared to the original one.

The placement order of rectangles and the selection of the free space container definitely affects the final layout. As described in the following, ARITA uses a *greedy approach* for finding a new position for all rectangle in $\mathrm{R}_{\mathrm{adj}}$.

Algorithm 2 shows the pseudo-code for the PLACERECTANGLES algorithm. First, it sorts all rectangles in $\mathrm{R}_{\mathrm{adj}}$ in descending order by their surface area (line 1). This assumes that rectangles that with larger surface are more difficult to place, i.e., there exist fewer free space candidates that match these rectangles. The algorithm loops through all rectangles in the passed rectangle set and performs the following steps. First it identifies all free space candidates that can completely fit the passed rectangle (line 3). If this set is not empty, it returns the container c with the minimal *container-rectangle difference* in respect to r (line 5). If no container candidates could be found, which means that the rectangle does not fit in any free space, the algorithm returns the free space container that leads to minimal overflow (line 7). The current rectangle is placed at origin (0/0) in the selected container (line 9). Since usually a rectangle does not fill out the container completely, the resulting sub container spaces on each side (top, bottom, left, right) are calculated and added to the current free spaces set (lines 10-11). The selected container space is removed from the free spaces set (line 12).

The result of PLACERECTANGLES is an initial, temporary layout that contains the newly positioned rectangles in $\mathrm{R}_{\mathrm{adj}}$ and the unaltered rectangles in $\mathrm{R}_{\mathrm{fix}}$. Figure 4.5 shows the temporary new rectangle layout and highlights all *free spaces* after placing all adjustable rectangles.

```
Input: A set of free space $FS$; a set of rectangles $R$ that should be positioned in $C$
1  Sort rectangles in R descending by area
2  foreach rectangle $r \in R$ do
3      Identify candidate containers $CC$ for $r$ in $FS$
4      if $CC$ is not empty then
5          Let $c$ be the selected container returned by MinimalDifference(CC, r)
6      else
7          Let $c$ be the selected container returned by MinimalOverflow(FS, r)
8      end
9      Place $r$ in $c$ at origin (0/0)
10     Compute $SubSpaces$ set for $c$ after placing $r$
11     Add $SubSpaces$ to $FS$
12     Remove $c$ from $FS$
13 end
```

Algorithm 2: PlaceRectangles

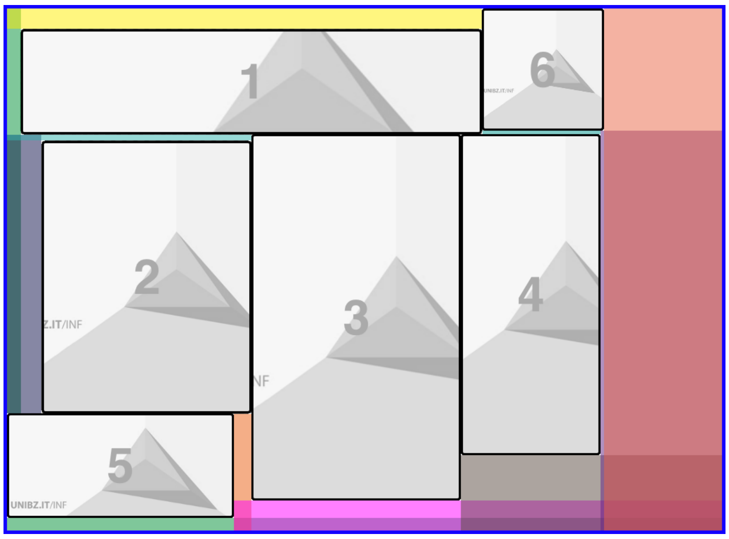


Figure 4.5: The temporary landscape rectangle layout. Remaining *free spaces* are highlighted with translucent colors.

4.5 The Layout Graph

The PlaceRectangles algorithm places all adjustable rectangles without any margins in a selected free space container. This behavior is desired during rectangle placement, since it guarantees that the free space in C is not unnecessarily wasted or "blocked" until all adjustable rectangles are placed.

Once all rectangles are placed, we perform layout adjustments. This way we achieve a more balanced rectangle layout, i.e., equally distributed margins between rectangles and screen bounds. In order to understand how much we can move each rectangle in horizontal and vertical direction without introducing additional rectangle overlapping or screen overflow, we use a graph data structure to represent a rectangle layout. We call this graph *layout graph*.

In other words, the results of PlaceRectangles defines the "raw" rectangle layout and the *layout graph* is used for balancing the layout, i.e., introducing margins between adjustable rectangles.

4.5.1 Definition

The layout graph is an *undirected graph* with nodes and edges, whereas *each node represents a rectangle* in the layout. In addition to the nodes for all rectangles, the layout graph contains 4 *special nodes* that represent the screen bounds in all 4 cardinal directions. A node exposes 4 anchors, that are named like cardinal directions, namely west anchor, east anchor, south anchor and north anchor. Each node anchor bundles all edges for one of the 4 possible directions. An edge describes a *connection between a node and a neighbor node.* Two nodes are connected in one of the four directions, if the underlying rectangles are directly adjacent to each other in a particular direction, i.e., no other rectangle is in between.

Definition 4.5 (Neighbor Rectangle). 2 not overlapping rectangles r and s are called horizontal (vertical) *neighbor rectangles* iff they can be connected with a perfectly horizontal (vertical) line without crossing any other rectangle in the layout.

Definition 4.6 (Layout Graph). The *layout graph* is a 3-tuple $G = \{N, E, R\}$ s.t.

- N ... finite set of nodes
- E ... finite set of edges
- R ... finite set of rectangles
- each n ∈ N corresponds to one r ∈ R
- 2 nodes are connected if the corresponding rectangles are neighbors

Nodes in the layout graph can be either *fixed* or *movable.* Edges between nodes can be either *fixed* or *adjustable.*

Definition 4.7 (Fixed and Movable Node). We say a node is *fixed* if its underlying rectangle r is fixed, i.e., $r \in R_{\text{fix}}$. We say a node is movable if its underlying rectangle r is adjustable, i.e., $r \in R_{\text{adj}}$.

Definition 4.8 (Fixed and Variable Edge). We say an edge is *fixed* if it connects 2 *fixed nodes.* We say an edge is *variable* if it connects 2 nodes, whereas *at least one* of them is *movable.*

Figure 4.6 shows the layout graph for the rectangle layout illustrated in Figure 4.5. Since nodes and rectangles are related to each other, all nodes are depicted at their approximate rectangle center point. Fixed edges and nodes are shown in light gray, whereas movable nodes are shown in bold. Horizontal variable edges are shown in orange, whereas vertical ones are illustrated in red. Moreover all variable edges are labeled with their edge distance.

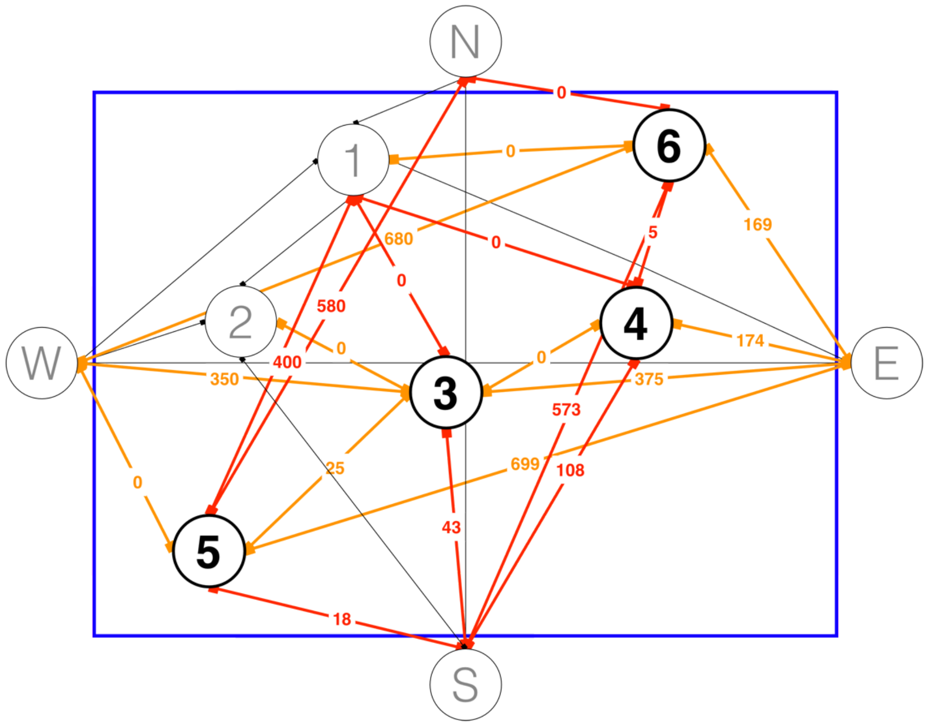


Figure 4.6: The layout graph with all nodes and edges for the rectangle layout in Figure 4.5.

4.5.2 Constructing the Layout Graph

In the following we will describe how we construct the layout graph. The layout graph is build in 3 main steps:

1. Node instantiation
2. Identification of *west* and *east* neighbors for all nodes
3. Identification of *north* and *south* neighbors for all nodes

The first step initializes the layout graph with the 4 special nodes that represent the screen bounds in a particular cardinal direction. Since these nodes are not included in the temporary rectangle layout, we have to link them to virtual rectangles. For instance, *WEST* node is defined with a rectangle with frame (-∞, 0, ∞, C.height). Next, we create nodes for every rectangle. Figure 4.7 shows the layout graph after initializing all nodes for the rectangle layout in Figure 4.5. All movable nodes are shown in bold.

The second step for building the layout graph is performed by SetWestAndEast-Neighbors (algorithm 3), an algorithm that concatenates each node with its *west and east neighbors.* The algorithm loops through the set of nodes (line 1) and identifies for each of them the west neighbor nodes by using the helper algorithm WestNeighbors (algorithm 4). After identifying all west neighbors, it concatenates all of them with the current node (lines 5-6).

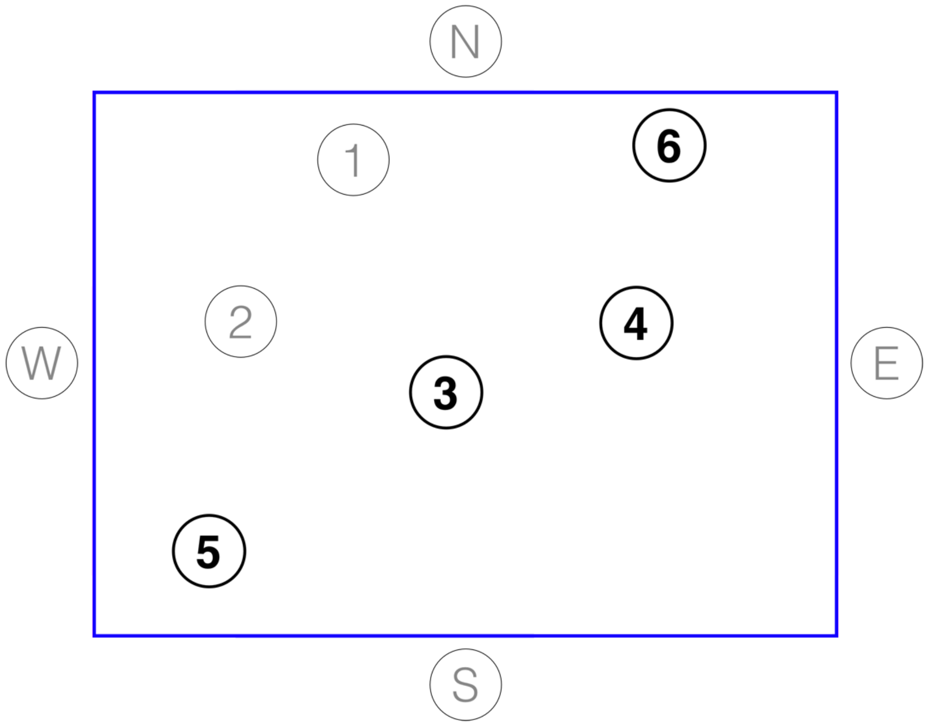


Figure 4.7: The layout graph after initializing all nodes; movable nodes are bold.

```
Input: A layout graph $lg$
1 Let $N$ be the set of nodes in $lg$
2 foreach node $n$ in $N$ do
3     Let OtherNodes be the set $N \setminus n$
4     Compute west neighbors WN returned by WestNeighbors(n, OtherNodes)
5     foreach west neighbor $wn$ in WN do
6         Add edge from west anchor of $n$ to east anchor of $wn$
7     end
8 end
```

Algorithm 3: SetWestAndEastNeighbors

The helper algorithm WestNeighbors (algorithm 4) uses a *backward* sweep line technique to identify all west neighbors for a given target node n. First, it initializes an empty set for storing all identified west neighbor nodes (line 1) and another one for keeping track of the currently allowed *neighbor ranges* (line 2). A *range* is vertical line with a start point and length. An allowed *neighbor range* describes a vertical section that is currently monitored for collisions with rectangles when swiping in west direction. The initially allowed west *neighbor range* is defined with start point equal to the target node's rectangle y-coordinate and length equal to the rectangles height (line 4). This initial range is added to the set of all currently allowed *neighbor ranges* (line 5). All rectangle ending event points that are positioned more western as the rectangle of the target node are collected in a set (lines 7).

