AN INTELLIGENT LAYOUT ALGORITHM FOR VARIABLE SCREEN RESOLUTIONS

DEĞİŞKEN EKRAN ÇÖZÜNÜRLÜKLERİ İÇİN AKILLI BİR YERLEŞTİRME ALGORİTMASI

BARIŞ ÇELİK

ASST. PROF. DR. BURKAY GENÇ
Supervisor

Submitted to Informatics Institute of Hacettepe University
as a Partial Fulfillment to the Requirements
for the Award of the Degree of Master of Science
in Computer Animation and Game Technologies

July 2021

ABSTRACT

AN INTELLIGENT LAYOUT ALGORITHM FOR VARIABLE SCREEN RESOLUTIONS

Barış Çelik
Master of Science, Computer Animation and Game Technologies
Supervisor: Asst. Prof. Dr. Burkay Genç
June 2021, 87 pages

In this thesis, we present an algorithm that will enable a relationally defined user interface to adapt to variable screen resolutions in real time with a smart and dynamic approach. User interfaces are design and software products that enable interaction between computers and users. Both designers and software have their duties in revealing these interfaces. In today's world, with the use of different devices and many different screen resolutions supported by these devices, it is expected to work successfully on each screen resolution in its developed applications. Unfortunately, designers and developers are developing different interfaces for all different resolutions, and maintenance and updating of these interfaces both in the initial development and in the ongoing process brings serious burden. Although different approaches have been proposed in the literature for the solution of this problem, these approaches are non-real-time approaches to reduce the effort spent in the design phase. In this study, we present a data structure that will take this load from the designers and developers, to be defined once and during the application's development, with a simple relational model, and a real-time approach that transforms this data structure into the most suitable interface to the given screen resolution. Preparation of the data structure we use is extremely easy, and it can be prepared in a shorter time than a single interface design. The creation of the interface is under a second during runtime, even during real-time resizing operations, the interface does not get stuck or stutter. In addition, we used the aesthetic scoring method to choose the best placement and thanks to this method, we selected the interface that appeals to the eye which increased the user experience.

Keywords: variable resolution, smart placement, user interface, responsive design, aesthetics

ÖZET

DEĞİŞKEN EKRAN ÇÖZÜNÜRLÜKLERİ İÇİN AKILLI BİR YERLEŞTİRME ALGORİTMASI

BARIŞ ÇELİK

Yüksek Lisans, Bilgisayar Animasyonu ve Oyun Teknolojileri

Danışman: Dr. Öğr. Üyesi Burkay GENÇ

Haziran 2021, 87 sayfa

Bu çalışmada ilişkisel olarak tanımlanan bir kullanıcı arayüzünün akıllı ve dinamik bir yaklaşımla, gerçek zamanlı olarak değişken ekran çözünürlüklerine en iyi şekilde adapte olmasını sağlayacak bir algoritma sunuyoruz. Kullanıcı arayüzleri bilgisayarlar ve kullanıcılar arasındaki etkileşimi sağlayan yazılım ürünleridir. Bu arayüzlerin ortaya çıkartılmasında tasarımcılar ile yazılımcılara düşen görevler vardır. Günümüzde değişik cihazların desteklediği çok farklı ekran çözünürlüklerinin kullanımda olmasıyla birlikte geliştirilen uygulamaların da her ekran çözünürlüğünde başarıyla çalışması beklenmektedir. Fakat, mevcut durumda tüm farklı çözünürlükler için tasarımcılar ve yazılımcılar farklı arayüzler geliştirmekte ve bu arayüzlerin hem ilk geliştirimi hem de devam eden süreçte bakım ve güncellenmesi ciddi bir külfet getirmektedir. Bu problemin çözümü adına literatürde farklı yaklaşımlar önerilmiş olmakla birlikte, bu yaklaşımlar tasarım aşamasında harcanan eforu azaltmaya yönelik, gerçek zamanlı çalışmayan yaklaşımlardır. Bu çalışmada biz tasarımcı ve yazılımcıların üzerinden bu yükü alacak, uygulamanın geliştirilmesi esnasında bir defa ve basit bir ilişkisel modelle tanımlanacak bir veri yapısını ve bu yapıyı gerçek zamanda işleyerek verilen ekran çözünürlüğüne en uygun arayüze dönüştüren gerçek zamanlı bir yaklaşım sunuyoruz. Kullandığımız veri yapısının hazırlanması son derece kolay olduğu gibi, tek bir arayüz tasarımından daha kısa bir zamanda hazırlanabilmektedir.

Uygulamanın çalışması esnasında arayüzün oluşturulması da saniyenin altında gerçekleşmekte, gerçek zamanlı yeniden boyutlandırma işlemleri esnasında dahi arayüzde gecikme yaşanmamaktadır. Bunun yanında en iyi yerleştirmeyi seçmek için de estetik puanlama metodu kullandık ve bu metod sayesinde göze en hitap eden arayüzü seçip kullanıcı tecrübesini arttıran etkenler kattık.

Anahtar Kelimeler: değişken çözünürlük, akıllı yerleştirme, kullanıcı arayüzü, duyarlı tasarım, estetik

ACKNOWLEDGEMENTS

I would like to express my endless gratitude to my advisor Asst. Prof. Dr. Burkay Genç, who supported me at every stage of my work and guided me with his knowledge and experience,

I would like to heartily thank my wife Bengisu Sarıbaz Çelik who was always by my side and supported me, and to my best friend Buğra Alp Çevikgibi for his moral support in this difficult journey.

I would like to also show my sincere appreciation to my thesis committee members; Prof. Dr. Haşmet Gürçay and Asst. Prof. Dr. Fatih Sağlam for taking their time to review and providing insightful comments.

Finally, I would like to thank my family, my mother Semra Çelik and my father Ali Cahit Çelik for their unconditional love and support.

CONTENTS

FIGURES

TABLES

ABBREVIATIONS

UI	User interface
UX	User Experience
CSS	Cascading Style Sheet

1. THE DEFINITION OF THE PROBLEM

1.1. Overview

Softwares that are produced today is designed to work on many different devices and platforms. However, in order to support all devices and platforms, a separate interface design must be designed for each. There are hundreds of phone models with different screen sizes and resolutions. Existing software libraries offer automatic placement algorithms to facilitate this task. However, these algorithms are quite simple and most of the work is done by designers and developers. The algorithm we developed will minimize the amount of work that will fall on software developers and designers. Roudaki[1] summarized various approaches to transfer the web experience to mobile devices but these are specific approaches that requires most of the work done by the designer and developers, which we aim not to.

The method we developed takes minimal information from the developer. This information is about the minimum and maximum size of each component and which components are "relevant". Being relevant is not geometric but abstract information and does not require the developer to choose a specific placement algorithm at this point. Later, when the interface is drawn on the screen, using the information received during software development, the width, height and position of all components are calculated using a smart algorithm and the interface is created automatically. To do this, the algorithm finds a good solution by pruning the tree of all possible placements. Since this process is done in less than a second, full optimization is achieved in relatively small interfaces and very close to the best in very complex interfaces. Thus, developers do not have to constantly rearrange their pages for different resolutions. It should be noted that, our goal is to place all the elements on the screen in a way that appeals to the eye, rather than hiding the elements that do not fit on the screen like most smart layout algorithms or responsive designs in the market.

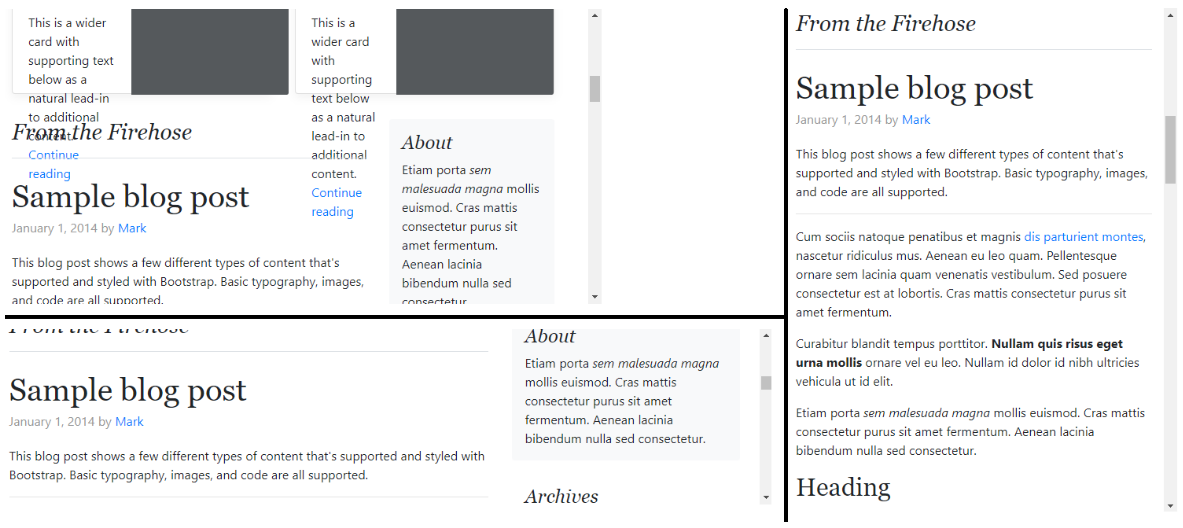
1.2. Motivation

Our main motivation to study automatic layout placement was the amount of work put into it manually. Generally, companies with internet sites develop this site to a platform with the most user base (phone, tablet, browser etc.). Then develop it for the other platforms with less care as it sees these other platforms as supporting platforms of the product. Which results with the company altering and redesigning whole UI according to User Experience on that device and redo all the work for all the other platforms and devices. If we give an example of a mobile game company; this company, which creates the most user base on the phone and makes games that appeal to this base for months, will need to redesign all the interfaces in the game when the tablets become popular and they want to bring tablet support at the intense demand of the users. The problem continues as these redesigns need maintenance as development continues and new features are introduced to the users. Suddenly one project becomes multiple projects and costs are increased substantially. There should be a way to design this layout once and port it to any platform - even to any resolution.

Of course the biggest problem with this idea is the aesthetic concerns which drove these companies to spend the time and effort to make these adoptions to different platforms beautifully. However, the question ”What if there is a way to create a relation between all the page elements and automatically create a layout aesthetically which bases aesthetics on mathematical formulas?” drove us to make some research on this topic.

In the Java programming language, the placement operations are performed by the LayoutManagers, but none of the LayoutManagers change the relative positions of the components according to the changing screen resolutions, they only change their size. Similarly, in the Android mobile operating system, nesting algorithms such as LinearLayout and RelativeLayout are used, but these are only concerned with the size of the components. In web design, things are even worse. Changing the interface according to variable screen resolutions is an application we see seldomly on web pages. Generally, web pages are made by assuming a certain width, in screen sizes above this width, most of the page remains blank, and on smaller screens, the content does not fit on the screen. As the use of mobile devices in daily life increases, companies have started to create additional designs that will alternatively cover mobile device sizes. However, these designs are quite costly and it has been very difficult to cover the new scaling that occurs as technology progresses.

In time, alternative solutions to this problem have been tried to be found. The most used of these is the framework developed by Twitter called Bootstrap. Bootstrap tries to change the interface in different screen resolutions with the logic of ”Responsive Design” defined. However, Bootstrap is not a smart system. It changes the interface according to some conditions determined during the design phase. An example is ”Reduce the width of the Y component at a resolution below X”. With Bootstrap, designs that are thought to be pleasing to the eye at every screen width and can be adapted to every scaling have started to be produced. However, these systems, which automatically resize themselves at different resolutions, bring glaring errors as the resolution difference gets larger as can be seen in Figure 1.1. Top left image has overflowing content, bottom left page has most of its’ content not present in the page and for the right page which looks the best in all three; is just all page elements vertically placed in order.


This is a wider card with supporting text below as a natural lead-in to additional content. Continue reading

From the Firehose

This is a wider card with supporting text below as a natural lead-in to additional content. Continue reading

About

Etiam porta *sem malesuada magna* mollis euismod. Cras mattis consectetur purus sit amet fermentum. Aenean lacinia bibendum nulla sed consectetur

Sample blog post

January 1, 2014 by Mark

This blog post shows a few different types of content that's supported and styled with Bootstrap. Basic typography, images, and code are all supported.

From the Firehose

Sample blog post

January 1, 2014 by Mark

This blog post shows a few different types of content that's supported and styled with Bootstrap. Basic typography, images, and code are all supported.

About

Etiam porta *sem malesuada magna* mollis euismod. Cras mattis consectetur purus sit amet fermentum. Aenean lacinia bibendum nulla sed consectetur.

Archives

From the Firehose

Sample blog post

January 1, 2014 by Mark

This blog post shows a few different types of content that's supported and styled with Bootstrap. Basic typography, images, and code are all supported.

Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum. Sed posuere consectetur est at lobortis. Cras mattis consectetur purus sit amet fermentum.

Curabitur blandit tempus porttitor. **Nullam quis risus eget urna mollis** ornare vel eu leo. Nullam id dolor id nibh ultricies vehicula ut id elit.

Etiam porta *sem malesuada magna* mollis euismod. Cras mattis consectetur purus sit amet fermentum. Aenean lacinia bibendum nulla sed consectetur.

Heading

FIGURE 1.1. Some examples where Bootstrap can’t handle non feasible resolutions which are taken from Bootstrap’s example pages.

In the development process, the prerequisites are determined by the developer and tested repeatedly at different resolutions. Despite all these efforts, it is quite possible to overlook possible errors and defects. The algorithm we propose is a smart algorithm that will decide and apply the best interface layout during the execution of the application without creating such conditions beforehand.

1.3. Research Questions

The aim of this thesis can be summarized as finding answers to these questions:

- What is an efficient way to store the geometric and topological constraints of interface elements?
- Can we compute all feasible layouts based on the constraints defined in real time?
- How can we choose the most aesthetically pleasing layout among all feasible layouts?

1.4. Organization of the Thesis

Structure of the thesis can be outlined as follows:

In **Chapter 2**, we talk about the previous studies on this topic and how they handled the problem. Some of the studies tried to mathematically formulate aesthetics which is essentially what we are aiming to achieve in our layouts; to produce an aesthetically pleasing layout for all resolutions.

In **Chapter 3**, the topological relation tree that we use to process interfaces is explained, with what we need from the developers and how we process it. Here we give some examples and detailed explanations on how the rules apply. After we discuss how it is done, the performance problem and the solution that makes this tree viable in real time executions, is also mentioned.

After explaining the underlying structures, in **Chapter 4**, we move on to the algorithm itself. Here we explain both how we create the tree and how we layout the components.

In **Chapter 5**, the aesthetic concern is added to the problem. We handpicked some aesthetic features from the literature and fine-tuned those mathematically formulated aesthetic parameters which is then used in calculation of aesthetic features of the layout.

In **Chapter 6**, we analyze some experiments and their results. In this analysis we inspect the results in both aesthetical aspects and performance aspects.

In **Chapter 7**, we conclude our work and talk about the potentials and downfalls of this project.

2. LITERATURE REVIEW

Smart environments require user interfaces to dynamically adapt to usage contexts, which are often unpredictable at the time of design. To solve this problem, it has been suggested to use real-time interface models[2]. Studies that find optimal results by manipulating the interface with correct modeling are available in the literature. Gajos and Weld[3] proposed a structure similar to the one we have proposed in their study, and using constraint solvers on their structure, they ensured the automatic development of an interface in accordance with the constraints received from the designer. Using a more specific partition method, the interface elements are added into a topological structure as in our study, and a tree is formed by using the data type in this structure. While the leaf nodes in the tree represent the interface elements that appear on the page, the branch nodes represent the relationships. In this study, constraint solvers were used differently than what we suggested, but the focus was not on finding alternative placements. Roscher[4], on the other hand, establishes a topological structure, but as the closest method to the structure we have built, it also includes position, size and horizontal or vertical values in each unit in its structure. The work done in the study is generally expressed in a verbal language and there is not much mentioning of theoretical details; this prevents the reproducibility of the study.