After initialization, WestNeighbors sweeps backward starting from *most eastern* rectangle ending event point. At every stop, the algorithm checks if the *neighbor ranges* set is not empty (line 9). If so, it identifies all neighbor candidate nodes that are associated with a *horizontally ending* rectangle at the current event point (line 10). It loops through all identified nodes and checks if the vertical *range* of the node is in scope of the currently allowed *neighbor ranges* (line 11-12). If this is the case, it adds the current neighbor candidate node to the set of identified west neighbors (line 13). It also deletes the vertical range of the added node from the currently allowed neighbor ranges (line 14).

Figure 4.8 shows the layout graph after identifying *west and east neighbors* for all nodes, highlighting all variable edges in orange. Fixed edges and nodes are shown in light gray, whereas variable edges are shown in orange with their actual edge distance.

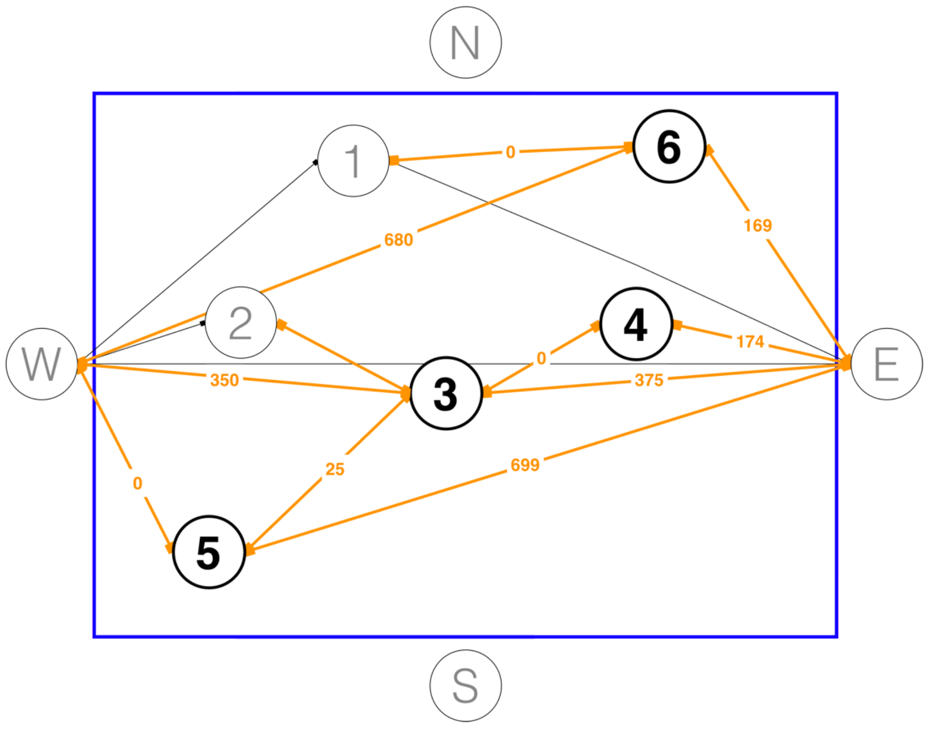


FIGURE 4.8: The layout graph after setting all west and east edges to all nodes. Variable edges are labeled and highlighted in orange.

The third and last step for building the layout graph is performed by SETNORTHAND-SOUTHNEIGHBORS, an algorithm that works analogously to SETWESTANDEASTNEIGHBORS. The difference is founded in the fact, that all subroutines target the y rather than the x axis and use rectangle widths rather than the rectangle heights for identifying neighbors. After concatenating all nodes with their vertical neighbors, the *final layout graph* corresponds to the layout graph illustrated previously in Figure 4.6.

4.6 Balancing the Layout Graph

For balancing, we have to *extract the variable paths* in the layout graph (subsection 4.6.1) and use them for deciding how the *edge distances* between nodes must be adjusted (subsection 4.6.2).

```
Input: A target node n; A set of nodes N
Output: The set of west neighbors for target node n
1  Initialize empty set WestNeigbors
2  Initialize empty neighbor ranges set NeighborRanges
3  Let r be the underlying rectangle of n
4  Create initial range ir with start point = r.y and length = r.height
5  Add ir to NeighborRanges
6  Let NC = {nc_1, nc_2, ..., nc_k} be the set of neighbor candidates with underlying
   rectangle s maxX(s_k) < minX(r)
7  Let X = {x_1, x_2, ..., x_k} be the set of x-values of rectangle ending points for all nodes
   in NC
8  foreach x ∈ X in descending sorted order do
9      if NeighborRanges not empty then
10         Identify nodes NN = {nn_1, nn_2, ..., nn_l} ending at x
11         foreach nn in NN do
12             if range of nn in NeighborRanges then
13                 Add nn to WestNeigbors
14                 Delete range of nn from NeighborRanges
15             end
16         end
17     end
18 end
19 return WestNeigbors
```

Algorithm 4: WestNeighbors

4.6.1 Extracting Variable Paths

We are using the layout graph for adjusting the temporary position for rectangles in R_{adj}, i.e., we are balancing or equating the margins between these rectangles and screen bounds. We call this activity *layout graph balancing*, and perform it in 2 *independent* steps: First, we balance the *layout graph* horizontally, from *WEST* node to *EAST* node. Secondly, we are balancing it vertically, from *NORTH* node to *SOUTH* node. In the following we will describe the horizontal layout graph balancing process in detail. The vertical process is analogous and differs only in the vertical graph processing direction.

For balancing we exploit the *variable paths* in the layout graph, either in horizontal or in vertical direction. Movable nodes that are part of a variable path are considered for repositioning.

Definition 4.9 (Variable Path). A *variable path* is a sequence of *variable edges* starting from a *fixed node* to another *fixed node*.

The previous Figure 4.8 highlights all horizontal *variable edges* in orange for the example layout graph. In order to identify the *variable paths* in the layout graph, we use the algorithm HorizontalVariablePaths (algorithm 5). Figure 4.9 lists all horizontal *variable paths* in the example layout graph with labeled edges *before* initiating the layout balancing process.

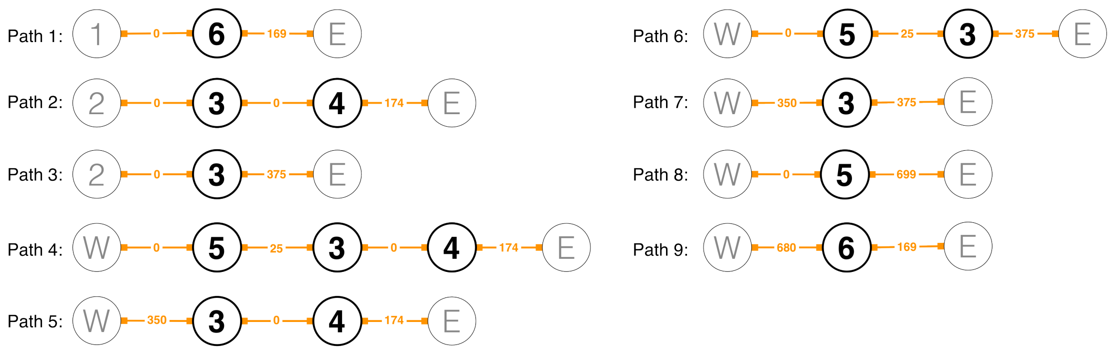


FIGURE 4.9: The list of all horizontal *variable paths* in the layout graph with labeled edges *before* layout balancing.

The algorithm HORIZONTALVARIABLEPATHS first initializes an empty set for storing all identified horizontal variable paths in the layout graph (line 1). The algorithm repeats the following steps as long as a start node with *unprocessed* edges can be found, starting at *WEST* node and heading against *EAST* node. It identifies the current start node with NODEWITHUNPROCESSEDEGDE, a depth-first search algorithm [21] that also starts at *WEST* node and progresses horizontally. It returns the first found node with *at least one* unprocessed edge (line 4). If a start node could be found, the algorithm uses the sub routine PATHS for identifying all *variable paths* from the current start node to the next *fixed node* (line 6). PATHS flags internally all visited edges as processed, in order that HORIZONTALVARIABLEPATHS terminates as soon as no further node with unprocessed edges could be found, i.e., the algorithm NODEWITHUNPROCESSEDEASTEDGE returns no further start node.

Input: A layout graph *lg*
Output: A set of horizontal variable paths in *lg*

```
1  Initialize empty variable paths set $VP$
2  Initialize current start node $sn$ as unspecified
3  repeat
4      Find $sn$ with NodeWithUnprocessedEastEdge(lg)
5      if $sn$ found then
6          Compute current variable paths $CVP$ starting from $sn$ with Paths(sn)
7          Add paths in $CVP$ to $VP$
8      end
9  until $sn$ not found
10 return $VP$
```

Algorithm 5: HORIZONTALVARIABLEPATHS

4.6.2 Balancing Variable Paths

The result of the HORIZONTALVARIABLEPATHS algorithm is a unordered set of horizontal variable paths. This set can contain variable paths that share the same nodes completely or partially. All variable paths that include at least one common *variable node* are called *conflicting variable paths*, since it it is not obviously clear, which of these variable paths should be used for the balancing process.

Definition 4.10 (**Conflicting Variable Path**)**.** Let P and Q be 2 horizontal (vertical) *variable paths* and N = $\{n_1, n_2, ..., n_k\}$ and M = $\{n_1, n_2, ..., n_l\}$ their traversed *variable node sets*, respectively. We call P and Q horizontally (vertically) *conflicting* iff $N \cap M \neq \emptyset$.

We have to decide how to balance *conflicting paths*, i.e., which of the *conflicting paths* should be used for balancing the contained nodes. Choosing the "wrong" path for balancing could lead to undesired overlapping of *adjustable rectangles*, therefore it is important to resolve these conflicts. However it is also common that there exist various possibilities that lead to no additional rectangle overlapping but result, depending on the chosen *variable path*, in different rectangle layouts.

In order to resolve the problem of conflicting paths, we elaborated a strategy that introduces no *additional* overlapping of *adjustable rectangles*, i.e., it does not add further overlapping in comparison to the temporary rectangle layout that resulted after performing the PLACERECTANGLES algorithm. Our current solution does not weight which of the non-overlapping solutions is *best* against *any criteria.*

In order to balance all *variable paths* efficiently based on the previously stated requirements, we have elaborated a solution that, instead of identifying and resolving all *variable path conflicts*, it sorts and processes variable paths in such a way that conflicts can be avoided. More precisely, we sort all horizontal (vertical) variable paths in *descending order* by their *priority* and *do not move* a visited node anymore after it was discovered and processed for the first time. The priority of a variable path p is higher if the rectangle of the *start node* of p is located *more eastern* and the rectangle of the *end node* of p is located *more western.*

Definition 4.11 (**Priority of a Variable Path**)**.** Let P and Q be 2 horizontal (vertical) *variable paths* and R = $\{n_1, n_2, ..., n_i\}$ and S = $\{m_1, m_2, ..., m_k\}$ the *rectangles* of their traversed *variable node sets*, respectively. The *priority* of a variable path is defined as positive integer, such that the priority of P is higher as of Q iff *at least one* of the conditions is true:

1. $\mathrm{maxX}(n_1) > \mathrm{maxX}(m_1)$
2. $\mathrm{minX}(n_i) < \mathrm{maxX}(m_k)$
3. the *number of edges* in P is *higher* as in Q

The previous Figure 4.9 shows the horizontal variable paths in the example layout graph *sorted by their priority* in descending order.

After sorting, we balance the priority-sorted *variable paths* list one after another. A single *variable path* is balanced by calculating the arithmetic average of all edge distances starting from the first edge and ending with either the last edge in the path (i) or the last edge that leads to an already processed node (ii). The identified average edge distance is set as edge distance to all encountered edges. If the balancing process was interrupted at edge *e* before reaching the last edge in the path (ii), the balancing process begins again, starting at the next following edge of *e*. Otherwise (i) we continue the balancing process with the next following path.

Figure 4.10 illustrates the *horizontal balancing process* in the example layout graph. The paths and and their containing nodes are balanced in the following color order: *red*, *orange* and lastly *yellow*. All edges show the edge distance *after completing the horizontal balancing*.

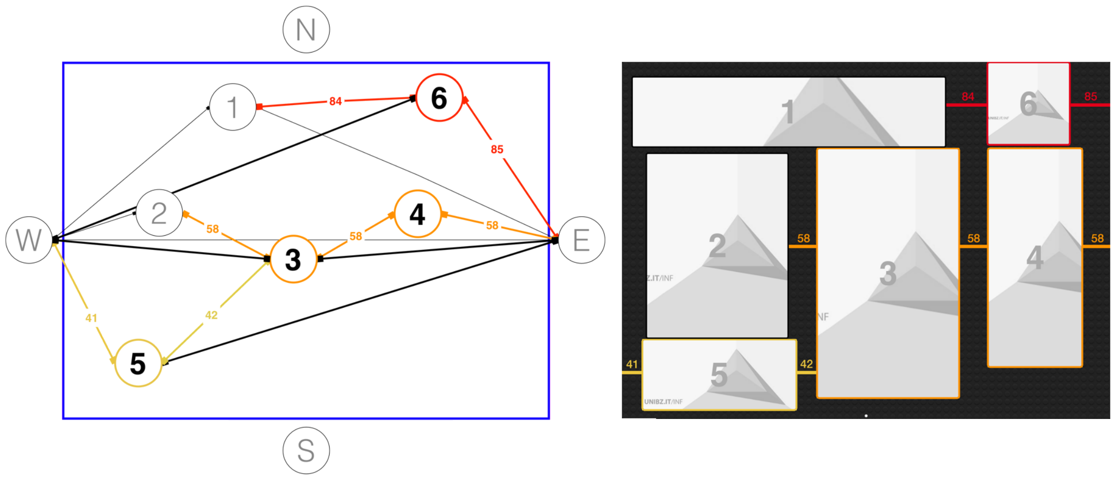


FIGURE 4.10: The layout graph and the resulting rectangle layout after completing the horizontal balancing process. The path balancing was performed in the following order: *red*, *orange*, *yellow*.

Since for *horizontal balancing* we do not need north and south edges, we build only the *horizontal part* of the layout graph (as shown in Figure 4.8), rather than the complete graph with north, south, west and east edges. After horizontal balancing, we create the *vertical part* of the layout graph and extract all *vertical variable paths* with VERTICALVARIABLEPATHS, an algorithm that works analogously to Algorithm 5. Finally, we balance the *vertical variable paths* in the same manner as the *horizontal variable paths*.

4.7 ARITA

ARITA (algorithm 6) uses all the reasoning and algorithms described in this chapter. First, it identifies the set of adjustable rectangles R_{adj} (lines 1-3). If this set is not empty, the algorithm continues by calculating the set of fixed rectangles R_{fix} (line 5), otherwise ARITA terminates and does not perform any changes to the rectangle layout. The algorithm continues by calculating the free spaces in container C, constraint to the rectangles in R_{fix} (line 6). It repositions all rectangles in R_{adj} to a new preliminary location (line 7), which results in a temporary rectangle layout. ARITA continues by building the layout graph on the basis of the temporary rectangle layout (line 8). Subsequently, we concatenate all nodes with their horizontal neighbors and balance the layout graph horizontally (lines 9-10). Finally, the algorithm repeats the last two steps in vertical direction (lines 11-12) and terminates.

Figure 4.11 shows the *final rectangle layout* in landscape interface orientation, after completing ARITA.

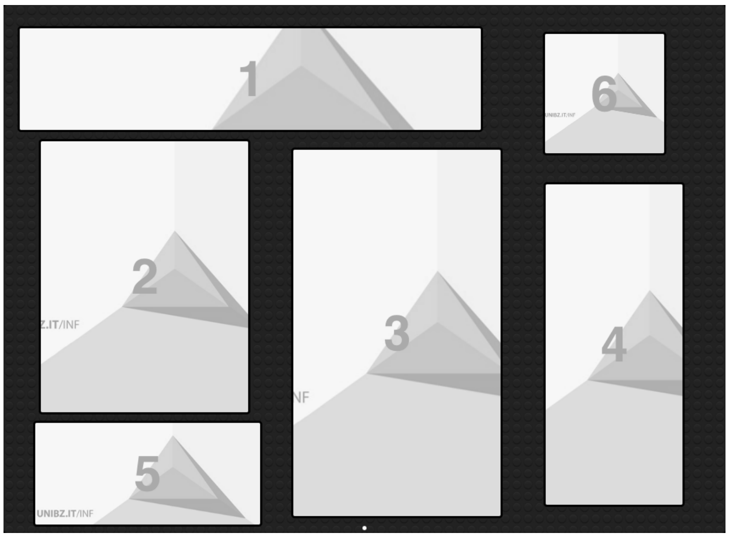


FIGURE 4.11: The resulting rectangle layout in landscape interface orientation completing ARITA.

```
   Input: A container rectangle $C$; a set of rectangles $R$
1  Identify rectangles set $RO$ that are positioned (partially) out of $C$
2  Identify rectangles set $RN$ that do not fulfill the no overflow criteria
3  Let $R_{adj}$ be the set of adjustable rectangles $R_{adj} = RO \backslash RN$
4  if $R_{adj}$ not empty then
5      Let $R_{fix}$ be the set of fixed rectangles $R_{fix} = R \backslash RA$
6      Compute $FS$ = ComputeFreeSpaces(C, $R_{FIX}$)
7      Reposition rectangles with PlaceRectangles(FS, $R_{ADJ}$)
8      Create layout graph $lg$
9      Set west and east neighbor nodes with SetWestAndEastNeighbors(lg)
10     Balance $lg$ horizontally using variable paths of HorizontalVariablePaths(lg)
11     Set north and south neighbor nodes with SetNorthAndSouthNeighbors(lg)
12     Balance $lg$ vertically using variable paths of VerticalVariablePaths(lg)
13 end
```

Algorithm 6: ARITA

4.8 Complexity Analysis

The algorithmic complexity of ARITA can be determined by identifying the complexity of all sub algorithms.

The sweep line based COMPUTEFREESPACES algorithm (algorithm 1) has complexity $O(n^2 \cdot \log n)$, where n denotes the number of rectangles in C. The sweep line stops at

$2n$ event points. At each stop, the algorithm calculates, sorts and processes the *line list*, which can contain at most n lines.

The PLACERECTANGLE algorithm (algorithm 2) has complexity $O(n^2)$. A rectangle layout can contain at most $4n$ free spaces. In the worst case, the algorithm loops through all rectangles, and for each of them all free spaces are accessed.

For the construction of the layout graph, a loop first instantiates all nodes in $O(n)$ time. Then the SETWESTANDEASTNEIGHBORS algorithm (algorithm 3) connects the nodes and has complexity $O(n^2)$. This quadratic complexity comes from looping through all nodes and the complexity of the sub algorithm WESTNEIGHBORS (algorithm 4) of $O(n)$. The sweep line in WESTNEIGHBORS has to stop on at most $2n - 2$ points *or* has to validate at most $n - 1$ vertical lines on a single event stop. Thus, the overall complexity for creating the layout graph is $O(n^2)$.

For the horizontal balancing of the snippets we extract from the layout graph the horizontal variable paths using HORIZONTALVARIABLEPATHS (algorithm 5) that has complexity $O(e)$, where e denotes the number of edges in the layout graph. This complexity results from the fact that every edge is visited only once. Since we are balancing the layout graph by sorting the extracted variable paths, the complexity depends on the number of variable paths p in the graph. Therefore the overall complexity for the horizontal balancing step is $O(e) + O(p * logp)$. The vertical balancing has the same complexity.

By putting all formulas together, we get the overall complexity of ARITA as

$$O(n^2 * logn) + O(e) + O(p * logp)$$

4.9 Adjusting Multiple Pages

As stated in section 2.3, on mobile devices energy efficient implementations are crucial. Therefore, in order to save battery and increase the performance for adjusting *multiple independent rectangle layouts*, we cache the computed layouts with the corresponding interface orientation in a database. We only need to recompute the layout for a specific interface orientation if the user manually repositioned, added or deleted a snippet on a page. Therefore we do not have to perform ARITA on every single device rotation, but can fall back on precomputed results. This reduces CPU load and thus helps to save valuable battery.

Moreover, on device rotation, we compute the rectangle layout for the currently visible page with the highest priority and process remaining pages in the background. This way we achieve a very fluent repositioning impression even on many pages with complex rectangle layouts.

Chapter 5

Evaluation

This chapter evaluates ARITA against running time and user satisfaction. In section 5.1 we perform various experiments to analyze the runtime for different rectangle layouts. Section 5.2 evaluates the layout results of ARITA with a user experience study.

5.1 Runtime Experiments

5.1.1 Setup

We evaluated the running time of ARITA in various experiments with different settings. Table 5.1 lists all parameters under investigation, their default settings and range values. In each experiment we vary only a single parameter (in the defined range) and set all remaining ones to their default values, if not stated differently.

Parameter	Default	Range
Number of rectangles (R_{all})	5	1–20
Number of adjustable rectangles (R_{adj})	50% of R_{all}	0, 1, 2, 3, 4, 5
Maximal rectangle width/height (R_{max})	350	150, 250, 350, 450, 550, 650
Minimal rectangle width/height (R_{min})	75	25, 75, 125, 175, 225, 275
Number of pages (P)	1	1, 2, 4, 8, 12, 16

Table 5.1: The parameters under investigation and their default values.

In each experiment we create a randomly generated portrait snippet layout (with no snippet overlapping) and rotate it to landscape interface orientation. The randomly generated portrait layout is parameterized by the values in Table 5.1.

The default configuration describes a rectangle layout with 5 rectangles. The percentage of adjustable rectangles, i.e., rectangles that after rotation are positioned (partially) out of the screen bounds, is set to 50% (on decimal values we round always up, i.e., to the upwards nearest integer value). Moreover, in the default configuration, we randomly created snippets with height and width values in the interval of [75, 350] points. Since

ARITA considers every page independently from all others, the default configuration describes a snippet layout on a single page.

We performed the following experiments on an iPad (4th gen), which includes a dual core Apple A6X processor. Since we created the size and position of snippets randomly, we repeated every experiment with every parameter configuration *3 times* and used the arithmetic average of the values.

5.1.2 Results

Figure 5.1 measures the effect of number of rectangles (ranging from 1 to 15 rectangles) for repositioning. In the default settings (*orange curve*) ARITA needs from 33ms (1 rectangle) to 799ms (15 rectangles) to compute the landscape layout. The *blue curve* shows the behavior of ARITA on layouts where all rectangles are out of screen bounds and therefore must be adjusted. The *green curve* depicts layout situations with only one adjustable rectangle. We can observe that ARITA performs best if all rectangles are adjustable, and it takes the most time to compute layouts where all rectangles, except one, are fixed. The difference between the 3 curves grows with increasing number of rectangles.

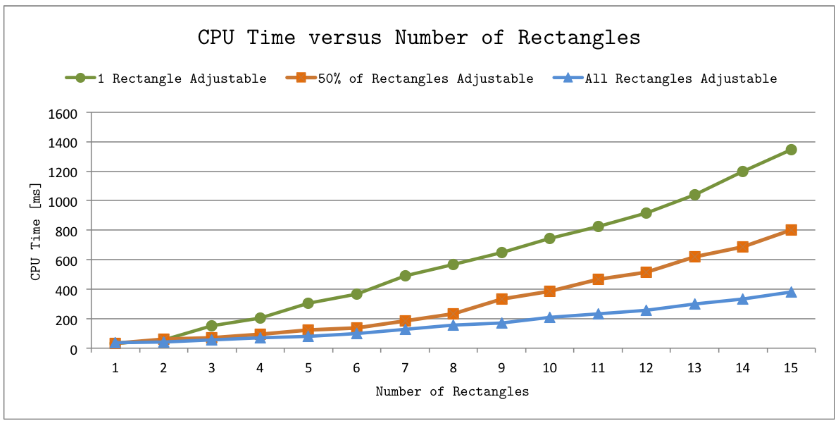


FIGURE 5.1: CPU time versus number of rectangles for ARITA with 3 different settings for adjustable rectangles.

In order further analyze the different performance behavior when the number of adjustable rectangles changes, we increased the number of adjustable rectangles step by step. Figure 5.2 shows the needed CPU time for a 5 rectangles layout with increasing percentage of adjustable rectangles. If after rotation no rectangles are placed out of screen bounds, ARITA terminates almost immediately (6ms). If one rectangle must be adjusted, the CPU time peaks to 131ms and falls continuously with every additional adjustable rectangle until 69ms, when all of the 5 rectangles must be adjusted.

In order to find the reason for this behavior, we performed the same experiment on a more complex rectangle layout (12 rectangles) and measured the CPU time for the two main parts of ARITA:

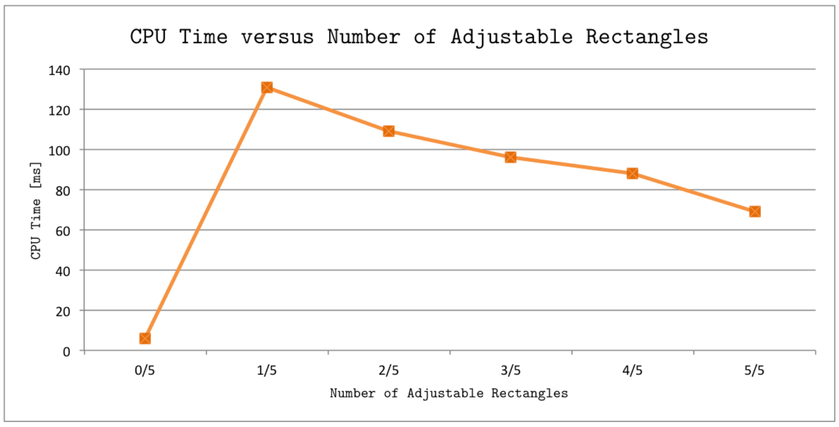


Figure 5.2: CPU Time versus Number of Adjustable Rectangles for ARITA.