When we examine the literature, we observe that although we see similar studies, a real-time, smart placement algorithm has not been studied yet, as we have defined in this project. Similar studies in the literature are mainly based on the principle of making the interface design over the constraints, then solving these constraints on an optimization library and statically embedding the solution into the software. Therefore, studies mainly focused on creating interface design models and solving the emerging models efficiently, and provided algorithms[4][5] and tools[6][7] as outputs. For example, Jamil[8] made the Kaczmarz algorithm work with the inequality constraints encountered in interface design. Again, Jamil and Noreen[9] investigated how constraint solvers can be used in interface design and developed a prioritized grouping constraints algorithm to improve this method[10]. This algorithm is an algorithm that tries to find a constraint system without contradiction by repeatedly adding or removing constraints from the constraints system. Borning[11] similarly worked on dual simplex method-based solvers that can efficiently solve interface constraint models for news sites. Jacobs[12] defined grid-based templates and matched the content with these templates.

Nielsen[13], on the other hand, made studies on user experience in his research, and in these studies, he tries to answer the question of how to design a highly usable interface by going through the elements on the page. In our study, it does not seem possible to refer to usability within the scope of this thesis, since it is not an option to see the interface elements on the screen or which interface component to be visualized, but rather an information that the designer determines and gives to the algorithm as input. In addition, it is possible to carry out automatic placement studies in which the usability based on our algorithms is optimized in the future.

Trying to gain a new perspective rather than proceeding through editor-based approaches, Jiang[14] developed a structure that supports both Grid Layout and Flow Layout by blending the standard layouts used with his work called ORC Layout, and this structure uses a solver called Satisfiability Modulo Theory, shortly SMT, to produce results. In the study, an editor was written to be able to edit the ORC Layout, and the design was made through this editor and constraint-independent approach we aimed in our study was abandoned.

Zeidler[15] worked on constraint analysis using the program named Auckland Layout Editor [16]. However, these approaches do not fully describe a smart system, they only offer methods that can be used at the design stage. In another study, Zeidler[17] made detailed descriptions of editing operations, developed an algorithm that ensures that page elements do not interfere with each other in the design phase, and developed a plug-in that can see the behavior of an element resized in the design phase. The main purpose of the developments is that the order is always preserved and it does not contain intertwined elements in any way. In order not to overwhelm the constraint resolver with more than one option, it aimed to produce a specific and unique solution. In 2017, Zeidler[18] tried to increase the power of solvers by developing the algorithm and by doing a parceling study between interface elements. However, at the end of these three studies, Zeidler is still able to operate with the constraints determined at the design stage, and this method is only a more technological approach to the processes normally performed. Our aim is to get only the basic information during the design phase and update the design in the light of this information during the study.

Similarly, template-based approaches ensure that pre-determined templates and contents are matched to each other, so that the mapped content is drawn in accordance with the predefined and designed template. For example, in order to present the same content in magazines, newspapers and tabloids, templates previously prepared for these environments are used. For example, Jacobs[12] defined grid-based templates and synchronized the content with these templates. This study actually parallels the method used by Bootstrap in the design of Web pages today.

All of these approaches are more relevant to Bootstrap and CSS logic and are based on the relative fixing of the interface during design. In other words, the designer decides in which order and in which direction (horizontal-vertical-grid) the components in the interface will be placed before the program runs. It creates a constraint model using this information and solves it in a constraint solver. It also reflects the solution to the application interface. However, in the system we have developed, the designer will only indicate which components are related to each other, and will not interfere with their appearance on the screen in what order or direction. Such decisions are made by the algorithm itself while the application is running.

In addition, there are studies on evaluating a user interface in terms of aesthetics and usability and even scoring it numerically. One of the most recent comprehensive studies was conducted by Pajusalu[19]. Pajusalu examined different websites in this study and applied questionnaires to users through these websites to reveal which website was liked and why. Buanga and Mbenza[20], on the other hand, worked on the automatic evaluation of interface aesthetics in their thesis, examined how an interface can be evaluated from an objectivist perspective and how it will be graded on which criteria. However, these studies are not about the placement of the interface but about the interpretation of how it is placed, and for this reason are beyond the scope of this article. However, it was a step forward in the next stages of our research which lead us to the Gestalt principles[21] and Ngo's research[22][23] on measuring aesthetics.

Gestalt principles provide an overview of how people process what they see. These principles include Law of Proximity, Law of Similarity, Law of Symmetry, Law of Simplicity, Law of Closure, Law of Prägnanz, Law of Past Experience and are detailly explained in Chapter-3. Since these principles are quite applicable to interfaces, we have seen many papers trying to use them in their work. For example, Zeidler [15] examined these principles and claimed that the principles can be transferred to user interfaces. In the studies of Ngo[22][23] and Zen[24], we found the closest criteria to the aesthetic measuring we wanted. He was able to both incorporate Gestalt principles[21] into the decision-making mechanism and put these principles into mathematical formulas. Thus, after parsing real websites into UI elements and making calculations based on his formulas, he obtained efficient results and we intend to use those parameters in our work.

After Ngo, Zen[24] created a questionnaire called QUESTIM where users evaluated web pages on some aspects. Later on these results were used to compare the results of the aesthetic formulas with real world answers. This study brought further validation to the parameters Ngo generated, as the results of the questionnaire and formulas were mostly very close to each other.

3. CREATING AN AUTOMATIC LAYOUT

In this chapter, the topological relation between UI elements, the constraints of these elements and how they create a formula that we can use to draw a layout will be explained. First, we will explain the building block of this project, which is creating a layout with minimal input from developers. Then from this layout we will expand this idea to satisfy developers terms and conditions so that the result looks similar to what the developer thought of. After that we will discuss about the problem we encountered when we applied this solution to different resolutions.

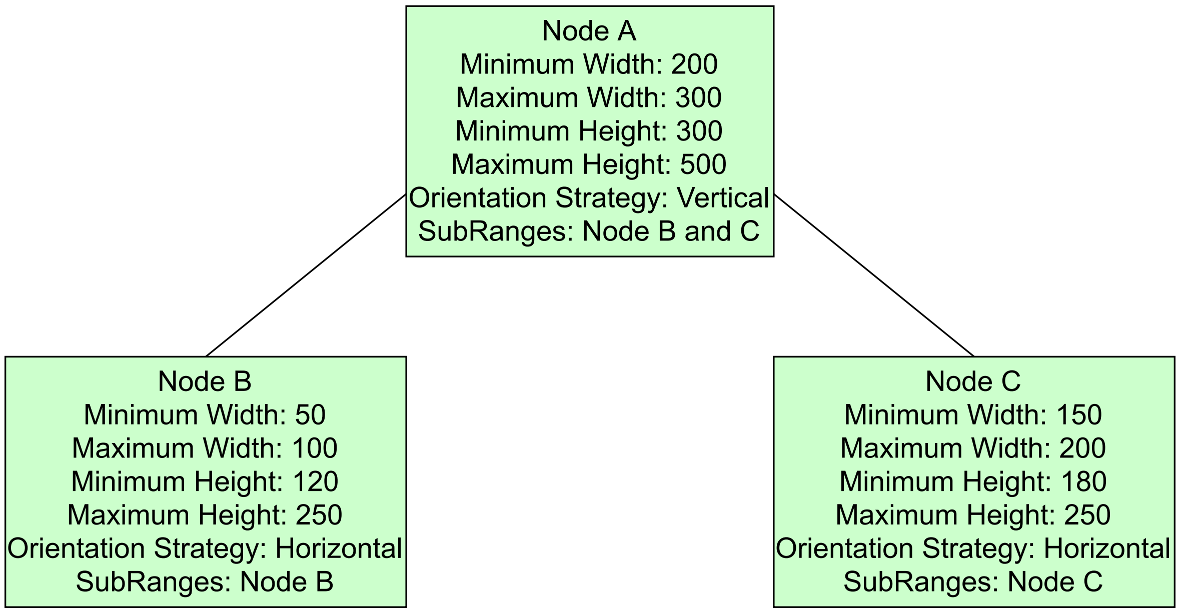


FIGURE 3.1. `WidthHeightRange` example

Before we start this process, we need to define what we need from the developers. Our topological relational tree nodes each hold a `WidthHeightRange` object in them which is the most basic element in a tree. In this smallest element of our tree, we hold minimum and maximum widths and heights as well as the orientation strategy which represents the node placement as vertical or horizontal, and lastly sub-ranges which holds the child nodes information that consist of other `WidthHeightRange` objects. We need this tree's `WidthHeightRange` objects to start processing and won't be needing any more information than this. In Figure 3.1., we can see an example of a small tree with `WidthHeightRange` nodes. Notice the node A is a container node and wraps node B and C, hence the minimum and maximum width and heights are the total amount of nodes B and C. Also notice the child nodes have the same orientation strategy needed in order to be applied correctly. Subranges of B and C are their own `WidthHeightRange` objects but in node A, subranges is a list of `WidthHeightRange` objects which consists of both node B and C's `WidthHeightRange` objects.

3.1. Creating a Layout With Minimal Information

One of the main goals of our solution is to minimize the information to be requested from the software developers during the design and coding stages and to ensure that the software developer makes as few decisions as possible. You have to make some very critical decisions when you want to create an interface in existing software languages. For example, if you want to develop a user interface with the Java Swing library, you must first decide which of the embedded algorithms Java[25] offers you to use. BorderLayout, for example, is one of them, and it allows you to divide the screen into five parts, top, bottom, right, left and center, and place a component in each piece.

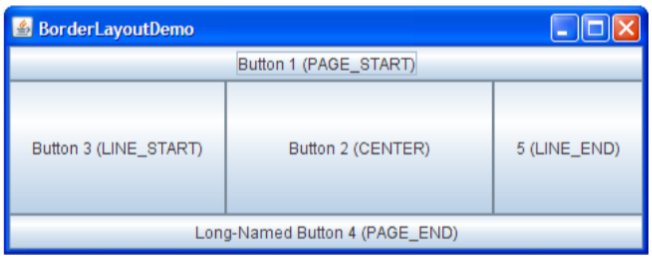


FIGURE 3.2. An example of a layout created by the BorderLayout algorithm

Usually when the app or screen size is changed, BorderLayout tolerates this change by making the center pane bigger or smaller. However, in addition, the software developer can assign preferred size information to each component. If deemed necessary, this information is used by the BorderLayout algorithm to calculate the size of the partitions.

Another example is the GridBagLayout algorithm. In this algorithm, the screen is divided into rows and columns and the components are placed in these rows and columns. The developer has to determine in advance which rows and columns each component will contain. In addition, many restrictions are given, allowing the screen image to be created as desired.

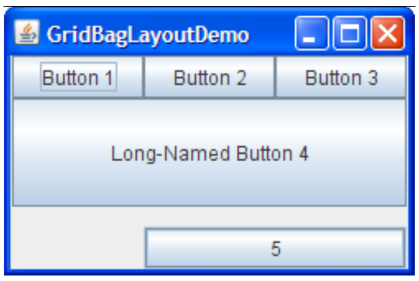


FIGURE 3.3. An example of a layout created by the GridBagLayout algorithm

3.2. Fundamental Structures

The algorithm we developed uses a tree structure to keep the relativity information of the components on the page. Each component that the programmer wants to see in the interface is included in this tree as a leaf, while the branches of the tree store the topological relationships between the components. Each leaf node holds its width and height ranges and assigned coordinates and assigned width and height values. While each branch symbolizes only a topological relationship, this topological relationship will evolve into a geometric relationship when the algorithm completes its analysis. As an example let's examine Figure 3.4.

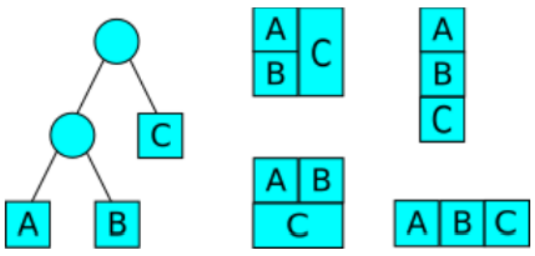


FIGURE 3.4. Topological relation tree on the left, four possible layouts on the right

In this picture, we see the topological relationships of 3 interface components stored in a tree. This tree tells us that components A and B are topologically related, in other words, they are linked, while component C is topologically related to the set of components A and B. These topological relations can evolve into 4 different geometric relations. These are shown in the same picture on the right. Since node C is “to the right” of the parent node of nodes A and B, the C component is positioned to the right or below the A and B components during insertion. This is information extracted from the creation of the tree but not directly provided by the software developer. Therefore, if this information is specifically indicated by the programmer that this information is invalid, the C component can be positioned to the left and above of the A and B components.

This results in 4 more geometric placements, and the total number of placements increases to 8. However, the general acceptance will be that the order in the tree will be preserved in the placement. In addition to these 4 possible placements, since each leaf in the tree, namely components A, B and C, also holds its minimum and maximum sizes, they can accept any of the unlimited number of values in the intervals they define as width and height. Even if we discretize the plane, the number of possible absolute insertions can easily rise to thousands or even millions. Besides, this is only a very rough calculation we made for a 3-component system. As we will talk about in a moment, as the diversity of local placements that can be preferred in each branch and the total number of components increases, this tree data structure will only grow linearly, but will contain an exponentially growing solution space. Continuing with the example in Figure 3.4., let’s assume that the following component size ranges are also given to the algorithm by the software developer:

As can be seen here, the programmer has to make 4 numerical inputs for each component. The first two of these show the horizontal dimension boundaries of that component, and the other two show the vertical dimension boundaries. That is, the horizontal size of the A component should be limited to between 100 and 200 units, while its vertical size should be between 50 and 150 units. This corresponds to about 10000 different rectangles in a discrete Euclidean space. Similarly, component B can be implemented with 7000 and component C with 7000 different rectangles. It will immediately be noticed that only some of these tens of

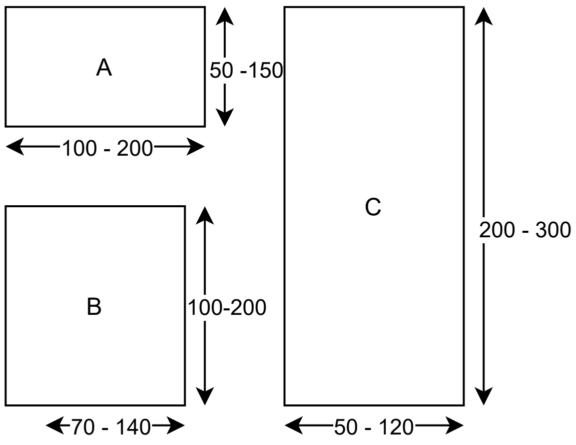


FIGURE 3.5. A numerated example based on Figure 3.4.

thousands of rectangles that we have defined for components A, B and C will satisfy visually pleasing placements. What the algorithm does is to choose the most ”correct” possibility among all these possibilities.

In addition, for the convenience of the developers, a certain range of values has been defined and a sizing system has been developed to use these values as width and height ranges. Size ranges starting from 20 to 50 pixels go up to sizes from 890 to 2000 pixels. It is designed to be playable as desired, as the correction and fine tuning of the gaps will vary from developer to developer. However, quite logical results have been obtained with the default values. And again, to make it easier for the developer, form elements that can be considered as ”classical” have been defined as default functions. For example, if the developer wants to create a name input field, both a label field and a text input field will be automatically added and created in a way that can be sorted side by side or one under the other. Of course, apart from these ”classic” elements, we also support custom elements written by the developers themselves. Ultimately, the important information of this element for our algorithm; the size range it will occupy and its relationships with other elements. Apart from this information, it works without any extra information and without distorting the given elements.