1. **Placing rectangles**: Thus includes the initial checks for identifying all *adjustable rectangles*, the *free spaces* calculation (sweep line) and the selection and placement of rectangles in a free space container.
2. **Adjusting rectangles**: Thus includes the calculation and balancing of the *layout graph*, i.e., the extraction and sorting of variable paths from the graph and the modifications on the edge distances.

Figure 5.3 shows the runtime to adjust the layout with 12 rectangles with increasing percentage of adjustable rectangles, split in the two main parts (*placing rectangles* and *adjusting rectangles*) of ARITA.

The curve reaches again its maximum (624ms), if one rectangle is adjustable and all others are fixed. We can also observe that, at the curves peak, almost 84% of the runtime was needed to complete the placing of rectangles. The remaining 16% of the time was used for adjusting the rectangles. As soon as the number of adjustable rectangles increases, the needed CPU time for placing the rectangles falls continuously, whereas the CPU time for adjusting the rectangles increases only slightly. As soon as 11 out of 12 rectangles are adjustable, the needed CPU time for placing the rectangles (127ms) is smaller than the time for adjusting the rectangles (155ms).

This experiment proves that the CPU time for placing rectangles is very sensitive to the number of fixed rectangles in the layout. More fixed rectangles lead to more events in the sweep line processing, which again lead to more extensive free spaces processing, such as processing, splitting and closing of active spaces. In addition, more free spaces in the rotated interface lead to a costlier free space container selection, when positioning adjustable rectangles.

By contrast, the adjusting rectangles step is relatively stable with increasing number of adjustable rectangles. This is founded in the fact that, for the construction of the layout graph, it makes no difference if the layout is composed of *only fixed*, *only variable* or *fixed and variable* nodes (rectangles). The slightly CPU time increase in the adjusting rectangles step for layouts with a higher percentage of adjustable rectangles is justified

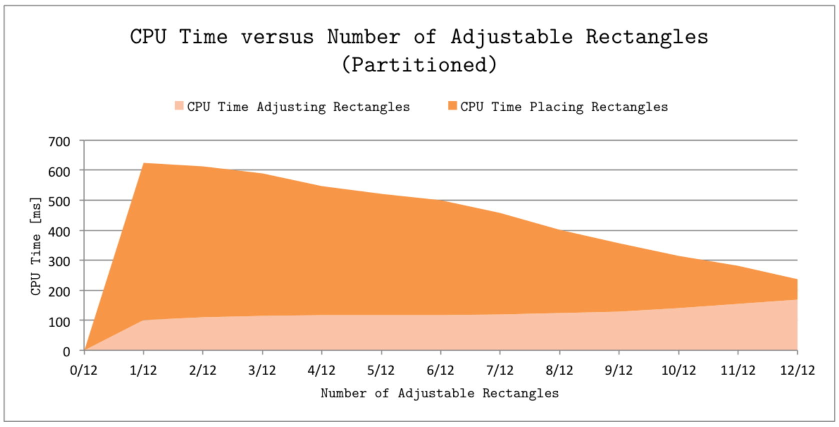


FIGURE 5.3: CPU time versus number of adjustable rectangles partitioned in the two main parts of ARITA.

by the higher number of variable paths in the graph. This consequently leads to a certain higher effort to extract, sort and balance all variable nodes (rectangles) in the graph.

In order to measure the effect of rectangle sizes in the layout, we ranged the maximum and minimum width and height limits for generating rectangles. Since this experiment depends strongly on the randomly generated position and size of the rectangles, we repeated every configuration 5 times instead of 3 times and used again the arithmetic average of the values.

Figure 5.4 shows the effect of decreasing/increasing the size of rectangles and fixing the number of rectangles in the layout. We could not find any significant differences in terms of needed CPU time. In the average, ARITA performs on layouts with small or big rectangles almost identical.

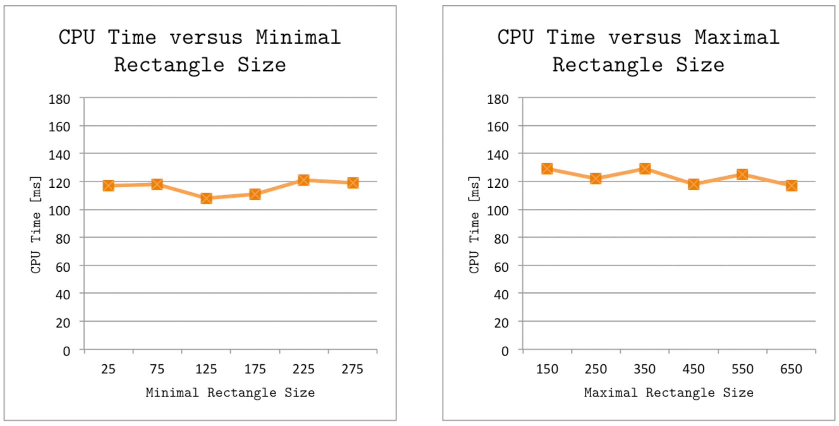


FIGURE 5.4: CPU time versus rectangle size for ARITA.

All the experiments presented so far used a single page. In reality, in NetSnips users can place their web snippets on up to 20 pages. But since the snippet layouts on different pages do not depend from each other, ARITA can process pages independently and sequentially.

We performed an experiment with default layout parameters and increased the number of pages. Figure 5.5 shows the result of performing this experiment sequentially (one page was processed after another on a single CPU core) and in parallel (pages are processed in parallel on two independent CPU cores).

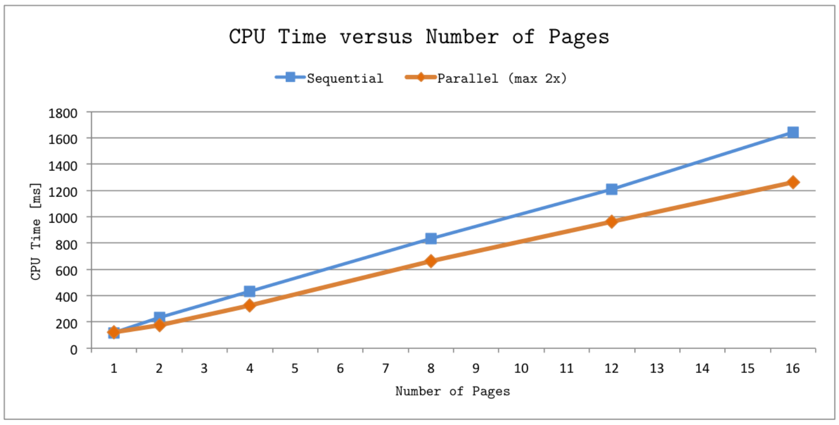


FIGURE 5.5: CPU time versus number of pages for ARITA.

As expected, the sequential processing (blue line) shows a linear increase in CPU time, by adding approximately 100ms for every additional page. The parallel processing is limited to the two CPU cores of the iPad, therefore we expected again a linear but much flatter increase of needed CPU time; ideally 50% faster in comparison to the sequential processing. However, the parallel processing (orange line) shows a decrease in the elapsed CPU time, but only at a maximum of approximately 25%. To implement the parallel page processing we used Grand Central Dispatch (GCD), a high-level Apple technology for multithreading, by simply dispatching the processing of each page on the same concurrent operation queue. This leads to separate threads for every page, which introduced some overhead in spawning and closing threads. A solution with two *fixed threads* separated on the two CPU cores, which process ARITA on all pages in parallel, might lead to a higher reduction in CPU time.

Please note that, as described in section 4.9, it is very uncommon that we actually need to *compute* a new layout for *all pages* in one single device rotation, since we save and reuse the computed layouts for both interface orientations in a database on the device.

5.2 User Experience Evaluation

Due to the fact that it is difficult to validate the layout results of ARITA objectively, we decided to perform a user experience study.

5.2.1 Setup

In order to discuss and decide conceptual, practical and ethical issue in the evaluation, we used the DECIDE framework of Rogers et al. [22], which structures evaluation studies in the following points:

- ***D*etermine overall goals of the evaluation:** The purpose of this evaluation is to find out how users react on the auto-layouting behavior of ARITA. Moreover we focus on the general satisfaction with the *automatically generated* landscape layouts, which are based on the *user defined* portrait layouts. This evaluation should help us to examine the current level of user satisfaction of ARITA, to find out the strengths and weaknesses of our approach and to refine it in future versions.

- ***E*xplore the specific questions to be answered:** Do users *expect* the automatically generated landscape snippet layout in comparison to the user-defined portrait snippet layout? Do users perceive the new layout as *clear* (i.e. well-structured, comprehensible) and *equally distributed*? Are user generally satisfied with the automatically generated snippet layout?

- ***C*hoose the evaluation paradigm and techniques to answer questions:** Since we focus only on the evaluation of the automatically generated snippet layouts, we use a **questionary** in which the user has to evaluate **20 different situations regarding 4 criteria**. Every situation (in the following references as *case*) depicts 2 screenshots from an iPad device running NetSnips. The first one shows a *user-defined* snippet layout in portrait screen orientation. The second shows a screenshot of the *automatically generated* landscape layout computed by ARITA. The task of the user is to evaluate the *automatically generated* landscape layout on a scale of 1-5 (1 = worst; 5 = best) regarding the criteria *expectation*, *clarity*, *equally distribution* and *overall satisfaction*.

- ***I*dentify the practical issues that must be addressed:** Since we are convinced that only users that are actually familiar with NetSnips can evaluate the automatically generated layouts, we asked real **NetSnips users around the world** to participate in this evaluation. We accomplish this by presenting our participation request on the NetSnips web site[1] and by inviting persons that follow the NetSnips fan pages on Twitter[2] and Facebook[3]. The participants have to rate the 20 example cases based on 4 criteria. Ideally users would not evaluate these cases by using screenshots, but rather on examples running on real devices. However this would be difficult to accomplish, for instance due to technical limitations. iOS devices do not allow users to simply download and run a demo/test app from a webpage (due to security concerns iOS devices allow only installations from the Apple App Store).

- ***D*ecide how to deal with the ethical issues:** Participants understand that their ratings are used in anonymized and summarized form. Since users should send their results per email after completing their ratings, they can also change their mind and decide not to participate in this study.

- ***E*valuate, interpret and present the data:** An issue of this evaluation approach is certainly that some user might evaluate snippet layouts differently if they use only screenshots, rather than actual devices with a test application. Since

[1]http://netsnipsapp.com/2013/09/05/netsnips-needs-your-help/
[2]http://twitter.com/NetSnipsApp
[3]http://www.facebook.com/NetSnipsApp

we involve only users that are familiar with NetSnips, we assume that the participants in this study are able to evaluate all depicted cases the same way as they would do it on their own device. We use the received data for analyzing cases that receive the most positive as well as critical ratings. We are also interested in cases that receive very diverse ratings among the 4 criteria.

5.2.2 Results

Users could participate in the study for **one week** (05.09.13 - 12.09.13), in which we received and analyzed results from **31 participants**. Please note that all 20 cases can be found in Appendix B. Figure 5.6 shows the average results for every case for the 4 criteria:

- **Expected**: *"Do you expected the landscape layout? Was it foreseeable?"*
- **Clear**: *"Do you think the landscape layout is ordered and comprehensible?"*
- **Equally distributed**: *"Are the snippets in the landscape layout equally distributed?"*
- **Overall satisfaction**: *"What is your overall satisfaction with the landscape layout?"*

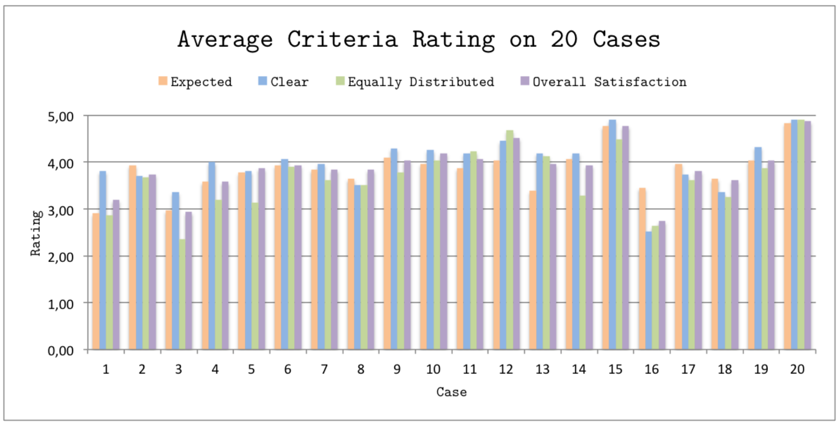


Figure 5.6: Average criteria rating on 20 cases.