3.3. Finding Feasible Layouts

The data structure we have described above includes all possible placements without any criteria. However, there are some absolute criteria that need to be defined among the inputs. The most basic of these is the resolution of the screen where the placement will be made. Because if the width of the placement to be made is more than the screen width and the height is more than the screen height, the placement will not fit on the screen. There are two situations here. The first case is that the range values of at least one of the layouts overlap with the screen resolution. In this case, non-overlapping placements are eliminated and these placements are processed. Here, of course, it is extremely slow and cumbersome to use in a real-time application to calculate all the layouts one by one and see if they fit on the screen. Instead, we scan the tree from its leaves to its root to move the size range information from the leaves to the root, and since the root node symbolizes the entire layout, the range values that will appear at the root node are directly comparable to the screen resolution, helping us to choose.

If we go through our previous example, we have shown the existence of four possible layouts. Let's consider the first of these, that is, the placement where component A is located above component B and component C is located to the right of them. Also suppose, for aesthetic concerns, that components A and B must be of the same vertical size. Let's call X the smallest rectangle containing A and B, which is formed when components A and B are placed on top of each other. The X rectangle can be at least 100 and at most 140 horizontal units. Otherwise, the limit values of either the A or the B component would be violated. Similarly, the lower and upper vertical size limit values of the X rectangle are found as 150 and 350. New limit values can be found by adding the vertical limit values of the components placed on top of each other. In this case, the boundary values of the X rectangle are (100-140), (150-350). Since the components of X and C will be placed side by side, the boundary values of the smallest rectangle that includes X and C, let's call it the Y rectangle, are also found as (120-260), (200-300). Thus, we find that the limit values will be (120-260), (200-300) if the placement shown in the upper left in Figure 3.4. takes place. Let's say we want

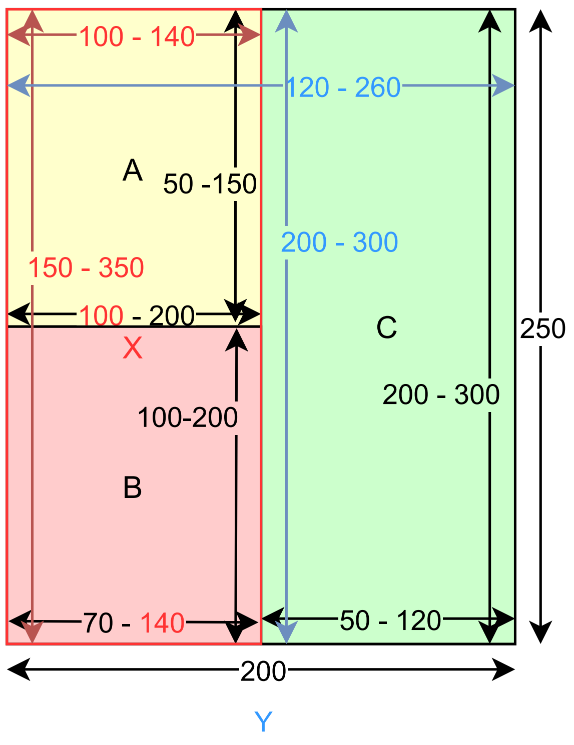


FIGURE 3.6. Feasible layout execution based on Figure 3.5.

to run this application on a screen with a resolution of 200x250. In this case, our algorithm finds that this placement can be used for the given screen resolution and lists the appropriate placements. However, if our screen resolution was 400x200, then our algorithm would not include this placement among possible placements, as we could not provide the horizontal size.

Note that each branch node in the tree offers two alternative placements: horizontal and vertical. For this case, the calculation should be made for 2^k placements in a tree with k branch nodes. This might not be too much of an issue on a 10-component display. However, as the number of components increases, this number will rapidly rise above the practical limits and it will not be possible for a real-time application to examine this many placements one by one. To solve this problem, our algorithm filters out the infeasible nodes before they reach the root node. For example, if the horizontal component of the screen resolution is given as 600 while the possible horizontal size limits of a branch node is (800-1200), then the calculation of this branch node is stopped because there is no point in moving the boundary range of that branch node up. In this way, many layouts are eliminated while still calculating branch nodes and the tree can be pruned quickly.

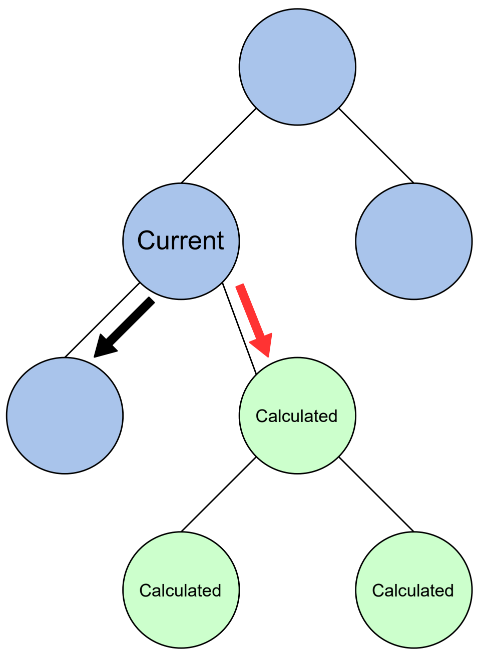


FIGURE 3.7. Green circles represent the calculated nodes which have their memoization fields filled and blue circles represent non calculated nodes

In addition, another process to speed up this process is the memoization mechanism. If we want to extract the appropriate tree structure again for the same resolution, which is a situation we often encounter while roaming our entire tree, then we don't need to calculate the nodes that we processed again. Our recalculation will return to us as a performance decrease since none of our parameters have changed. In Figure 3.7., green nodes indicate that they have been calculated before in the recursion, so while calculating current node, instead of calculating the green child node the memoization values return instantly and calculation continues with the blue child node. If we go through the previous example, when we want to calculate the possible height and width of the elements in the Y rectangle, we can see that we have already calculated the width and height of the X rectangle and this calculation will remain constant since the resolution or tree does not change. For this reason, we do not need to make a recalculation. But if we were doing this calculation for another resolution, the information kept in this record would have to be cleared.

In the second case we mentioned at the beginning, none of the calculated placements fit on the screen. In this case, we have two possible action plans. The first is to inform the user that there is no viable layout placements, which will not be the preferred way in most cases. In the second, the placement closest to the target resolution can be proposed and this placement supported by scroll bars of the interface. However introducing the second option is not in our thesis' scope and we only deal with the possible feasible layouts.

The algorithm also ranks the layout options corresponding to the screen resolution by assigning absolute values to the component sizes. The challenge is that each installation actually has infinite solutions in continuous space. Even with the discretization method, hundreds of solutions have to be overhauled one by one. Therefore, it seems very difficult to use brute force algorithms. Now, we describe a smart heuristic to deal with this problem.

By calculating the horizontal and vertical boundaries of all possible placements, using these boundaries and scanning the tree from top to bottom, the horizontal and vertical boundaries of all nodes are constrained between these values. In other words, all components must comply with the limit value set from above.

For example, suppose that components A and B are placed horizontally, and the horizontal boundary values are given as (50-150) for A and (100-200) for B. Therefore, the horizontal limit value of the node formed by the combination of A and B becomes (150-350). However, let's assume that it is desired to fit this node into a rectangle with a horizontal size of 220. In this case, it is not possible for A to have a horizontal magnitude of 150. Because the corresponding horizontal value of B will be 70, which will be outside the boundaries of B. Therefore, the horizontal boundaries of both A and B are trimmed to find (50-120) for A and (100-170) for B. Among these new values, there is a value that B can take for every value that A will receive.

At this stage, the tree is scanned from top to bottom and screen resolution constraints are applied to all branches and leaves. In other words, the specified ranges are reduced to a single value to exactly reflect the horizontal and vertical size of the screen. This reduction can be in two ways: If we want to fix the horizontal values of a cluster whose components are vertically placed, we equal the horizontal size of each component of the cluster to the horizontal size of the cluster. If we want to fix the horizontal size of a horizontally oriented cluster, then we distribute the horizontal size of the cluster among the components. Making this distribution process fast and aesthetically is one of the most critical functions of the algorithm. For this, we could use the simplex-like optimization routines used by Borning, as well as produce faster solutions with the help of greedy and heuristic methods, and we preferred greedy and intuitive methods in our algorithm and created a special control mechanism.

3.4. Considering Aesthetics

As described above, the topological relationships used as input may correspond to a large number of possible layouts. To determine which of these layouts will be selected and reflected to the interface, these layouts must be comparable to each other. This is only possible if we can assign a numeric value to each layout. The biggest problem here is that this assignment will be based on aesthetic values, and that the aesthetic values are relative. Therefore, an algorithm is developed to provide aesthetic evaluation of application interfaces based on the existing literature.

At this point, we have a list of feasible layouts that are able to provide (or closest to) the full screen resolution and Smart Layout algorithm will assign an aesthetic score to each implementation of these layouts. It should be noted that this scoring algorithm assumes that the position and size of each component in the interface is fully known. However, the information we have is the minimum and maximum of width and height values of each component. We obtain the required information by distributing the given resolution to the tree which is explained in Chapter-4.

When making the aesthetics decisions, Zen[24] has identified several methods and tried to remain as loyal to these methods as possible. These methods consist of; to consider less criteria and to focus on those criteria, to make use of real and used interfaces rather than artificial interfaces in creating aesthetic scoring system, and finally to make an annotated link between subjectivities and examine this linkage. As for Smart Layout algorithm, while making these decisions, we will also benefit from Gestalt's principles in the literature. In the field of aesthetic studies, Zeidler[15] examined Gestalt principles and argued that these principles can be transferred to user interfaces. In particular, a few of Gestalt's principles come to the fore that Smart Layout will benefit from:

- Law of Proximity: According to this principle, people perceive various objects while forming close groups.
- Law of Similarity: According to this principle, if the objects are similar, these various objects are grouped perceptually. For example, buttons of the same size are combined in the mind as if they were in the same group.
- Law of Symmetry: According to this principle, the mind perceives objects symmetrically and by shaping them around a center point. It is perceptually pleasing to divide objects symmetrically by an equal number.
- Law of Simplicity: Instead of perceiving a complex shape, turning to simple shapes is the general habit of our brain, and instead of analyzing more than one intertwined object, people choose to understand and define a few simple objects.
- Law of Closure: According to this principle, if the majority of an object is drawn but some parts are left incomplete, our brain automatically completes them and we detect what the object is.
- Law of Prägnanz: If the parts of an object form a regular, simple and ordered pattern, they are perceived as forming a group. Prägnanz is a German word that directly means succinctness, meaning clarity and regularity.

- Law of Past Experience: Under certain circumstances, elements tend to be perceived according to an observer's past experience.[26]

In the light of this information, Smart Layout establishes a system with a strong aesthetic aspect by incorporating Gestalt principles into it's decision making mechanism. The exact parametric values of the aesthetic scoring algorithm is explained in detail in Chapter 5.

4. THE ALGORITHM BEHIND IT

In this chapter, we first explain how developers can create a *topological relation tree* and give constraints to it. Later, we analize how the algorithm uses this topological relation tree and finds suitable layouts which then, distributes the resolution to those layouts so that the program outputs all the possible placements it can fit in the given resolution. This chapter is solely focused on the processing and distributing algorithms and aesthetic concerns are not in the scope of this chapter.

Here we briefly give an outline of our algorithm:

1. A topological relationship tree is created.
2. The width and height value ranges that each node can take are calculated with a recursive approach.
3. According to the resolution of the application, the width and height ranges found are fixed to a value which gives us feasible layouts.
4. All feasible layouts are subjected to aesthetic scoring in order to choose the best one among these layouts.
5. The layout with the highest aesthetic score is selected and drawn as the interface.

4.1. Creating Topological Relation Tree

For the sake of simplicity Java 8 language is used to write this experimental project. To create the topological tree we only need to define some Java GUI components. Since we extend those components, while the default GUI elements are still available, they are additionally linked to a topological relational tree. Leaf nodes of this tree is our GUI elements and we define them as we define native Java GUI elements, but when we are creating branch nodes or root node, no additional GUI elements gets defined. These branch nodes only carry leaf nodes and do not need extra information. It will get all the information it needs from the leaf nodes underneath while calculating possible layouts. Defining relations is fairly simple. After creating leaf nodes, the nodes given when creating branch nodes are linked from left to right. As explained in the previous section, the first node within the branch nodes and the second node can be related left to right or top to bottom. Nodes within a branch node can be both branch nodes and leaf nodes. With these rules, all nodes are connected to each other in some way and a tree structure is formed without establishing a circular connection, and the node that is not a parent of all these branch nodes is called a root node. Our data structure progresses through this root node and all information is accessible from the root node.

4.1.1. Contents of the Tree Nodes

Leaf nodes keep the `WidthHeightRange`, horizontal or vertical placement information, assigned coordinates, and assigned width and height information. `WidthHeightRange` is the list which contains all the possible width and height ranges for different layouts. This list actually holds the width and height ranges of each of the nodes under it. If it does not have a node below it, this node is already a leaf node and holds the width and height ranges assigned by the developer.

Branch nodes, on the other hand, hold the child nodes along with the information in the leaf nodes, as well as the pre-calculated nodes used for memoization. The recursive Algorithm-1, GetRanges, runs in branch nodes which calculates `WidthHeightRange` of node's children and thus calculates its' own `WidthHeightRange`.

4.2. Parsing The Tree

After mentioning the creation of the tree and the types of data it holds, we will now examine how to use this tree structure to find all possible placements and how the resolution given to these placements is distributed.

4.2.1. Finding the Range of Width and Height

Algorithm-1 finds possible layouts of a node based on given resolution values. When this algorithm is run for the root node, all possible layout arrangements of the whole tree is found. Flow of the algorithm is as follows; if it is running for the first time, that is, if the memoization is not filled in, the child nodes are navigated to the lowest elements. When looking at these nodes, calculations are made for both horizontal and vertical probabilities. If the current node is a leaf node, the `GetRanges` method returns us the `WidthHeightRange` of the leaf node which was given by the developer. However, if the current node is a branch node, new ranges are determined according to the minimum and maximum values of the width and height that the nodes below can take. After calculating these ranges, if the new maximum values we calculated for this node remain above the minimum values, this indicates that it is a viable range and it appears as a usable option in the next iteration. This option is added to the list as horizontal placement if considered for horizontal positioning, and vertical placement if considered for vertical positioning. Whenever the calculation of a node is finished, the memoization field is filled with the possible placements while returning to recursion so the upper branches do not have to recalculate this information. When the parent branch nodes go up one level in recursion, we can now have both leaf and branch nodes, but there will be no change in our calculations because still only the given width and height values are evaluated.

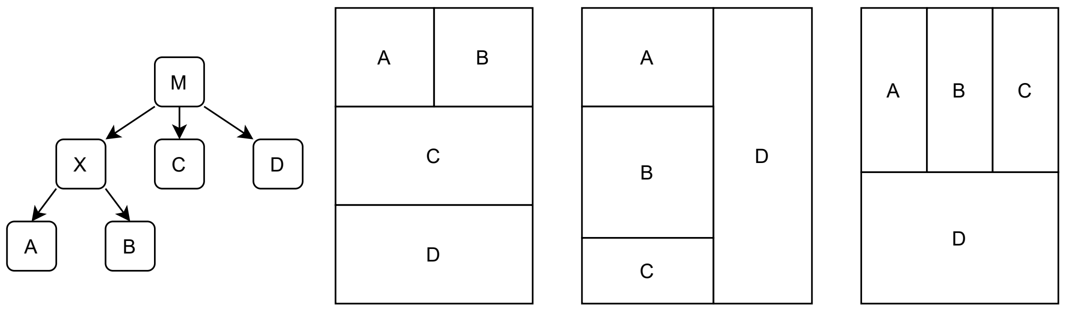


FIGURE 4.1. A sample tree and 3 possible layouts found by GetRanges

Here in Figure 4.1. we have an example tree and three possible layouts. In order to algorithm to accept these layouts as feasible, they need to fulfill the constraints of both resolution and each components. For example if we have a C component which can take 100 to 200 pixels of width and a resolution of (100, 300), second and third layout in Figure 4.1. will be infeasible and will be pruned by the GetRanges algorithm since component C can take a width of minimum 100 pixels which can't be applied.