Even though we are aware that the rating of layouts based on our criteria is subjective and depends greatly on the users point of view, we could identify cases that lead to a generally higher (case 12, 15 and 20) or lower (case 1, 3 and 16) average satisfaction. The cases that lead to higher satisfaction are generally layouts with many smaller snippets, for which ARITA could identify a layout without snippet overlapping. The cases with the worst satisfaction are these ones for that no solution without overlapping exists. Moreover cases that lead to overlapping or overflow, only because the fixed snippets were not adjusted, were rated generally lower.

It was also interesting that some cases received quite diverse ratings for the 4 criteria. For instance case 16 (shown in Figure 5.7) received higher ratings for *expected* (3.45

+/- 0.86) and illustrates a situation that could not be solved without introducing some snippet overlapping. In this situation ARITA does not provide good results. The layout problem with these 3 very broad snippets could be solved better (i.e. solved with fewer overlapping) if the fixed snippets would be moved just slightly up and down, respectively. We interpret the rating results, that users *expected* that the snippets in the layout overlap somehow, but nevertheless they are *not very satisfied* with the result.

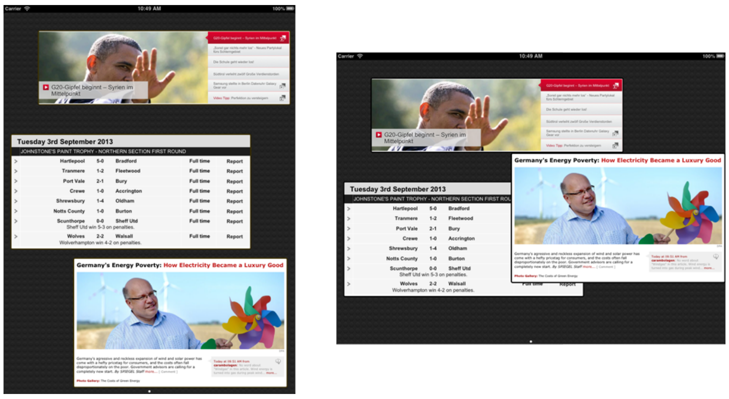

G20-Gipfel beginnt – Syrien im Mittelpunkt
Tuesday 3rd September 2013
Germany's Energy Poverty: How Electricity Became a Luxury Good

FIGURE 5.7: Case 16. Higher ratings for *expected*; lower ratings for *overall satisfaction* (most probably due to unnecessarily high overlapping).

Case 12 (shown in Figure 5.8) received generally high ratings for all criteria, especially for *equally distribution* (4.68 +/- 0.48). ARITA produces a layout with no overlapping by repositioning 2 out of 5 snippets. Most probably, users perceive this layout as equally distributed, since there is enough free space for centering the repositioned snippets vertically and horizontally in relation to other snippets and screen bounds.

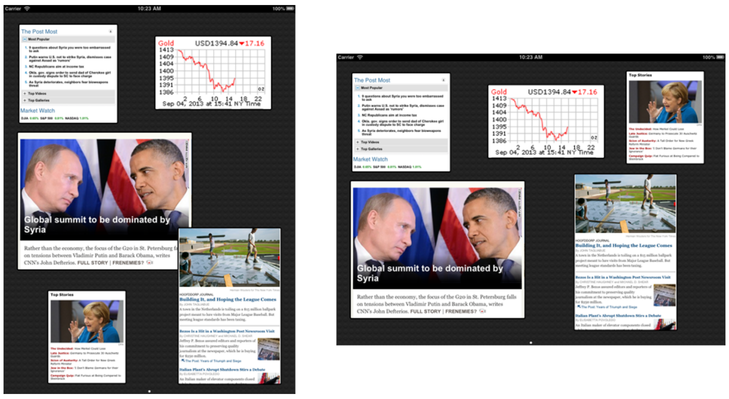


FIGURE 5.8: Case 12. High ratings for all criteria.

A diversely rated situation was case 1 (shown in Figure 5.9). The landscape layout with 4 snippets (2 where repositioned), received higher ratings for its clarity (3.81 +/- 0.94), but lower ones for expectation (2.90 +/- 0.90) and equal distribution (2.87 +/- 1.07). We have asked some users about their motivation for their ratings: even though the snippet at the bottom (depicts a cutout of Facebook) was perfectly horizontally centered on the screen, they felt it was displaced and should be placed in horizontal alignment with the other snippets. Again we interpreted these results, that even though users felt that the automatically generated layout is relatively *clear* and *well-structured*, it's not the layout that they *expected* and perceived as *equally distributed*.

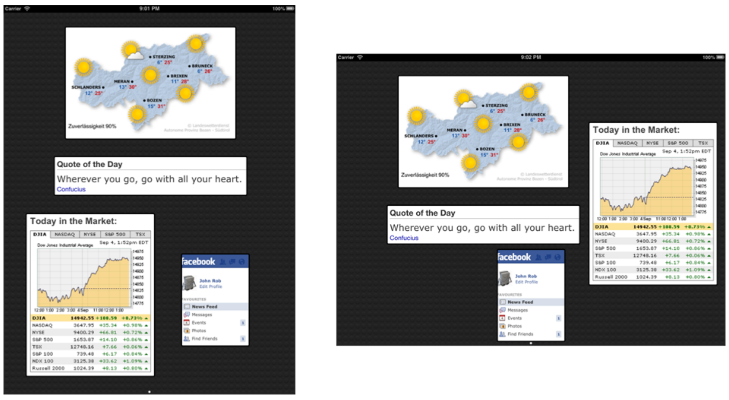


FIGURE 5.9: Case 1. Diverse ratings for *equal distribution.*

Figure 5.10 shows the average criteria ratings for all cases. On average users rate the automatically generated layouts for the overall satisfaction criteria with 3.87 +/- 0.52 (*average case satisfaction with standard deviation over all cases*). Users generally give better ratings for the clarity of the layout (3.98 +/- 0.55) as for the equal distribution of snippets (3.66 +/- 0.66). The average consensus between all participants can be described with the *standard deviation over all ratings in a particular criteria* and is 0.88 for expected, 0.82 for clarity, 0.85 for equal distribution and 0.77 for overall satisfaction.

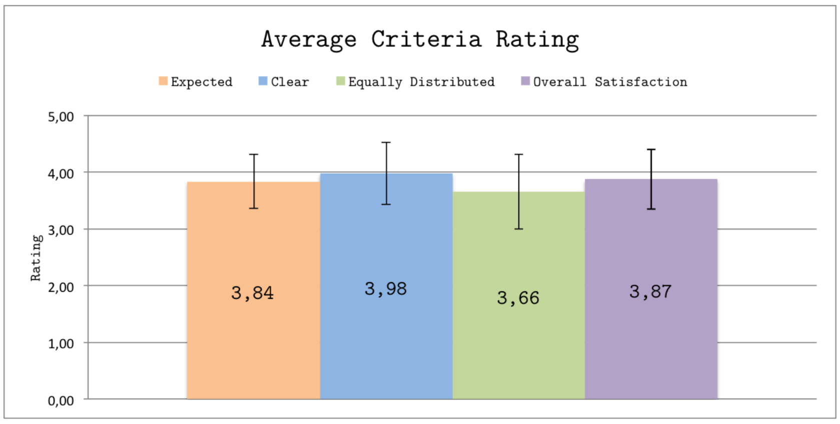


FIGURE 5.10: Average criteria rating for all cases.

5.2.3 Discussion and Lessons Learned

After discussing with some participants about their motivation behind their ratings, we learned that some cases were seen very differently by the participants. For instance, some users really appreciated that ARITA does not reposition snippets that stay in screen bounds after rotation. But others stated that in some situations they would expect that these snippets are also adjusted, at least slightly. Regarding to these users, this is especially important when a possible movement of fixed snippets would lead to a layout with no overlapping.

The criteria *equally distributed* was also a controversial concept, since some users are very sensitive for free space "gaps", others prefer that snippets are always centered and still others like that ARITA centers only the repositioned snippets in the available free space.

In general we are satisfied with the results that we gathered in this study. Nonetheless this user study led to many ideas how to improve ARITA. Future improvements should focus on the following points:

- **Balance margins of fixed rectangles**: Many participants proposed to balance also the margins of fixed snippet, i.e. snippets that stay in screen bounds after rotation. In some cases this might lead to a more harmonious appearance of the snippet layout. At the same time the user specified ordering of all snippets would not be changed dramatically, since in general, the balancing process repositions snippets only slightly. Moreover this would enhance layouts, where repositioning of fixed snippets leads to fewer or no overlapping.
- **Introduce satisfaction function for balance possibilities**: ARITA balances the layout without introducing additional overlapping. However, the current implementation does not validate different balancing possibilities, but uses simply on of these. A satisfaction function could score the different possibilities, which might lead to better visual results in some situation (see Figure 5.9 and its associated discussion).
- **Add setting for repositioning style**: Some users prefer that the algorithm repositions all snippets in a specific style (e.g. all snippets centered, snippets in increasing size from left to right), rather than minimizing the changes to the layout. This could be added with a setting for the repositioning style.

Chapter 6

Conclusion

6.1 Summary

In this thesis we proposed ARITA, a novel repositioning algorithm that adapts the position of rectangular web snippets on orientation changes of the user interface. Its feasibility was proven by implementing it in the mobile application NetSnips.

Even though ARITA follows specific constraints in order to fulfill our requirements, various concepts and sub algorithms may be applicable in other contexts. We have developed a sweep line based algorithm that identifies the largest free spaces between rectangles in a given container rectangle. We have elaborated a heuristic to place rectangles in a set of free space candidates. Moreover we have developed an algorithm that builds a graph that describes a rectangle layout and elaborated a strategy for balancing the horizontal and vertical margins between fixed/variable rectangles and screen bounds.

We have evaluated the runtime of ARITA for different rectangle layouts with various experiments on real devices. The experiments show that ARITA provides good performance results in realistic layout situations for NetSnips.

We have also evaluated the layout results with a user experience evaluation. The participants of the evaluation were users from the NetSnips community. The results confirm our assumption that the assessment of automatically generated layouts depends strongly on personal preferences. For instance, some users prefer an algorithm that tries to perform minimal changes on the original layout (like ARITA), whereas others favor complete repositioning of all snippets. However our results show also that, in general, the participants of your study validate the layout results of ARITA from satisfiable to very good.

6.2 Future Work

This work is a first attempt to address the complex problem of repositioning rectangles where no objectively best solution exists. The judgment of rectangle layouts depends strongly on user preferences, habits and general usage of the application.

Future work could *relax or alter some of the requirements* that we introduced. For instance, the constraint that prohibits the margin balancing of fixed snippets (i.e., snippets that remained in screen bounds after rotation) could be removed. In some cases this might lead to a more harmonious visual appearance of the snippet layout. At the same time the user specified ordering of all snippets would not be changed dramatically, since in general, the balancing process repositions snippets only slightly. Moreover this would enhance layouts, where repositioning of fixed snippets leads to fewer or no overlapping.

Another extension might introduce a *satisfaction function* for the common situation of various balancing possibilities in a snippet layout. The satisfaction function could be used to pick the most appropriate solution for a specific case, by scoring the different possibilities. This might lead to better visual results in some situations. ARITA, in its current implementation, does not validate different balancing possibilities, but simply uses one of the solutions that introduces no additional overlapping.

In order to give users more control which overall strategy should be used for automatic snippet repositioning, a *setting for the repositioning style* could be introduced. Other strategies might reposition all snippets in the layout following a specific pattern (e.g. all snippets centered, snippets in increasing size from left to right), rather than minimizing the changes to the original layout.

A new version of NetSnips will include the research work of this thesis and will be available on the Apple App Store shortly. After the public release, we will work on future improvements for the repositioning algorithm. The specific refinements will be defined after evaluating reviews and feedback of the established NetSnips user base.

References

[1] George H Forman and John Zahorjan. The challenges of mobile computing. *Computer*, 27(4):38–47, 1994.

[2] GB Mathews. On the partition of numbers. *Proceedings of the London Mathematical Society*, 1(1):486–490, 1896.

[3] GS Lueker. Two np-complete problems in nonnegative integer programming, report 178 (a6). *Computer Science Laboratory, Princeton University, Princeton, NJ*, 1975.

[4] Jens Egeblad and David Pisinger. Heuristic approaches for the two-and three-dimensional knapsack packing problem. *Computers & Operations Research*, 36(4): 1026–1049, 2009.

[5] Jens Egeblad. *Heuristics for Multidimensional Packing Problems.* PhD thesis, Københavns UniversitetKøbenhavns Universitet, Det Naturvidenskabelige FakultetFaculty of Science, Datalogisk InstitutDepartment of Computer Science, 2008.

[6] Andreas Bortfeldt and Tobias Winter. A genetic algorithm for the two-dimensional knapsack problem with rectangular pieces. *International Transactions in Operational Research*, 16(6):685–713, 2009.

[7] Bernard Chazelle. The bottomn-left bin-packing heuristic: An efficient implementation. *Computers, IEEE Transactions on*, 100(8):697–707, 1983.

[8] Patrick Healy, Marcus Creavin, and Ago Kuusik. An optimal algorithm for rectangle placement. *Operations Research Letters*, 24(1):73–80, 1999.

[9] Walter Hower and Winfried H Graf. A bibliographical survey of constraint-based approaches to cad, graphics, layout, visualization, and related topics. *Knowledge-Based Systems*, 9(7):449–464, 1996.

[10] Pat Langley and Haym Hirsh. User modeling in adaptive interfaces. *Courses and lectures-international centre for mechanical sciences*, pages 357–370, 1999.

[11] Jack Kustanowitz and Ben Shneiderman. Hierarchical layouts for photo libraries. *Multimedia, IEEE*, 13(4):62–72, 2006.

[12] Baback Moghaddam, Qi Tian, Neal Lesh, Chia Shen, and Thomas Huang. Visualization and layout for personal photo libraries. In *International Workshop on Content-Based Multimedia Indexing (CBMI'01)*, 2001.

[13] Qi Tian, Baback Moghaddam, and Thomas S Huang. Display optimization for image browsing. In *Multimedia Databases and Image Communication*, pages 167–176. Springer, 2001.

[14] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 21–21, 2010.

[15] James Burr, Laszlo Gal, Ramsey Haddad, Jan Rabaey, and Bruce Wooley. Which has greater potential power impact: High-level design and algorithms or innovative low power technology?(panel). In *Proceedings of the 1996 international symposium on Low power electronics and design*, page 175. IEEE Press, 1996.

[16] Sanjay Udani and Jonathan M Smith. Power management in mobile computing (a survey). *Technical Reports (CIS)*, page 164, 1996.

[17] Katie Roberts-Hoffman and Pawankumar Hegde. Arm cortex-a8 vs. intel atom: Architectural and benchmark comparisons. *Dallas: University of Texas at Dallas*, 2009.

[18] Christopher A Sanchez and Russell J Branaghan. Turning to learn: Screen orientation and reasoning with small devices. *Computers in Human Behavior*, 27(2): 793–797, 2011.

[19] Jun Gong and Peter Tarasewich. Guidelines for handheld mobile device interface design. In *Proceedings of DSI 2004 Annual Meeting*, pages 3751–3756. Citeseer, 2004.

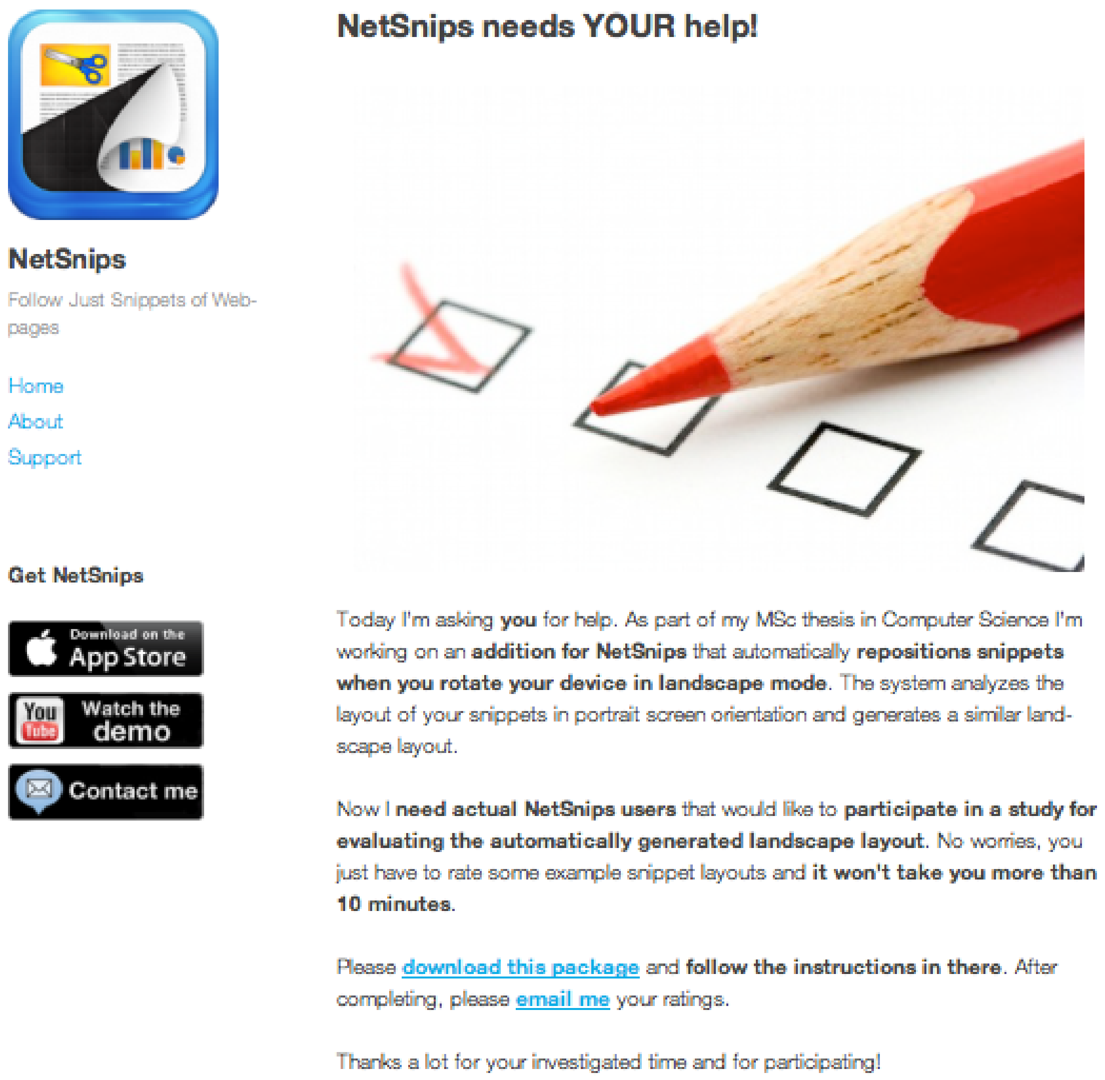
[20] Michael Ian Shamos and Dan Hoey. Geometric intersection problems. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*, pages 208–215. IEEE, 1976.

[21] Shimon Even. *Graph Algorithms.* Cambridge University Press, 1979. ISBN 9781139504157.

[22] Y. Rogers, H. Sharp, and J. Preece. *Interaction Design: Beyond Human-Computer Interaction.* John Wiley and Sons Ltd, 2002.

Appendix A

Evaluation Questionary


NetSnips needs YOUR help!
NetSnips
Follow Just Snippets of Web-pages
Home
About
Support
Get NetSnips
Download on the App Store
Watch the demo
Contact me

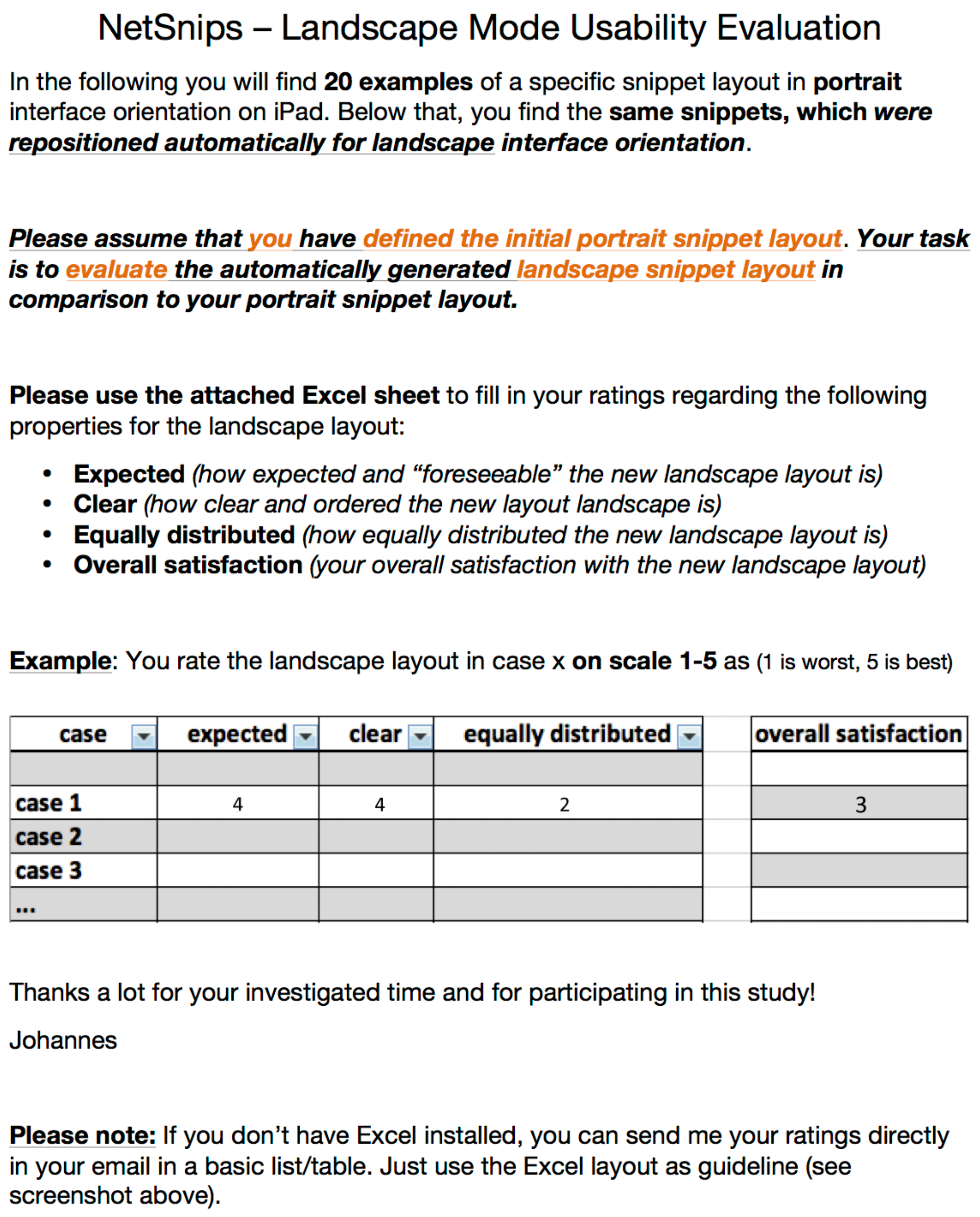
Today I'm asking **you** for help. As part of my MSc thesis in Computer Science I'm working on an **addition for NetSnips** that automatically **repositions snippets when you rotate your device in landscape mode**. The system analyzes the layout of your snippets in portrait screen orientation and generates a similar landscape layout.

Now I **need actual NetSnips users** that would like to **participate in a study for evaluating the automatically generated landscape layout**. No worries, you just have to rate some example snippet layouts and **it won't take you more than 10 minutes**.

Please download this package and **follow the instructions in there**. After completing, please email me your ratings.

Thanks a lot for your investigated time and for participating!

FIGURE A.1: Home page of www.netsnipsapp.com on September 5, 2013.



NetSnips – Landscape Mode Usability Evaluation

In the following you will find **20 examples** of a specific snippet layout in **portrait** interface orientation on iPad. Below that, you find the **same snippets, which *were repositioned automatically for landscape* interface orientation**.

Please assume that you have defined the initial portrait snippet layout. Your task is to evaluate the automatically generated landscape snippet layout in comparison to your portrait snippet layout.

Please use the attached Excel sheet to fill in your ratings regarding the following properties for the landscape layout:

- **Expected** *(how expected and "foreseeable" the new landscape layout is)*
- **Clear** *(how clear and ordered the new layout landscape is)*
- **Equally distributed** *(how equally distributed the new landscape layout is)*
- **Overall satisfaction** *(your overall satisfaction with the new landscape layout)*

Example: You rate the landscape layout in case x **on scale 1-5** as (1 is worst, 5 is best)

case	expected	clear	equally distributed		overall satisfaction
case 1	4	4	2		3
case 2					
case 3					
...					

Thanks a lot for your investigated time and for participating in this study!

Johannes

Please note: If you don't have Excel installed, you can send me your ratings directly in your email in a basic list/table. Just use the Excel layout as guideline (see screenshot above).

FIGURE A.2: The study description for participants.