Algorithm 1: GETRANGES Algorithm for detecting possible layouts

Input: Data structure that holds the tree information of the page

Output: Tree's WidthHeightRange object representing the form

```
1  if $memo \neq \emptyset$ then return $memo$
2  $movingRanges \leftarrow \emptyset$ list of WidthHeighRange
3  $tempRanges \leftarrow \emptyset$ list of WidthHeighRange
   // Horizontal possibilities
4  foreach child in children do
5      $compareList \leftarrow child.$GETRANGES
6      if $movingRanges = \emptyset$ then
7          Add all elements in $compareList$ to $movingRanges$
8      else
9          foreach $whr$ in $movingRanges$ do
10             foreach $newWhr$ in $compareList$ do
11                 if $newWhr$, is horizontally placeable with $whr$ then
12                     Add $newWhr$ to $tempRanges$
13         $movingRanges \leftarrow \emptyset$
14         Add all elements in $tempRanges$ to $movingRanges$
   // Vertical possibilities
15 foreach child in children do
16     $compareList \leftarrow child.$GETRANGES
17     if $movingRanges = \emptyset$ then
18         Add all elements in $compareList$ to $movingRanges$
19     else
20         foreach $whr$ in $movingRanges$ do
21             foreach $newWhr$ in $compareList$ do
22                 if $newWhr$, is vertically placeable with $whr$ then
23                     Add $newWhr$ to $tempRanges$
24         $movingRanges \leftarrow \emptyset$
25         Add all elements in $tempRanges$ to $movingRanges$
26 $memo \leftarrow movingRanges$
27 return $movingRanges$
```

One point to note here is that while looking at possible placements for a node, if both horizontal and vertical placement is suitable, we will have two different placement options. However, these options remain only as options. All calculations up to the root node must be consistent and completed in order for them to gain the status of layout. If there is a resolution with 900 units wide and the node has to range from 1000 to 1200 with some of its' child nodes added up for the horizontal option, then it is not a viable option, hence not added to the possible placement option list. Even if this option is valid, it can be overruled by adding the next child node to the account. In this case, this option cannot take its place as a possible placement in the final layout list.

Thanks to memoization, we do not need to perform recalculation when roaming nodes in the same topological relational tree structure at the same resolution.

After finding the `WidthHeightRange` of the root node, we have a list of all possible placements for this topological relational tree. However, this list does not contain any aesthetic concerns; it only gives a list of possible distributions with given constraints and resolution.

4.3. Distributing Resolution Amongst a Layout

How we selected the most "neat" interface from the list of all feasible layouts will be explained in the next chapter. However, we found the answer to the question of how to distribute a selected interface to a given resolution with more than one strategy.

4.3.1. Maximum Value Distribution Strategy

One of the first strategies that came to our mind was to give the elements their maximum values and distribute them, but this solution is quite monotonous and was quickly put to rest as it skipped most conditions. The first point we arrived at in brainstorming was how we would distribute the remaining values after assigning values to them all. The remaining values could be distributed equally, weighted according to their size, distributed by trying to equalize the values. These distribution strategies gave us an idea. The only obstacle to implementing all of these was to make sure that the minimum values were distributed. This strategy is completely overridden when the assigned maximum values exceed the resolution. For this reason, we abandoned this strategy.

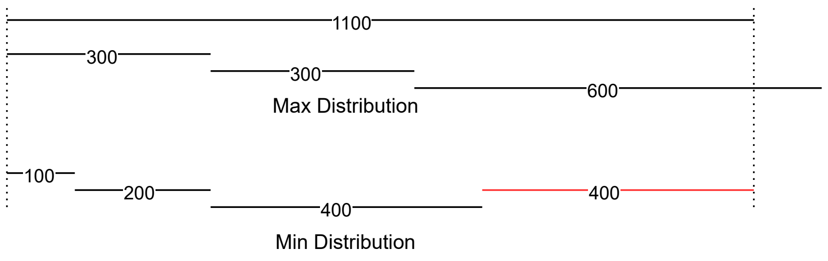


FIGURE 4.2. Minimum and Maximum Distribution Examples

To give an example to this strategy, assume we have 1100 units of width to distribute and A (100,300), B (200,300) and C (400,600) components' widths are given as seen in Figure 4.2. If we distribute the maximum values of these components, A will get 300, B will get 300 and C will get 600 which is a total of 1200 units and this distribution is not feasible for this resolution.

4.3.2. Minimum Value Distribution Strategy

Very similar to the previous strategy, we proceeded with this strategy by assigning minimum values. But this solution also started to create gaps throughout the layout. We scrapped this strategy too and tried to come up with a hybrid solution of these two strategies.

Assume we have the same example from maximum value distribution strategy. If we distribute the minimum values of these components, A will get 100, B will get 200 and C will get 400 which is a total of 700 units as seen in Figure 4.2. This distribution is also not feasible and leaves us with a 400 units wide gap in our layout.

4.3.3. Fair Distribution Strategy

First hybrid strategy was the fair strategy which distributed the minimum values of each element. Since we set minimum values, we do not have any cases that will cause overflows but in case of an underflow, remaining values are evenly distributed. Although this strategy did not produce bad results, roughness and imbalances in the interface were common. As a result we tried to draw a more weight-based strategy.

Algorithm 2: LAYOUT Fair Distribution Algorithm

```
Input: Tree information of the page
Output: Exact coordinates, width and height of each UI element
1  $feasible \leftarrow false$
2  $distribution \leftarrow \emptyset$ size is number of children
3  $remaining \leftarrow$ remaining value of horizontal or vertical amount
   // Distribute minimums to each child in node foreach $child$ in $children$ do
4      $distribution[current] \leftarrow$ minimum value of this child's distributed value
5      $remaining -= distribution[current]$
6  while $remaining > 0$ do
7      if $remaining/\sum remainingChild < minimum$ then
8          distribute $remaining/\sum remainingChild$
9      else
10         distribute $minimum$
11 $offset \leftarrow 0$
12 foreach $child$ in $children$ do
13     if $orientation = Horizontal$ then
14         if $\neg child.$LAYOUTFAIR($x+offset, y, distribution[current], height$) then
15             return $false$
16     else
17         if $\neg child.$LAYOUTFAIR($x, y+offset, width, distribution[current]$) then
18             return $false$
19     $offset += distribution[current]$
20 return $true$
```

Assume we have a width of 1100 units to distribute and widths of components A (100-153), B (300-600) and C (400-600) are given. First, the minimum values are distributed which are 100, 300 and 400 which totals 800 units. The remaining 300 can't be distributed

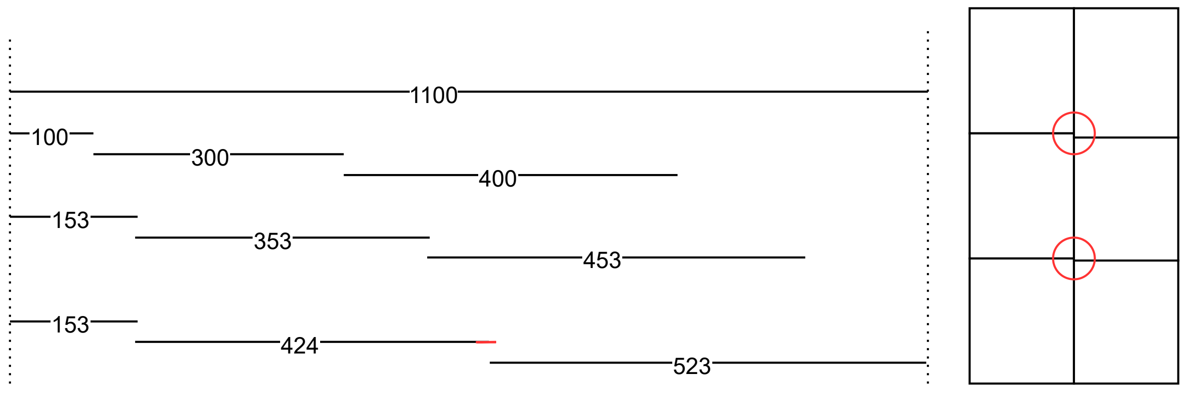


FIGURE 4.3. Fair Distribution Example

equally because component A's constaints prevents them. So the minimum value that makes a component reach its' max value which in this case is component A, is the distribution amount. A can take 53 units at most so 53 units are distributed equally which equates to 153, 353 and 453 with a total of 959. Now that A is out of distribution, we need to distribute 141 units to B and C, but the problem is we give B 71 units and C 70 units which gives a pixel off. This means that in each distribution, an imbalance will appear as much as the remaining unit of the partition. For example if we try to divide 7 to 4 components, last component will get the remainder, 3 units more than the other components. This look creates a displeasing and uneven layouts as the whole tree merges and the group next to this one is distributed evenly just like the one in Figure 4.3., which is not preferred.

4.3.4. Weighted Distribution Strategy

In the weight-based strategy, the values between the units to be distributed were proportioned and the total was deducted from the value to be distributed. The proportions can be chosen by the max values or min values of the components. This brings another level of detail as the use cases and results change depending on the situation. The most important point here is that the maximum values are respected and no overflow is created. Generally this approach gave us much more stable and aesthetically pleasing results.

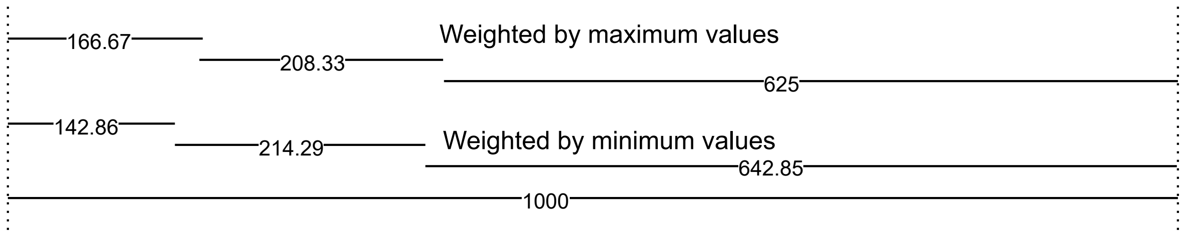


FIGURE 4.4. Weighted Distribution Example

For example, suppose there are three ranges in a branch node's width values, (100-200), (150-250), (450-750). When the algorithm wants to make a distribution of 1000 units, there are two options. We can pass maximum or minimum values of these width ranges, or even our own weight ratios. With that information, the algorithm calculates the weights of those given numbers according to the number of components it's going to distribute which in this case, is as 4, 5, 15 for maximum values given, and 2, 3, 9 for minimum values given. The algorithm then distributes these values as can be seen in Figure 4.4.

Algorithm 3: LAYOUT Weighted Distribution Algorithm

Input: Tree information of the page

Output: Exact coordinates, width and height of each UI element

```
1  $ratio \leftarrow$ totalOfChildren / (width or height)
2  $offset \leftarrow 0$
3  foreach child in children do
4      $value \leftarrow$ maxValueOfChild[i]
5      if $orientation = Horizontal$ then
6          if $\neg child.$LAYOUTFAIR$(x+offset, y, value, height)$ then
7              return $false$
8      else
9          if $\neg child.$LAYOUTFAIR$(x, y+offset, width, value)$ then
10             return $false$
11     $offset \mathrel{+}= value$
12 return $true$
```

4.3.5. Balanced Distribution Strategy

After the promising results gained from weighted distribution, we turned to balanced distribution by going back to equal distribution logic. As in the principle of equal distribution, we initially assigned the minimum values and then proceeded by adding the remaining values so that the totals were equal. We based this idea on Gestalt's Principle of Similarity[21]. Similar sizes tend to group in human mind and become more aesthetically pleasing. Of course, the primary purpose of the distribution algorithm is not aesthetics, but in the next step, when the aesthetic rules are applied, we should make the distribution as appealing to the eye as possible to ensure that the layout gets more aesthetic points. While respecting the maximum values, we tried to move all child nodes to similar dimensions by increasing incrementially between the elements that can be distributed.

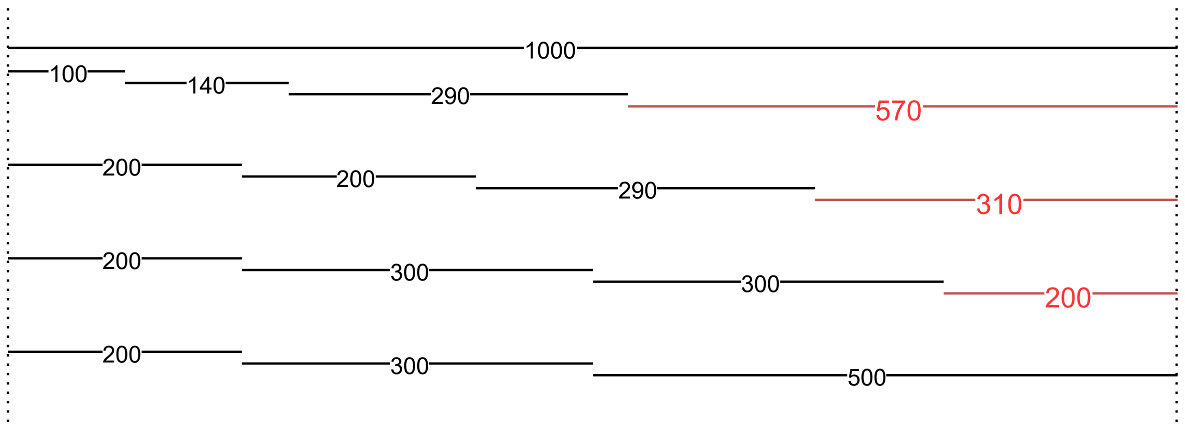


FIGURE 4.5. Balanced Distribution Example

Assume we have three ranges in a branch node's width values (100, 200), (140,300), (290,750) and the algorithm is trying to distribute 1000 units. As shown in the Figure 4.5., in the first step, the minimum values of all three ranges are entered, and thus the width values are 100, 140, 290. It then gradually increases, reaching the maximum value of the first range. The width values in the second case are 200, 200, 290. The distribution continues with the remaining two ranges, as the first range can no longer get any more value. When these two ranges are equalized, even though the distribution is 200, 300, 300 there are still 200 units that needs to be distributed, and since only the third range remains, the whole value goes to the third range and our final distribution becomes 200, 300, 500.

This distribution strategy paired with Gestalt's principles[21], gave the most pleasing results with no margin of error. Hence we decided to go on with this strategy only for the sake of simplicity since mixing strategies will hinder the performance yet again.

We chose to continue with this strategy, as we achieved the most stable results with the balanced distribution algorithm. After the tree is created and processed, and after the layout processing is completed with this strategy, we are now going to the stage of evaluating the aesthetic points of all feasible layouts and selecting the layout with the highest aesthetic score according to these points.

4.3.6. Comparing Strategies

We lastly made a comparison between fair distribution and balanced distribution, as other distributions often produce problematic results or do not adequately address all the distribution cases. The results of fair distribution in the same and different resolutions can be seen respectively in Figure 4.6. and Figure 4.7.

As seen in the Figure 4.6. interface, which is distributed at the same resolution without using aesthetic scoring, it seems that the two possible layouts have given very smooth results.

But when we change the resolution, we start to encounter problems like in Figure 4.7. Although it seems to have no difficulty in placing the elements properly when the space is narrowed and it seems to have achieved a phone or tablet-like image. But also, the dimensions of similar labels are not preserved. The combobox is given a range that can take up more space than other text fields, especially. We know that the combobox element will take the same size as the other text fields in a balanced distribution since the values are distributed one by one.

Algorithm 4: LAYOUT Balanced Distribution Algorithm

```
Input: Tree information of the page
Output: Exact coordinates, width and height of each UI element, capacity values of the
        current node
1  $feasible \leftarrow false$
2  $distribution \leftarrow \emptyset$ size is number of children
3  $remaining \leftarrow$ remaining value of horizontal or vertical amount
   // Distribute minimums to each child in node foreach child in
   children do
4      $distribution[current] \leftarrow$ minimum value of this child's distributed value
5      $remaining -= distribution[current]$
6      if $remaining \leq 0$ then
7          break;
8  while $remaining > 0$ do
9      if $distribution[current] < maxValues[current]$ then
10         Slot is full don't add to this node anymore
11     else
           // Increment the minimum node to balance all the
           nodes
12         $distribution[current] += 1$
13         $remaining -= 1$
14 $offset \leftarrow 0$
15 foreach child in children do
16     if $orientation = Horizontal$ then
17         if $\neg child.$LAYOUTBALANCED ($x+offset$, $y$, $distribution[current]$, $height$)
            then
18             return $false$
19     else
20         if $\neg child.$LAYOUTBALANCED ($x$, $y+offset$, $width$, $distribution[current]$)
            then
21             return $false$
22     $offset += distribution[current]$
23 return $true$
```

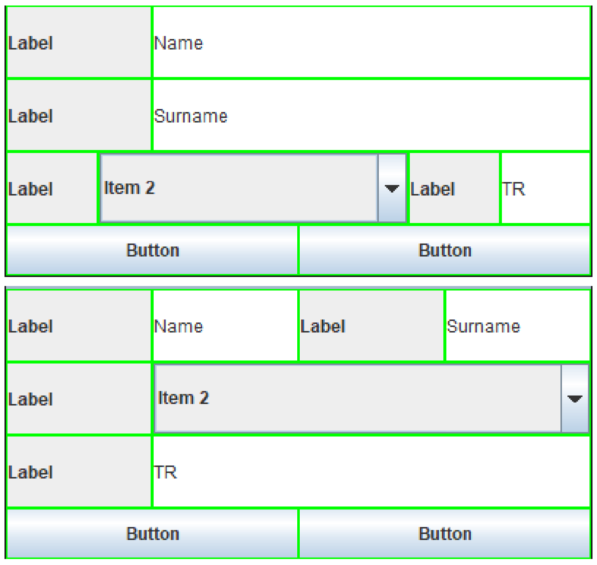


FIGURE 4.6. Fair Distribution, same resolution, different layouts

Just like in this example seen in Figure 4.8., while maintaining the label dimensions in the layout on the left, distributions that would disrupt the overall image were avoided. If the label dimension sizes can't be kept same like the others, then in order to preserve Gestalt principles[21], a perception such as a separate grouping has been created.

The protection of Gestalt principles is best understood when we look at distributions of different resolutions. While trying to fill only the constraints of the combobox with different resolutions in Figure 4.7., the *Name* and *Surname* fields were provided equal coverage, and the *Country* and the checkbox pair were also equalized, as they were distributed one by one in Figure 4.9. Whether we use a balanced distribution or a fair distribution, we have very workable layouts and we do not know at this stage which one to choose according to what. Since there are hundreds of layouts here, we need a decision-making mechanism and in this case, this mechanism is aesthetic scoring.

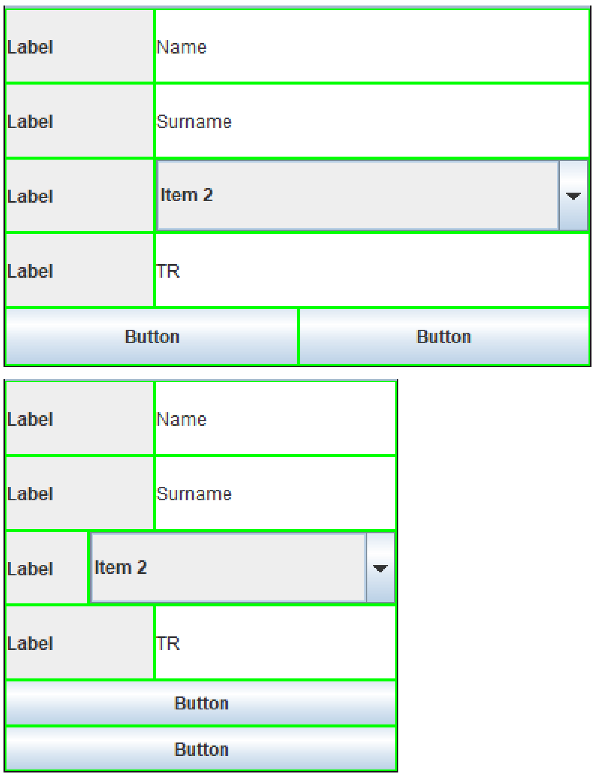


FIGURE 4.7. Fair Distribution, same tree, different resolutions

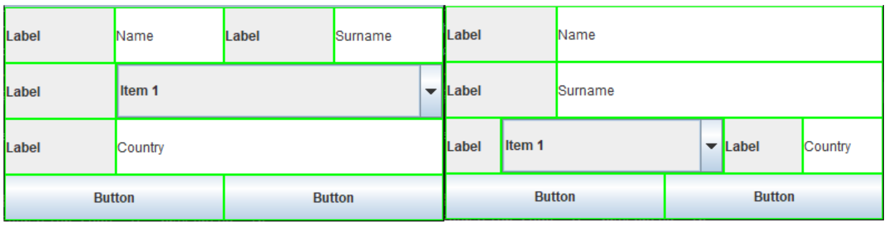


FIGURE 4.8. Balanced Distribution, same resolution, different layouts

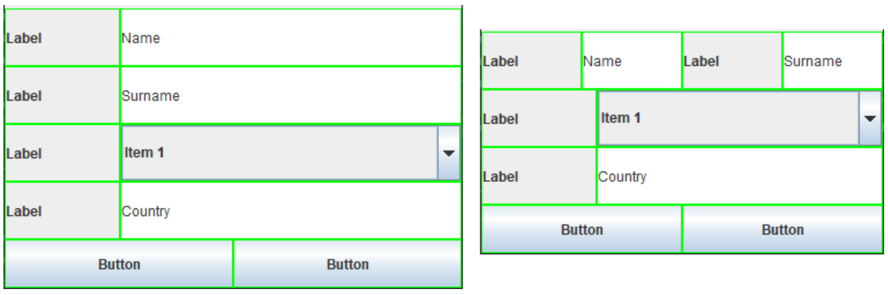


FIGURE 4.9. Balanced Distribution, same tree, different resolutions

5. AESTHETICS AS A DECISION MAKER

After finding all the feasible layouts, it's time to choose the most aesthetic result. After examining Gestalt principles[21] and Ngo's work[22][23], we realized that Ngo was trying to follow Gestalt principles as much as possible, and we found the mathematical formulas he derived by finding parameters compatible with these principles. We examined the aesthetic formulas that Ngo's study revealed, processed nine of them ourselves, and edited five of them and made them suitable for our work. We have assigned a weight to each aesthetic factor we use, and we have achieved an aesthetic score by the parameters and their respective weights. These parameters are; balance, equilibrium, symmetry, sequence, cohesion, unity, proportion, simplicity, regularity, economy, homogeneity, rhythm, order and complexity. Each parameter and weight coefficient can take a value between 0 and 1 so the result is always a positive decimal number. Equations we modified, are arranged in such a way as to derive their calculations since shapes and colors are not taken into account in our experiments. All of the aesthetic parameters we have explained below are taken from Ngo's work as a concept and these parameters have been redesigned by making them suitable for our study.

5.1. Balance

The first aesthetic parameter, balance, can be defined as the distribution of weight in a picture. Larger objects are heavier, smaller objects are lighter. Balance in screen design is achieved by giving equal weight to the left and right, top and bottom. The equation is given[22] as follows:

$$BM = 1 - \frac{|BM_{vertical}| + |BM_{horizontal}|}{2} \in [0, 1] \tag{1}$$

$BM_{horizontal}$ and $BM_{vertical}$ are horizontal and vertical balances that are explained in Equation 2 and 3.

$$BM_{vertical} = \frac{w_L - w_R}{max(|w_L|, |w_R|)} \tag{2}$$

$$BM_{horizontal} = \frac{w_T - w_B}{max(|w_T|, |w_B|)} \tag{3}$$

where

$$w_j = \sum_{n_j}^{i} d_{ij} \frac{\alpha_{ij}}{\alpha_{max}}, \quad j = L, R, T, B \tag{4}$$

$$a_{max} = max\left(a_{ij}, i = 1, 2, ..., n_j \quad j = L, R, T, B\right) \tag{5}$$

Letter j stands for the side component exists on which can take the values L for Left, R for Right, T for Top and B for Bottom. a_{ij} stands for the area of object i on side j; d_{ij} is the distance between center points of object and frame; n_j is the total number of objects on that side. The results of these formulas are then multiplied by the balance weight coefficient, and the result gives us the balance aesthetic score of this layout.

5.2. Equilibrium

Equilibrium in a layout is measured in how centered the overall layout is. The center of mass of overall elements and the frame's center of mass must align or be close together in order to achieve equilibrium in a layout. The equilibrium equation is as follows[22]:

$$EM = 1 - \frac{|EM_x| + |EM_y|}{2} \in [0, 1] \tag{6}$$

$$EM_x = \frac{2\sum_i^n a_i(x_i - x_c))}{nb_{frame}\sum_i^n a_i} \tag{7}$$

$$EM_y = \frac{2\sum_i^n a_i(y_i - y_c))}{nh_{frame}\sum_i^n a_i} \tag{8}$$

where (x, y) pairs are coordinates for object i and frame c; a_i is the area of object i; b_{frame} and h_{frame} are width and height values of the frame respectively; n is the number of objects in frame. Maximum values $|x_i - x_c|$ and $|y_i - y_c|$ can get as high as $b_{frame}/2$ and $h_{frame}/2$ respectively since otherwise these objects will not be in the screen.

EM_x represents the center of mass of the x coordinates of all objects while EM_y represents the same for the y coordinates. As the values approach zero, the equilibrium increases, as these values increases in the positive direction, the center of mass shifts upwards or right depending on looking at EM_x or EM_y value, and in negative direction, the center of mass shifts to the left or down. It should be noted that, since we don't use spaces in our study, we tried to implement it but the results were always 1 so we decided to weight this parameter as 0.

5.3. Symmetry

Symmetry can be achieved by mirroring images. Here we have three kinds of symmetry; Horizontal, Vertical and Radial[22]. Horizontal and Vertical symmetry is preserved by having equivalent elements in their respective axes while radial symmetry is preserved by having equivalent elements in two or more axes which in this case x and y axes create our symmetry parameters, allowing us to measure a diagonal symmetry.

$$SYM = 1 - \frac{|SYM_{vertical}| + |SYM_{horizontal}| + |SYM_{radial}|}{3} \in [0,1] \tag{9}$$

$$SYM_{vertical} = \frac{\begin{array}{l} \left|X'_{UL} - X'_{UR}\right| + \left|X'_{LL} - X'_{LR}\right| + \left|Y'_{UL} - Y'_{UR}\right| + \left|Y'_{LL} - Y'_{LR}\right| + \\ \left|H'_{UL} - H'_{UR}\right| + \left|H'_{LL} - H'_{LR}\right| + \left|B'_{UL} - B'_{UR}\right| + \left|B'_{LL} - B'_{LR}\right| + \\ \left|\Theta'_{UL} - \Theta'_{UR}\right| + \left|\Theta'_{LL} - \Theta'_{LR}\right| + \left|R'_{UL} - R'_{UR}\right| + \left|R'_{LL} - R'_{LR}\right| \end{array}}{12} \tag{10}$$

$$SYM_{horizontal} = \frac{\begin{array}{l} \left|X'_{UL} - X'_{LL}\right| + \left|X'_{UR} - X'_{LR}\right| + \left|Y'_{UL} - Y'_{LL}\right| + \left|Y'_{UR} - Y'_{LR}\right| + \\ \left|H'_{UL} - H'_{LL}\right| + \left|H'_{UR} - H'_{LR}\right| + \left|B'_{UL} - B'_{LL}\right| + \left|B'_{UR} - B'_{LR}\right| + \\ \left|\Theta'_{UL} - \Theta'_{LL}\right| + \left|\Theta'_{UR} - \Theta'_{LR}\right| + \left|R'_{UL} - R'_{LL}\right| + \left|R'_{UR} - R'_{LR}\right| \end{array}}{12} \tag{11}$$

$$SYM_{radial} = \frac{\begin{array}{l}\left|X'_{UL} - X'_{LR}\right| + \left|X'_{UR} - X'_{LL}\right| + \left|Y'_{UL} - Y'_{LR}\right| + \left|Y'_{UR} - Y'_{LL}\right| + \\ \left|H'_{UL} - H'_{LR}\right| + \left|H'_{UR} - H'_{LL}\right| + \left|B'_{UL} - B'_{LR}\right| + \left|B'_{UR} - B'_{LL}\right| + \\ \left|\Theta'_{UL} - \Theta'_{LR}\right| + \left|\Theta'_{UR} - \Theta'_{LL}\right| + \left|R'_{UL} - R'_{LR}\right| + \left|R'_{UR} - R'_{LL}\right|\end{array}}{12} \tag{12}$$

where X'_j, Y'_j, H'_j, B'_j, Θ'_j and R'_j are, respectively, the normalised values of X_j, Y_j, H_j, B_j, Θ_j and R_j

$$X_j = \sum_i^{n_j} \left|x_{ij} - x_c\right| \quad j = UL, UR, LL, LR \tag{13}$$

$$Y_j = \sum_i^{n_j} \left|y_{ij} - y_c\right| \quad j = UL, UR, LL, LR \tag{14}$$

$$H_j = \sum_i^{n_j} h_{ij} \quad j = UL, UR, LL, LR \tag{15}$$

$$B_j = \sum_i^{n_j} b_{ij} \quad j = UL, UR, LL, LR \tag{16}$$

$$\Theta_j = \sum_i^{n_j} \left|\frac{x_{ij} - x_c}{y_{ij} - y_c}\right| \quad j = UL, UR, LL, LR \tag{17}$$

$$R_j = \sum_i^{n_j} \sqrt{\left(x_{ij} - x_c\right)^2 + \left(y_{ij} - y_c\right)^2} \quad j = UL, UR, LL, LR \tag{18}$$

where j can get the quadrants upper-left (UL), upper-right (UR), lower-left (LL), lower-right (LR). (x, y) pairs are center coordinates for object i on quadrant j and frame c; b_{ij} and h_{ij} are the width and height of the object and finally n_j is the total number of objects on that quadrant.