Appendix B

Evaluation Cases

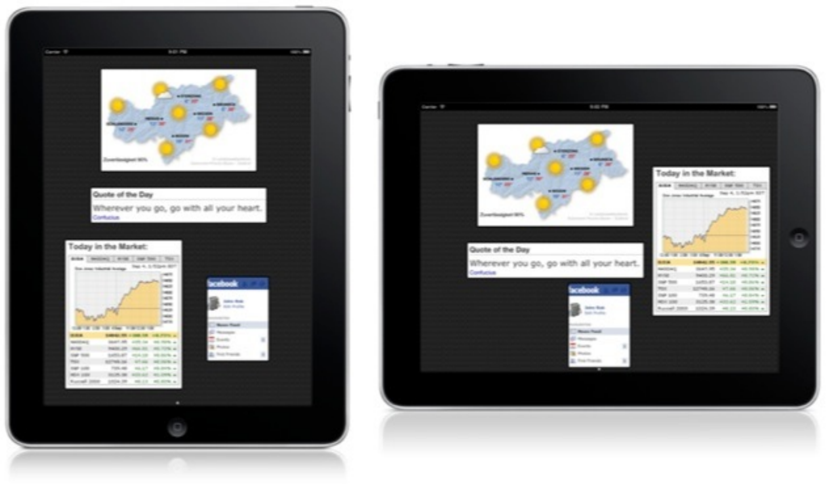


FIGURE B.1: Case 1

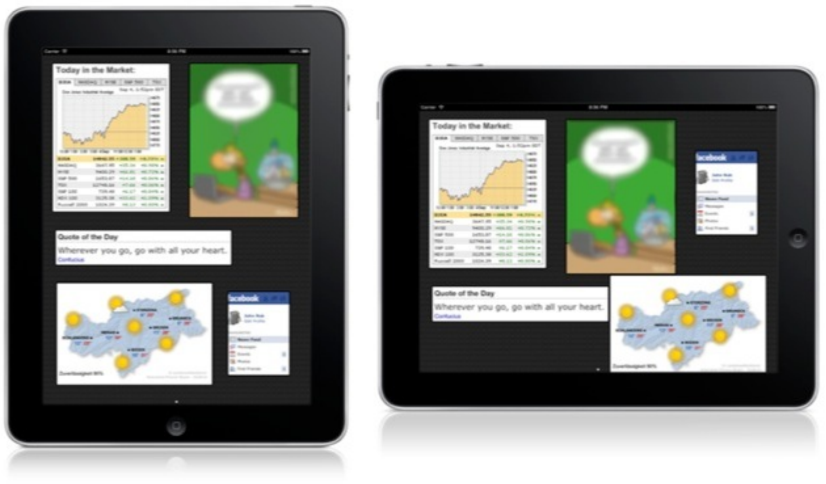


FIGURE B.2: Case 2

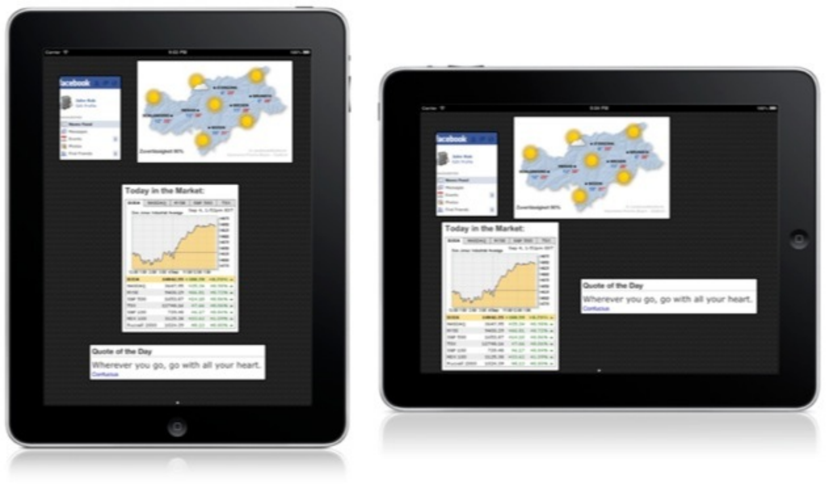


FIGURE B.3: Case 3

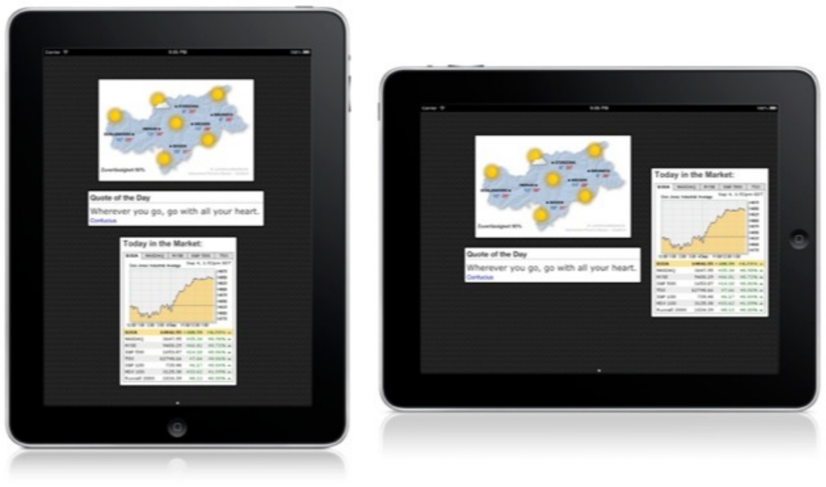


FIGURE B.4: Case 4

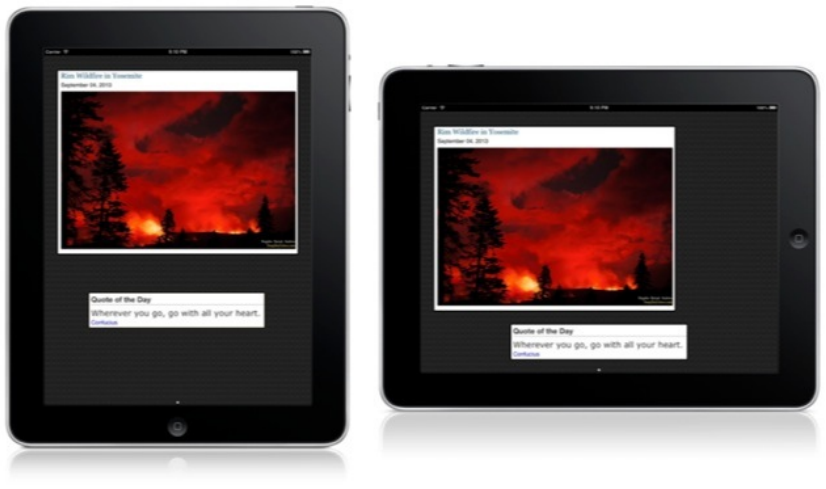


FIGURE B.5: Case 5

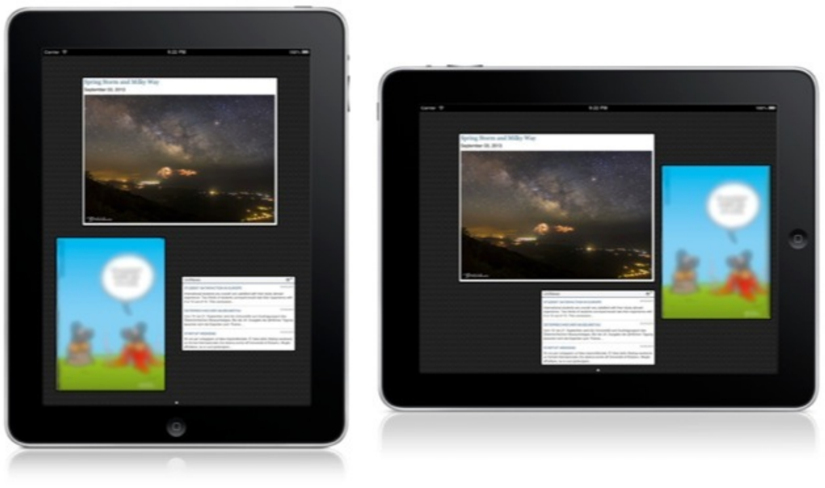


FIGURE B.6: Case 6

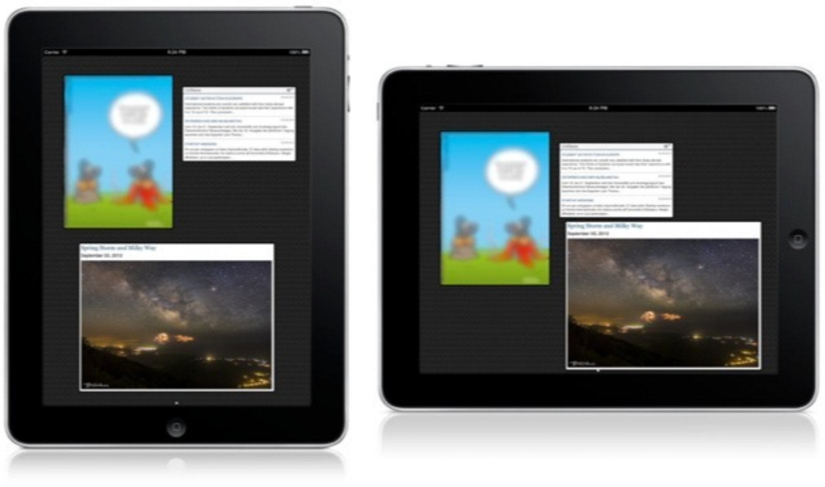


FIGURE B.7: Case 7

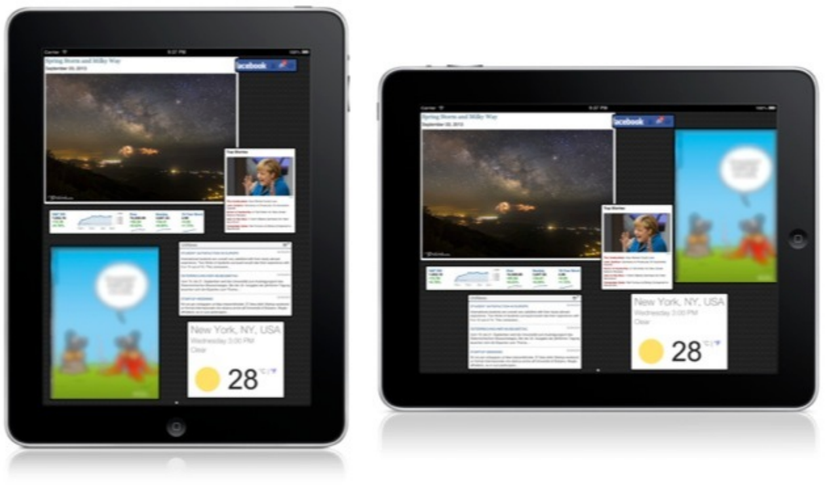


FIGURE B.8: Case 8

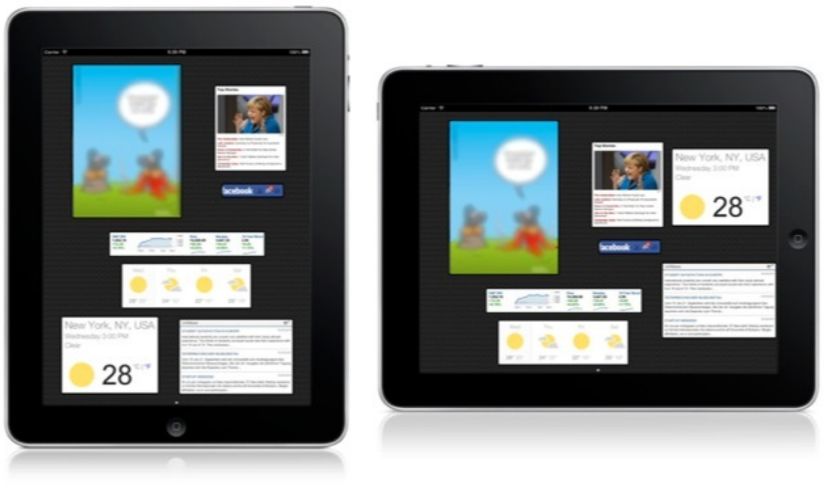


FIGURE B.9: Case 9

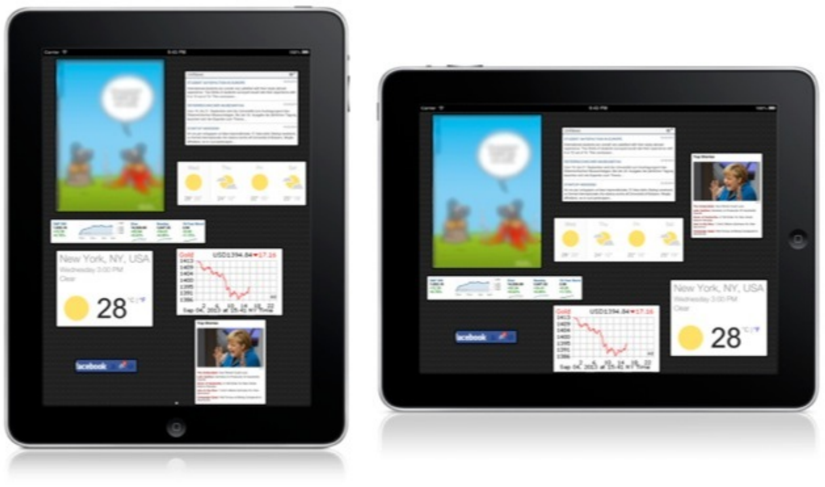


FIGURE B.10: Case 10

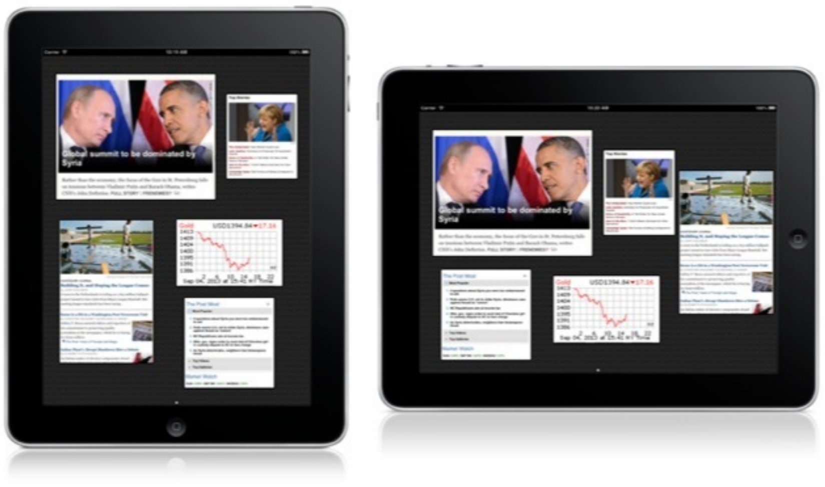

Global summit to be dominated by Syria
Global summit to be dominated by Syria

FIGURE B.11: Case 11

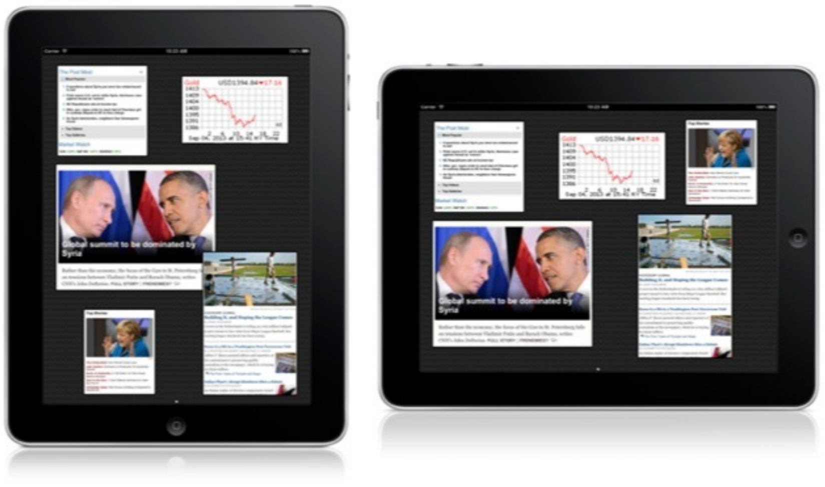


FIGURE B.12: Case 12

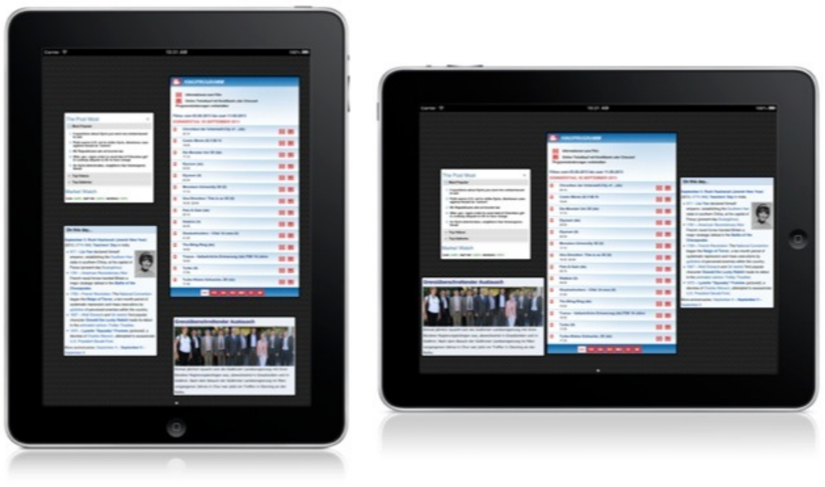


FIGURE B.13: Case 13

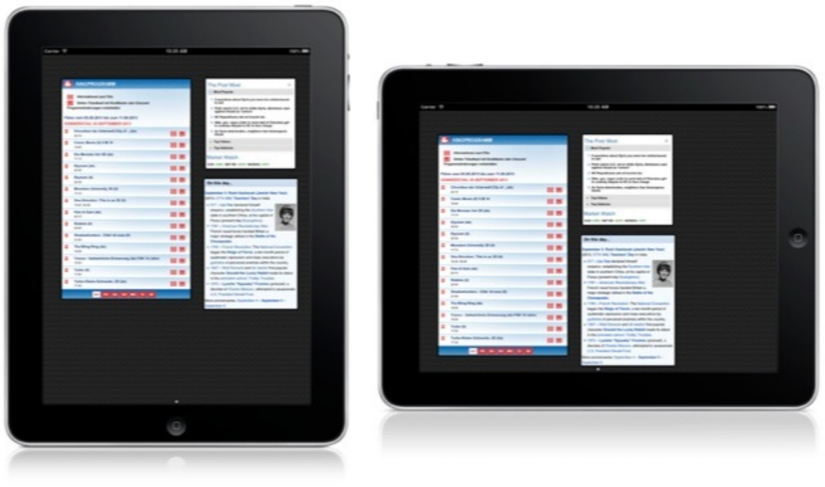


FIGURE B.14: Case 14

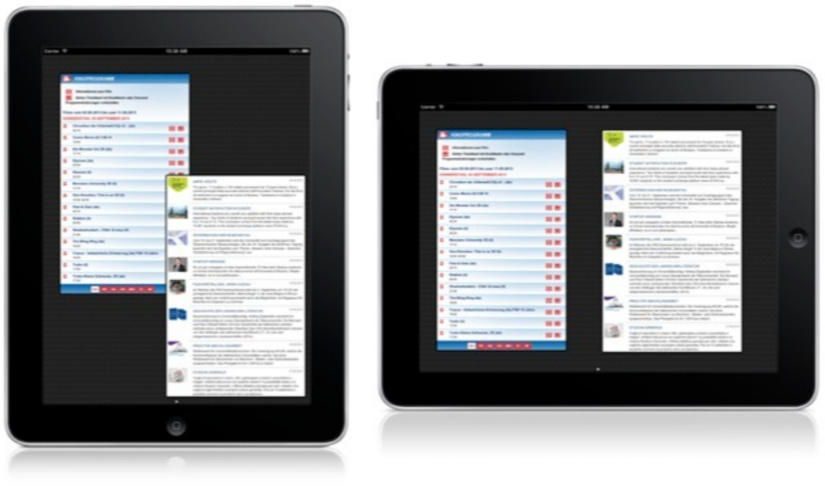


FIGURE B.15: Case 15

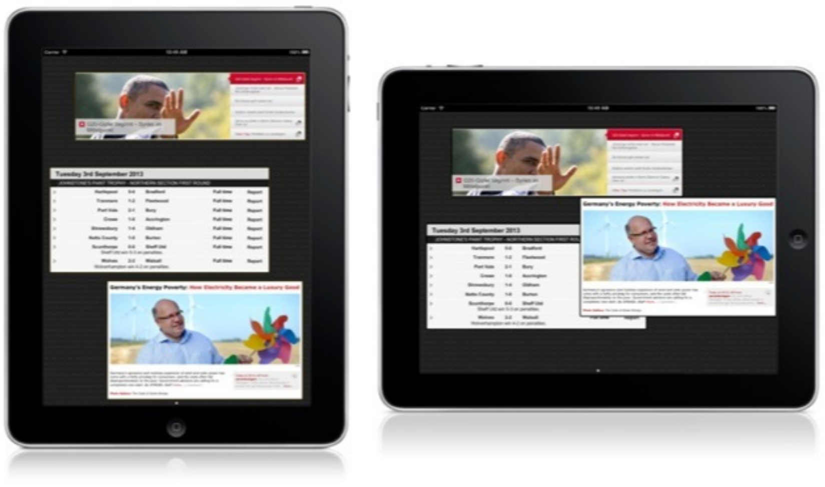
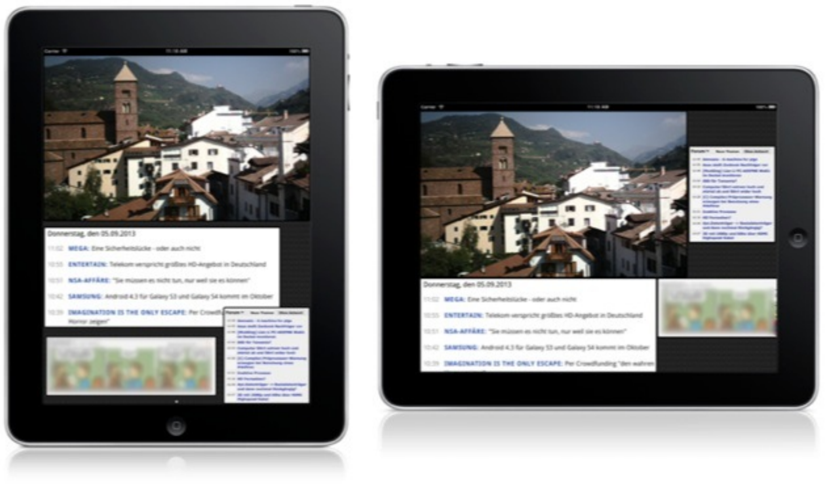


FIGURE B.16: Case 16



Donnerstag, den 05.09.2013

11:02 MEGA: Eine Sicherheitslücke - oder auch nicht

10:55 ENTERTAIN: Telekom verspricht größtes HD-Angebot in Deutschland

10:51 NSA-AFFÄRE: "Sie müssen es nicht tun, nur weil sie es können"

10:42 SAMSUNG: Android 4.3 für Galaxy S3 und Galaxy S4 kommt im Oktober

10:39 IMAGINATION IS THE ONLY ESCAPE: Per Crowdfunding "den wahren Horror zeigen"

FIGURE B.17: Case 17

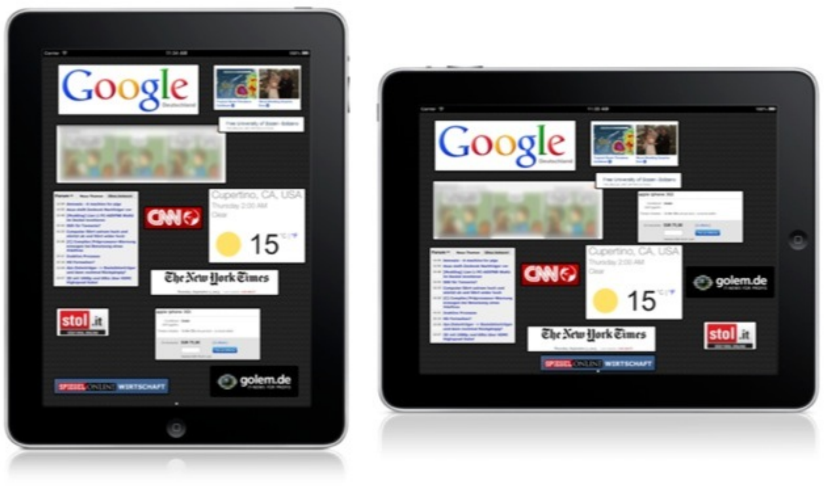


FIGURE B.18: Case 18

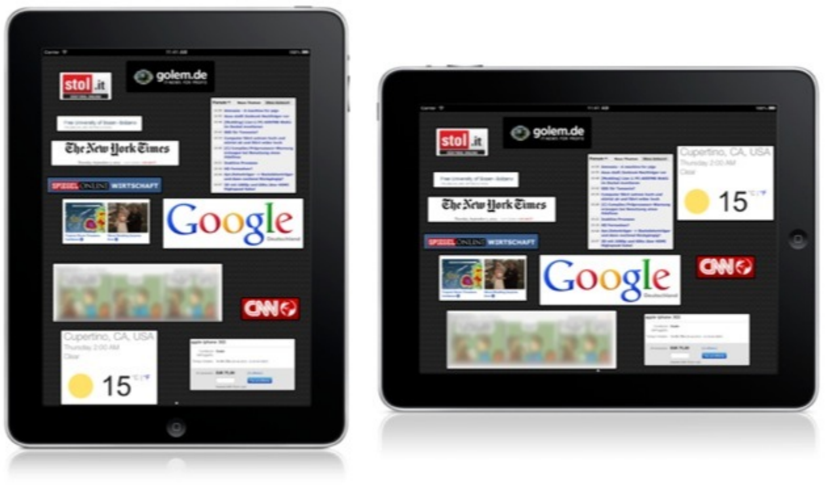


FIGURE B.19: Case 19

FIGURE B.20: Case 20