5.4. Sequence

In the habit of reading, the eye has gotten used to moving from the top left to the bottom right of the page. According to the findings of psychologists, the eye moves from large objects to small objects, from irregular shapes to regular shapes. In the light of this information, Ngo claimed that the large objects in the upper left quadrant are the most prominent objects, and revealed the following equations[22].

$$SQM = 1 - \frac{\sum_{j=UL,UR,LL,LR} |q_j - \nu_j|}{8} \in [0,1] \tag{19}$$

$$\{q_{UL}, q_{UR}, q_{LL}, q_{LR}\} = \{4,3,2,1\} \tag{20}$$

$$\nu_j = \begin{cases} 4 & if\ w_j\ is\ the\ largest\ in\ w \\ 3 & if\ w_j\ is\ the\ 2nd\ largest\ in\ w \\ 2 & if\ w_j\ is\ the\ 3rd\ largest\ in\ w \\ 1 & if\ w_j\ is\ the\ smallest\ in\ w \end{cases} \quad j = UL, UR, LL, LR \tag{21}$$

$$w_j = q_j \sum_i^{n_j} \alpha_{ij} \quad j = UL, UR, LL, LR \tag{22}$$

$$w = \{w_{UL}, w_{UR}, w_{LL}, w_{LR}\} \tag{23}$$

where j can get the quadrants upper-left (UL), upper-right (UR), lower-left (LL), lower-right (LR). a_{ij} stands for the area of object i on quadrant j. Normally, in Ngo's study[22], Equation 19 also checks color and shape as well but we don't use these parameters since it is not the scope of our study. In addition, since the sequence value can easily reach 1, we reduced the coefficient to 0.2 in order not to affect the aesthetic scoring too much. The reason for this is that, due to the structure of the algorithm, it tries to distribute evenly across the quadrants, as it generally tries to obtain balanced layouts. If we used a different distribution algorithm; for example, fair distribution, then we could get much more variable results from this metric.

5.5. Cohesion

Ngo claims that[22], similar aspect ratios represent cohesion, where our aspect ratio is width and height. Measuring layout with screen and objects with layout gives us the cohesion values of the given layout.

$$CM = \frac{|CM_{fl}| + |CM_{lo}|}{2} \in [0,1] \tag{24}$$

CM_{fl} is a relative measure of the ratios of the layout and screen with

$$CM_{fl} = \begin{cases} t_{fl} & if\; t_{fl} \leq 1 \\ \frac{1}{t_{fl}} & Otherwise \end{cases} \tag{25}$$

and CM_{lo} is a relative measure of the ratios of the objects and layout with

$$CM_{lo} = \frac{\sum_i^n f_i}{n} \tag{26}$$

$$t_{fl} = \frac{\frac{h_{layout}}{b_{layout}}}{\frac{h_{frame}}{b_{frame}}} \tag{27}$$

$$f_i = \begin{cases} t_i & if\; t_i \leq 1 \\ \frac{1}{t_i} & Otherwise \end{cases} \tag{28}$$

$$t_i = \frac{\frac{h_i}{b_i}}{\frac{h_{layout}}{b_{layout}}} \tag{29}$$

where b_i and h_i are the width and height of the object i; b_{layout} and h_{layout}, and b_{frame} and h_{frame} are the width and height of the layout and the frame; n is the total number of objects in the frame.

5.6. Unity

Unity creates the perception of all the elements of the page belong together. Ngo describes it as a totality of elements that is visually all one piece[22]. To achieve unity, the form must contain similar sized elements.

$$UM_{form} = 1 - \frac{n_{size} - 1}{n} \in [0,1] \tag{30}$$

In Equation 30, we differed from Ngo's study since we don't use spacers in our study. The number we obtained by proportioning the number of distinct areas, n_{size}, to the total number of areas, n, shows us how similar the dimensions of the objects in interface is. We see that the smaller this number, the more similar sizes are used. When we normalize the number to 1, we get our aesthetic score.

5.7. Proportion

Although beauty and aesthetics vary from person to person, from culture to culture, there are some things that do not change over time. For example, Fibonacci's golden ratio is still valid today. Using this knowledge, Ngo claimed that[22] these ratios are aesthetically pleasing:

- Square (1:1)
- Square root of two (1:1.414)
- Golden rectangle (1:1.618)
- Square root of three (1:1.732)
- Double square (1:2)

$$PM = \frac{\left|PM_{object}\right| + \left|PM_{layout}\right|}{2} \in [0,1] \tag{31}$$

PM_{object} is the difference between the proportions of objects and the closest proportioned shapes which is formulated as

$$PM_{object} = \frac{1}{n}\sum_{i}^{n}\left(1 - \frac{min\left(\left|p_j - p_i\right|, j = sq, r2, gr, r3, ds\right)}{0.5}\right) \tag{32}$$

and PM_{layout} is the difference between the proportions of the layout and the closest proportional shape which is formulated as

$$PM_{layout} = 1 - \frac{min\left(\left|p_j - p_{layout}\right|, j = sq, r2, gr, r3, ds\right)}{0.5} \tag{33}$$

$$\{p_{sq}, p_{r2}, p_{gr}, p_{r3}, p_{ds}\} = \left\{\frac{1}{1}, \frac{1}{1.414}, \frac{1}{1.618}, \frac{1}{1.732}, \frac{1}{2}\right\} \tag{34}$$

$$p_i = \begin{cases} r_i & if\ r_i \leq 1 \\ \frac{1}{r_i} & Otherwise \end{cases} \tag{35}$$

$$p_{layout} = \begin{cases} r_{layout} & if\ r \leq 1 \\ \frac{1}{r_{layout}} & Otherwise \end{cases} \tag{36}$$

$$r_i = \frac{h_i}{b_i} \tag{37}$$

$$r_{layout} = \frac{h_{layout}}{b_{layout}} \tag{38}$$

where b_i and h_i are the width and height of the object i; b_{layout} and h_{layout} are the width and height of the layout; p_j is the proportion of shape j.

5.8. Simplicity

Simplicity is achieved with easy-to-understand and clutter-free interfaces, which in our case, as few points of alignment as possible should be created using as few elements as possible[22]. This gives us the following equation:

$$SMM = \frac{3}{n_{vap} + n_{hap} + n} \in [0, 1] \tag{39}$$

where n_{vap} and n_{hap} are the distinct vertical and horizontal aligment point count; and n is the number of objects in the layout.

5.9. Density

Density represents how much the screen is covered with the layout. Since Smart Layout covers the whole given resolution, we gave this parameter a weight of 0 as it would always give 0 as result. Density can be achieved by having a spaced out layout. The more open space the layout has the more aesthetic score it will get out of density parameter.

$$DM = 1 - \frac{\sum_i^n a_i}{a_{frame}} \in [0,1] \tag{40}$$

Here a_i and a_{frame} represents the areas of the object i and the $frame$; and n is the number of objects in the layout.

5.10. Regularity

Regularity is more common in consistent interfaces, as if elements follow a pattern. In order to achieve regularity in screen design, as few alignment points as possible should be used, keeping the gaps as similar distances as possible. Ngo claims that regularity is a uniformity of elements based on some principle or plan[22].

$$RM = \frac{\left|RM_{alignment}\right| + \left|RM_{spacing}\right|}{2} \in [0,1] \tag{41}$$

$RM_{alignment}$, deals with minimizing alignment points

$$RM_{alignment} = 1 - \frac{n_{vap} + n_{hap}}{2n} \tag{42}$$

$RM_{spacing}$, deals with how consistently spaced the elements are in between themselves

$$RM_{spacing} = \begin{cases} 1 & n = 1 \\ 1 - \frac{n_{spacing} - 1}{2(n-1)} & Otherwise \end{cases} \tag{43}$$

where n_{vap} and n_{hap} are the distinct vertical and horizontal aligment point count; $n_{spacing}$ is the number of distinct distances between column and row starting points; and n is the number of objects in the layout.

5.11. Economy

Economy is to explain what is meant to be told by using as few patterns, elements, techniques and visuals as possible[22].

$$ECM = \frac{1}{n_{size}} \in [0,1] \tag{44}$$

where n_{size} is the number of distinct sizes of elements.

5.12. Homogeneity

Homogeneity measures how evenly the elements are distributed on the page. Here, if we divide the page into four quadrants, we can attribute the homogeneity to the equality of the number of elements in each quadrant.

$$HM = (1 - HT)^2 \in [0,1] \tag{45}$$

$$HT = \frac{\sum \frac{\left|\frac{n}{4} - n_j\right|}{\frac{n}{4}}}{6} \tag{46}$$

where n_j is the number of objects in quadrant j; and n is the number of objects in the layout.

5.13. Rhythm

Rhythm in design refers to the regular patterns of change in items. Rhythm is achieved by diversifying the order, size, number and shape of the elements[22]. To measure the objects that are systematically ordered following equation is used:

$$RHM = 1 - \frac{|RHM_x| + |RHM_y| + |RHM_{area}|}{3} \in [0,1] \tag{47}$$

$$RHM_x = \frac{\begin{array}{l}\left|X'_{UL} - X'_{UR}\right| + \left|X'_{UL} - X'_{LR}\right| + \\ \left|X'_{UL} - X'_{LL}\right| + \left|X'_{UR} - X'_{LR}\right| + \\ \left|X'_{UR} - X'_{LL}\right| + \left|X'_{LR} - X'_{LL}\right|\end{array}}{6} \tag{48}$$

$$RHM_y = \frac{\begin{array}{l}\left|Y'_{UL} - Y'_{UR}\right| + \left|Y'_{UL} - Y'_{LR}\right| + \\ \left|Y'_{UL} - Y'_{LL}\right| + \left|Y'_{UR} - Y'_{LR}\right| + \\ \left|Y'_{UR} - Y'_{LL}\right| + \left|Y'_{LR} - Y'_{LL}\right|\end{array}}{6} \tag{49}$$

$$RHM_{area} = \frac{\begin{array}{l}\left|A'_{UL} - A'_{UR}\right| + \left|A'_{UL} - A'_{LR}\right| + \\ \left|A'_{UL} - A'_{LL}\right| + \left|A'_{UR} - A'_{LR}\right| + \\ \left|A'_{UR} - A'_{LL}\right| + \left|A'_{LR} - A'_{LL}\right|\end{array}}{6} \tag{50}$$

where X'_j, Y'_j, A'_j are, respectively, the normalised values of X_j, Y_j and A_j

$$X_j = \sum_{i}^{n_j} \left|x_{ij} - x_c\right| \quad j = UL, UR, LL, LR \tag{51}$$

$$Y_j = \sum_{i}^{n_j} \left|y_{ij} - y_c\right| \quad j = UL, UR, LL, LR \tag{52}$$

$$A_j = \sum_{i}^{n_j} a_{ij} \quad j = UL, UR, LL, LR \tag{53}$$

where j can get the quadrants upper-left (UL), upper-right (UR), lower-left (LL), lower-right (LR). (x, y) pairs are center coordinates for object i on quadrant j and frame c; a_{ij} is the area of the object; and finally n_j is the total number of objects on that quadrant.

5.14. Order and Complexity

This parameter is the scale of complexity of the layout which is calculated as the mean of all the previous aesthetic category scores. Since fine-tuning all of the parameters' weight coefficients is an optimization problem, we assumed each parameter as equally important, with the exemptions that we mentioned in their own subsections, and it still gave good results.

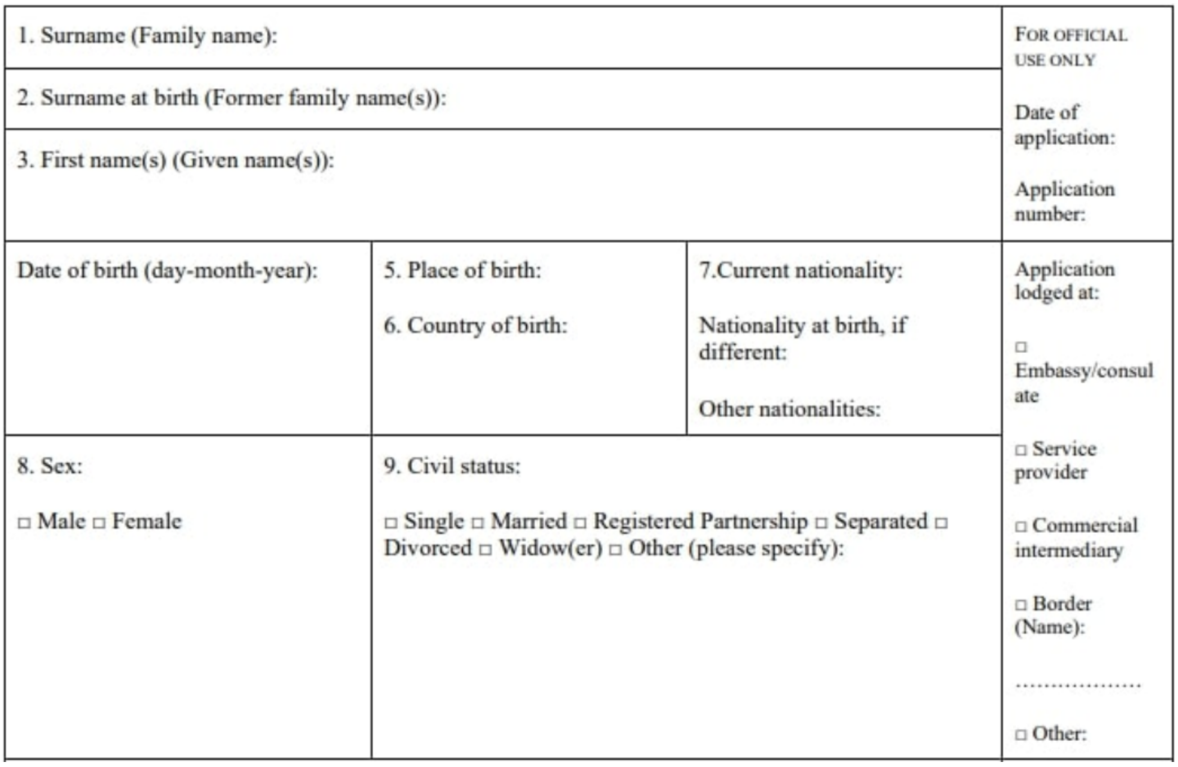
$$\frac{\sum_i^n P_i}{n} \in [0, 1] \tag{54}$$

where P_i is the value of parameter i; and n is the number of parameters that we use to score the aesthetic points.

6. EXPERIMENTS & RESULTS

6.1. Experimental Layouts

We used a fragment from the first page of the Schengen visa form to make a comparison with real life usage which can be seen in Figure 6.1. We recreated this layout as four separate layouts with 10, 31, 51 and 84 components. We tested all the layouts in 3 different resolutions. We chose one large, one equal, one tall resolution and got the visuals of the results found by the algorithm.



1. Surname (Family name):			FOR OFFICIAL USE ONLY Date of application: Application number:
2. Surname at birth (Former family name(s)):			
3. First name(s) (Given name(s)):			
Date of birth (day-month-year):	5. Place of birth: 6. Country of birth:	7.Current nationality: Nationality at birth, if different: Other nationalities:	Application lodged at: □ Embassy/consulate □ Service provider □ Commercial intermediary □ Border (Name): .... □ Other:
8. Sex: □ Male □ Female	9. Civil status: □ Single □ Married □ Registered Partnership □ Separated □ Divorced □ Widow(er) □ Other (please specify):		

FIGURE 6.1. Original layout

Layout 1

We tried to preserve all the components for the first layout and created a layout with 84 components.

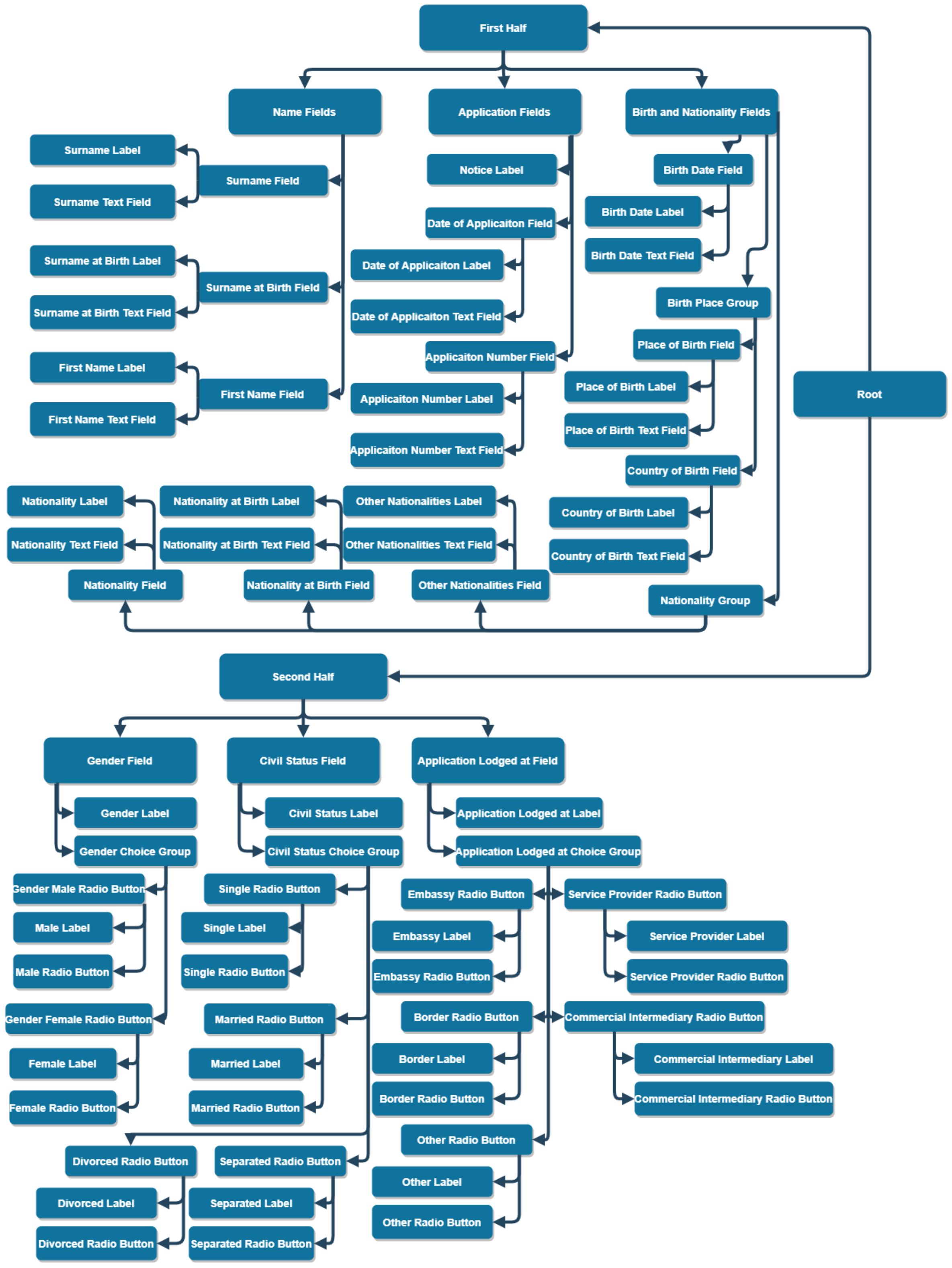


FIGURE 6.2. Layout 1, composed of 84 components, which are 48 UI components and 36 containers

Wide Resolution (1000, 270)

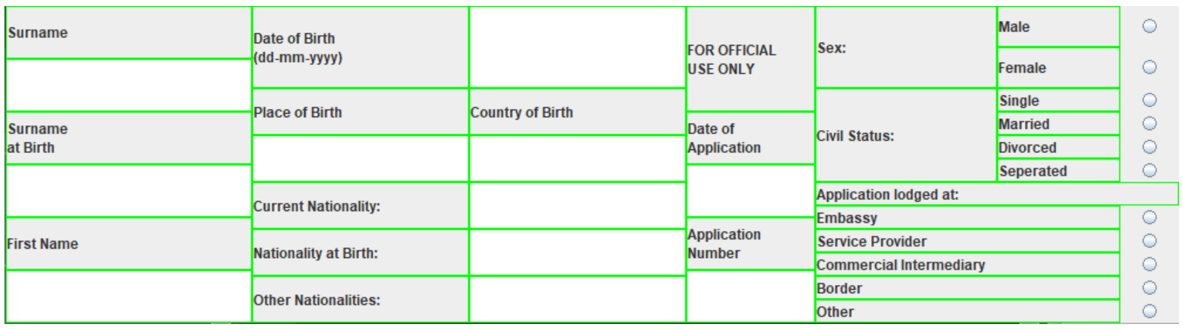


FIGURE 6.3. Layout 1, 1000 pixels wide, 270 pixels tall

After running Smart Layout, its response to a resolution of 1000 by 270 is as in Figure 6.3. In some places, the outlines of the given groups can be seen sharply. However, this is a result where Gestalt principles have been applied as much as possible. As a matter of fact, the salience of relational groups shows that Gestalt principles are doing its' work.

Square Resolution (600, 600)

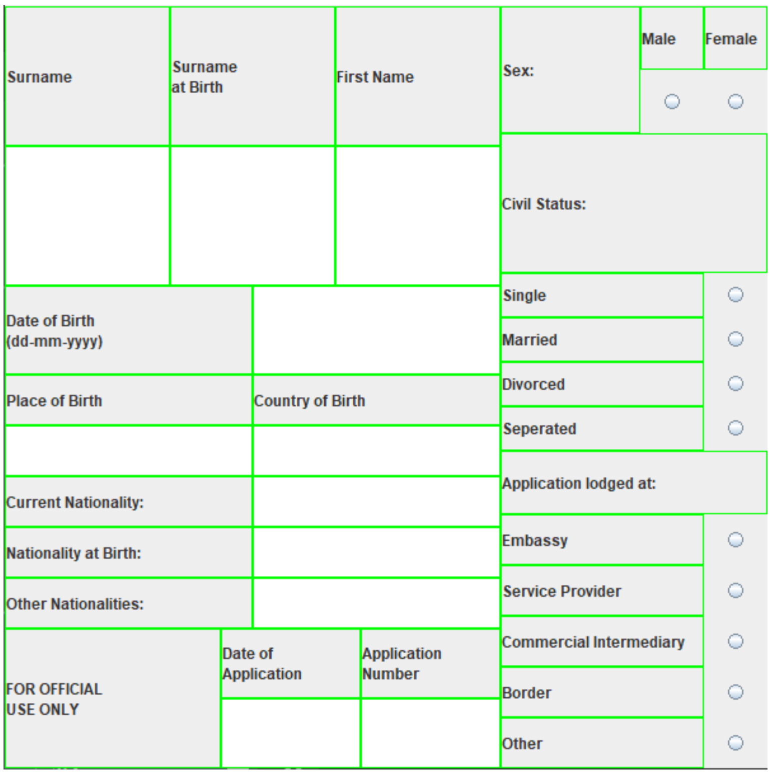


FIGURE 6.4. Layout 1, 600 pixels wide, 600 pixels tall

When we move on to a square installation, we see that the groups still do not lose their distinction. However, the dimensions have been kept the same as possible and there is a general consistency in areas other than *Place of Birth* and *Country of Birth*. Some areas may be too large to appeal to the eye, but since these are completely developers' values, they can be modified. The reason we gave such a large range here is that the larger our range, the more our possible options increased and we had the opportunity to find more layouts. In addition, we wanted to make a stress test on the application as a performance evaluation.

Tall Resolution (400, 600)

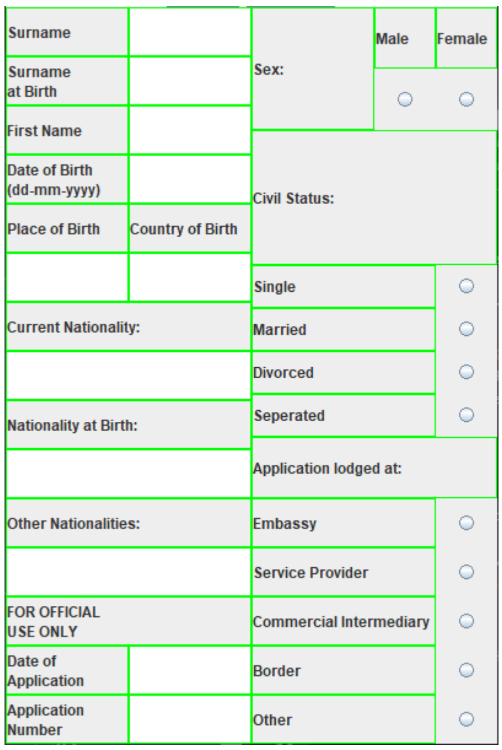


FIGURE 6.5. Layout 1, 400 pixels wide, 600 pixels tall

In this layout, we see a resolution closer to mobile and tablets. We see that vertical layouts are preferred than horizontal layouts in order to fit more.

Layout 2

In the second layout, we tried to keep everything the same as possible, and that's why we created a 51 component layout using only fewer components.

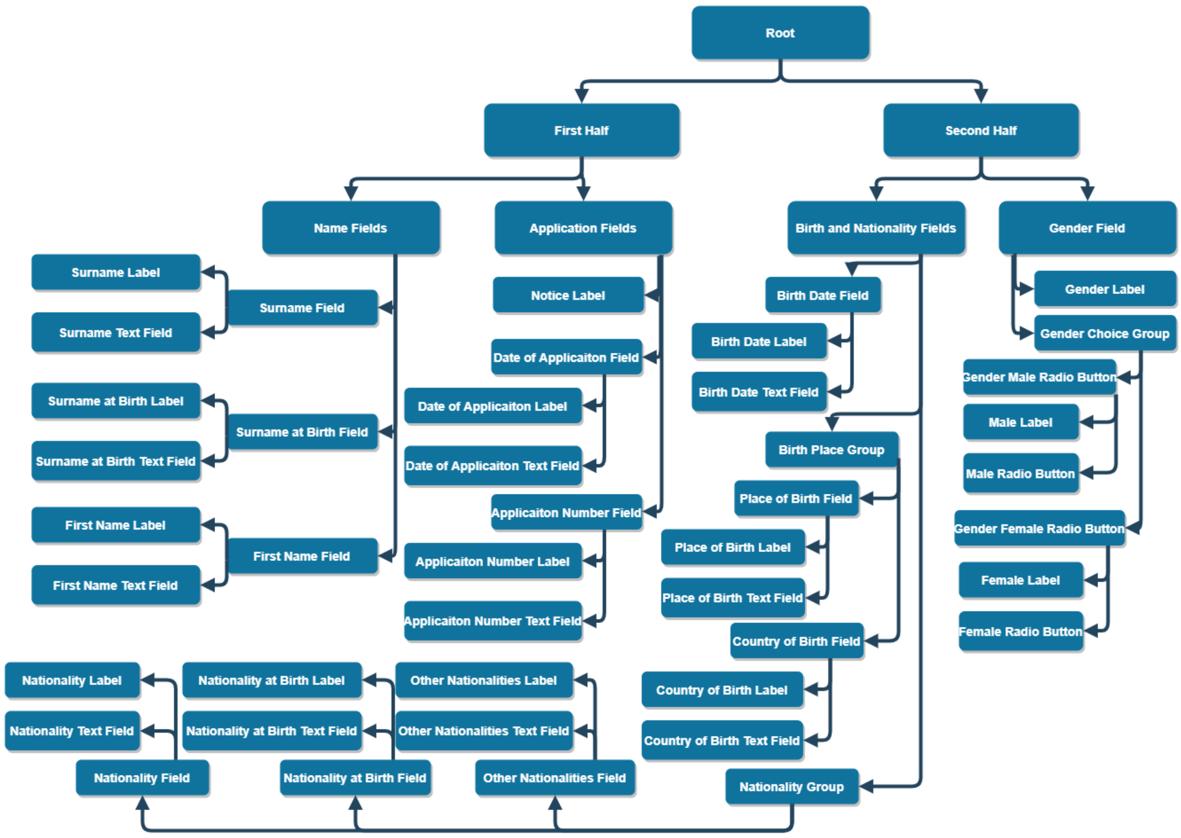
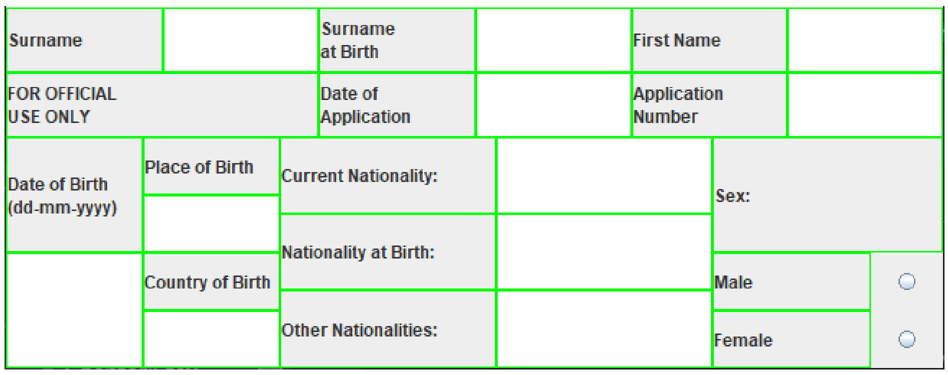


FIGURE 6.6. Layout 2, composed of 51 components, which are 28 UI components and 23 containers

Wide Resolution (650, 250)



Surname

Surname at Birth

First Name

FOR OFFICIAL USE ONLY

Date of Application

Application Number

Date of Birth (dd-mm-yyyy)

Place of Birth

Country of Birth

Current Nationality:

Nationality at Birth:

Other Nationalities:

Sex:

Male

Female

FIGURE 6.7. Layout 2, 650 pixels wide, 250 pixels tall

In the wide resolution, as can be seen, again, the relationships can be observed by grouping them with Gestalt principles. However, another observed feature is the grouping of the upper branch nodes with each other. A structure resembling a Grid Layout has been obtained and has earned the highest aesthetic score.

Square Resolution (400, 400)

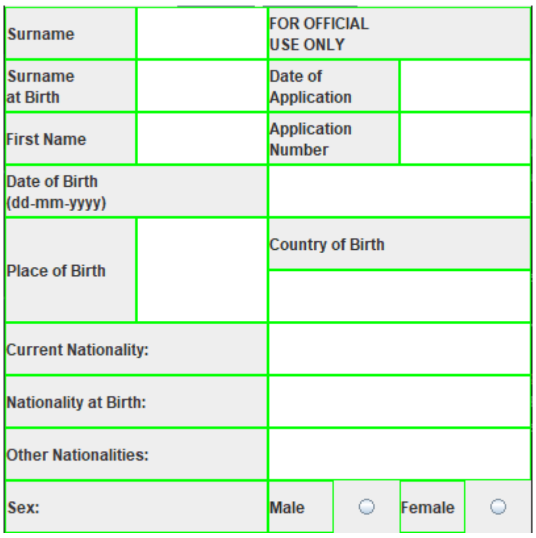


FIGURE 6.8. Layout 2, 400 pixels wide, 400 pixels tall

In square resolution, the algorithm sees the two column placement as the most appropriate and tried to do the distribution with this logic. However, minor inconsistencies have been observed. For example, while the majority of the fields are accumulated on either side of the columns (*Name Fields* etc.), some fields spread to two columns, breaking the integrity (*Country of Birth* etc.).

Tall Resolution (300, 600)

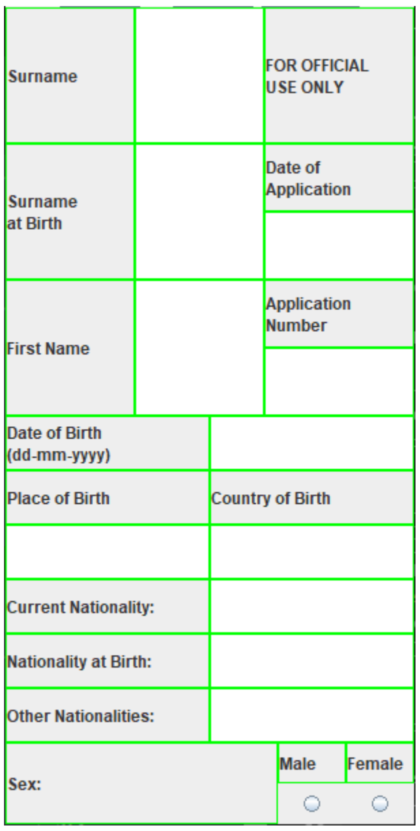


FIGURE 6.9. Layout 2, 300 pixels wide, 600 pixels tall

Here we see the main two nodes placed vertically which gave a phone or tablet like layout. Same inconsistencies can be found here but of course these inconsistencies can be avoided by adjusting the weights of the aesthetic scoring parameters. For instance, if we adjust *Symmetry* coefficient to higher numbers, radial symmetry will be more important which will result in more radial equalities across the layout hence choosing a layout that is more similar in i.e. top-left and bottom-right. But this is a whole other branch of aesthetic correction that is not in the scope of this thesis, although it would be a promising future work.

Of course this result might be one of the lowest aesthetic scores this layout can get, just because this was the only feasible layout or best of the worst layouts since none other layout contains a feasible solution to this specific screen resolution. The scoring of each layout for each resolution will be discussed in aesthetic results part.

Layout 3

In layout 3, the component count is reduced to 31 so we can observe how the relational tree and layout algorithm responds to fewer component distributions and how these distributions are scored.

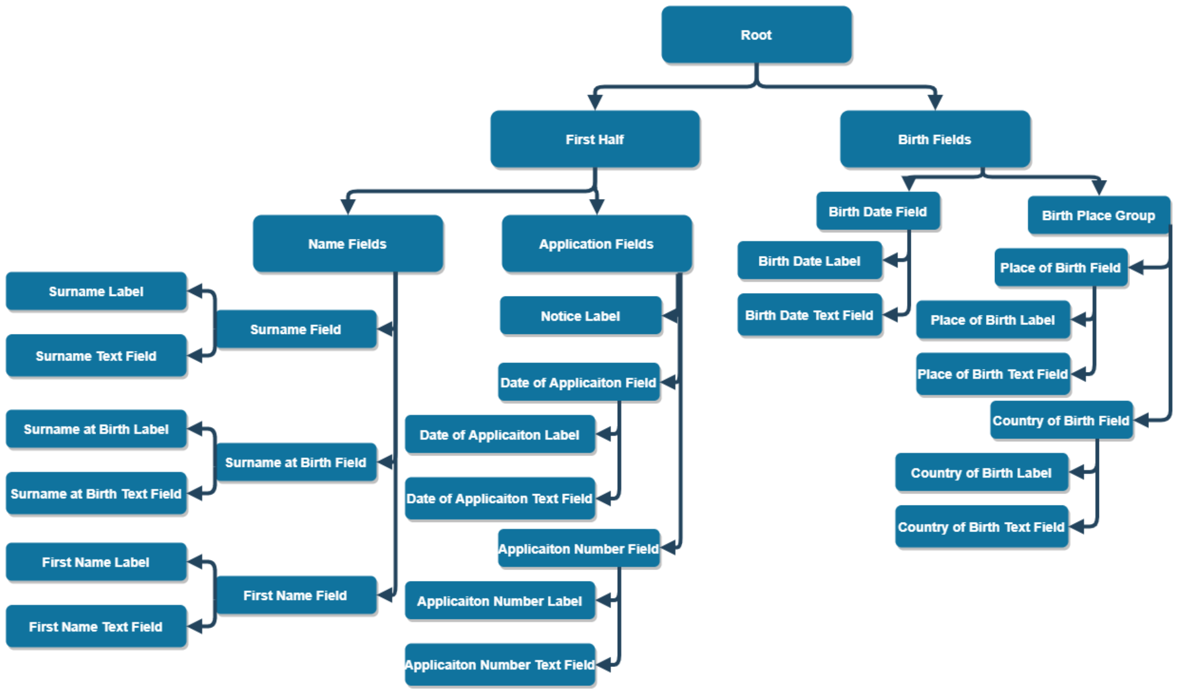


FIGURE 6.10. Layout 3, composed of 31 components, which are 17 UI components and 14 containers

Wide Resolution (400, 200)

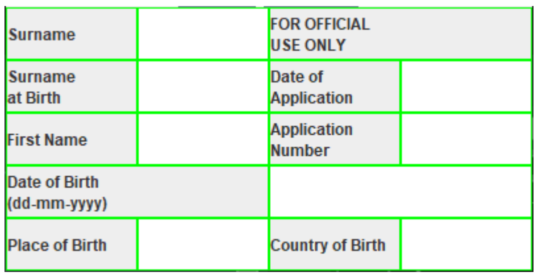


FIGURE 6.11. Layout 3, 400 pixels wide, 200 pixels tall

In the wide layout, *Date of Birth* field is used as a filler, because this field can still get as low as other text input fields but can be extended twice as much as a normal text field. When a node can stretch further, its possible solutions grow exponentially which creates these kinds of layout opportunities that normally does not satisfy at such resolutions. If *Date of Birth* fields couldn't stretch twice as much as a normal text field node, this layout would be a text field and label short to fill in and might become an infeasible layout.

Square Resolution (300, 300)

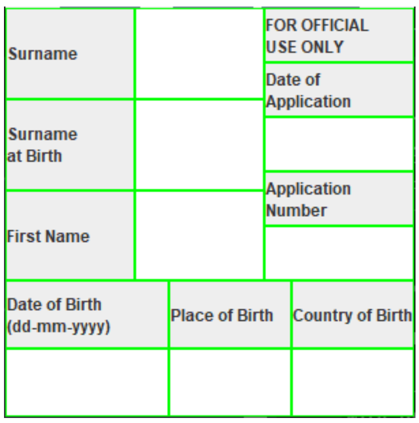


FIGURE 6.12. Layout 3, 300 pixels wide, 300 pixels tall

As the layouts get smaller and smaller, the aesthetic scores and possible outcomes become more scarce which results in more deformed layouts that have no better outcome than presented ones. The inconsistencies are even more bold in this layout.

Tall Resolution (200, 400)

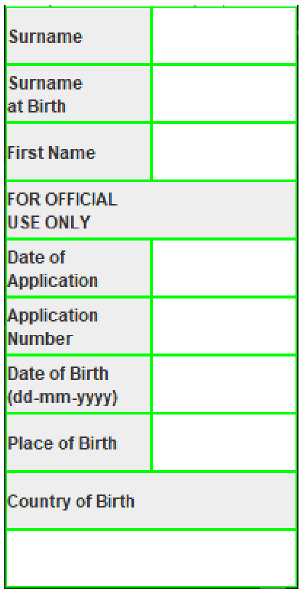


FIGURE 6.13. Layout 3, 200 pixels wide, 400 pixels tall

Aside from the last label and text field the layout is pretty consistent in this specific case, but it is worth noting that this might be the only solution to this layout and the last two components were stretched to fulfill the constraints.

Layout 4

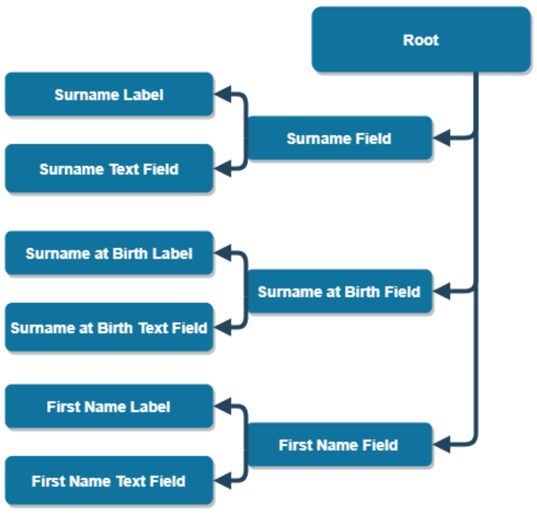


FIGURE 6.14. Layout 4, composed of 10 components, which are 6 UI components and 4 containers

For the final layout we picked as little elements as possible to see how Smart Layout behaves in minimal interfaces such as dialog boxes. The possibilities are pretty limited compared to the other layouts with much more complex structures.

Wide Resolution (600, 100)

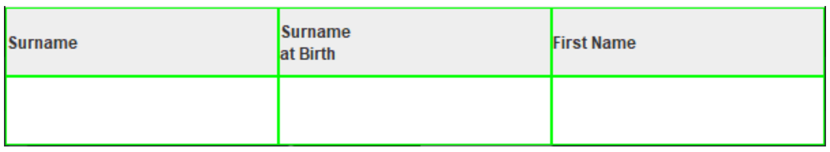


FIGURE 6.15. Layout 4, 600 pixels wide, 100 pixels tall

As predicted, this wide layout kept the labels at top and fields at bottom and if the layout was extended further all the components can only be placed horizontally to fulfill the constraints.

Square Resolution (200, 200)

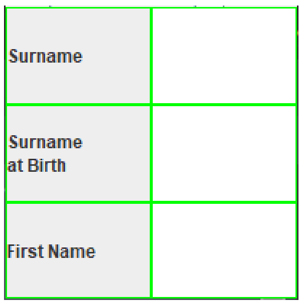


FIGURE 6.16. Layout 4, 200 pixels wide, 200 pixels tall

As it can easily be seen, for the horizontal arrangement, both the horizontal and vertical constraints of the components cannot accommodate this resolution. If this resolution is further narrowed down, placing each component one under the other seems to be the last option.

Tall Resolution (200, 300)

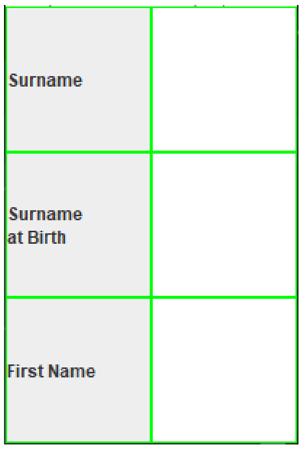


FIGURE 6.17. Layout 4, 200 pixels wide, 300 pixels tall

Here we see no significant changes to the square resolution as it can still fit the components vertically and vertical placement gives more aesthetic score because of *Proportion* parameter. Stretched objects gets lower score than equally proportioned objects as it can be calculated in Equation 32.

6.2. Results

In this section, we will be comparing the numerical values of the layouts we have. These numerical values include possible number of layouts, time complexity and aesthetic scores of layouts. The results are taken from a Java application, as secluded as it can be with restarting everytime and made one measurement beforehand to avoid cold starts.

6.2.1. Possible Layouts

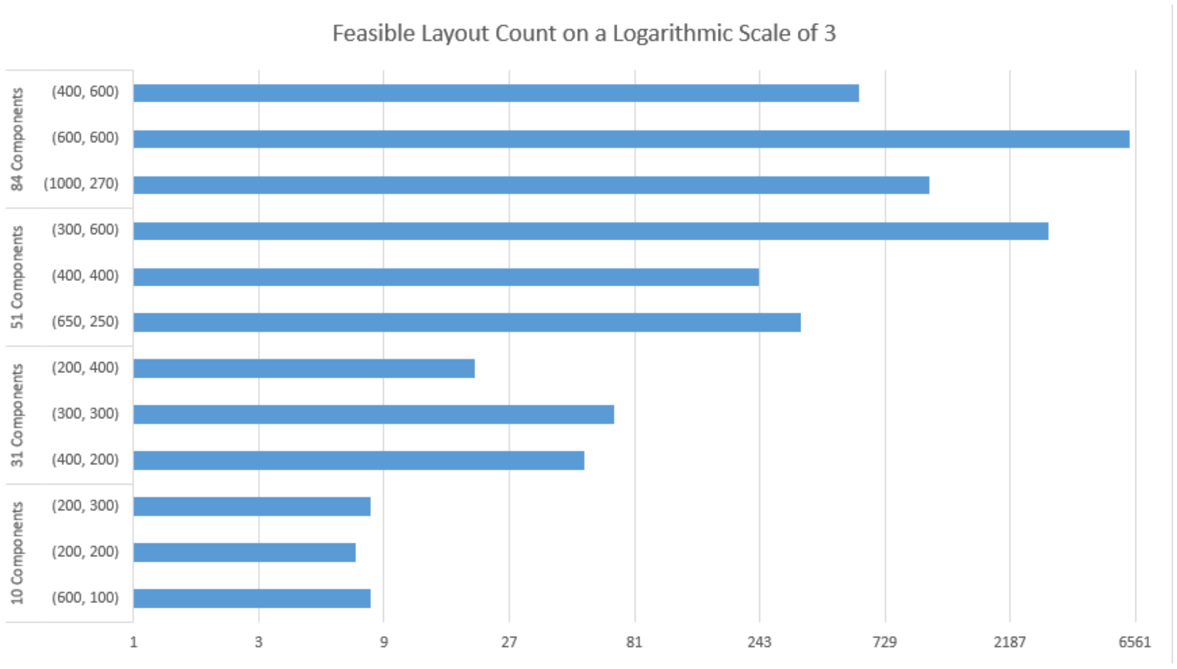


FIGURE 6.18. Feasible Layout Count for all layouts and resolutions experimented.

When we look at Figure 6.18., the first thing to notice is that certain resolution ranges give more results in each layout. In other words, the value ranges we give to the components seem to be most open to change at these points. In these resolutions, we have the biggest sample space for aesthetic scoring compared to the other two resolutions. These resolutions are, (600, 100) and (200, 300) for 10 components, (300, 300) for 31 components, (300, 600) for 51 components, and (600, 600) for 84 components. If we compare these resoultions to their siblings of the same layout tree, we might not find the most aesthetically pleasing result, but we find which layout had the biggest sample space for our aesthetic evaluation. As expected with more components feasible layout count increases with consideration of the tree's elements and its' current resolution.

6.2.2. Time Complexity

Time was the most important metric in our study since we aim Smart Layout to work under a second and if this metric didn't give us these results, the study would not be a success. To understand how it works in real time, we also took the time measurements while the GUI was running so that we could show that it still runs under a second despite other intervening operations. In Table-6.3., as expected, it is observed that as the number of feasible layouts increases, the time finding these layouts and the aesthetic scoring periods also increase. However, despite having a screen with 84 components with (600, 600) resolution, the entire process can be calculated in under a second. We have also noticed that with more components, it is harder to get higher scores on aesthetics. Also even with high number of components, finding all feasible layouts can be measured with microseconds.

TABLE 6.1. Timing values of a cycle

	Finding all feasible layouts (μs)	Total feasible layout count	Aesthetic scoring time (μs)	Total time (μs)
10 Comp - 600,100	3096	8	7791	10887
10 Comp - 200,200	2067	7	5466	7533
10 Comp - 200,300	1347	8	4253	5599
31 Comp - 400,200	317	52	5147	5463
31 Comp - 300,300	363	68	6993	7356
31 Comp - 200,400	694	20	1270	1964
51 Comp - 650,250	264	348	17122	17386
51 Comp - 400,400	296	242	10276	10570
51 Comp - 300,600	498	3077	153564	154061
84 Comp - 1000,270	3597	1074	93397	96994
84 Comp - 600,600	10615	6225	581954	582569
84 Comp - 400,600	341	580	48868	49209

In Table-6.1., breakdown of the cycle can be seen with proper values. We can connect the spike in all feasible layout finding times in the 10-component tree to more branch node trips because it could not find appropriate layouts. It should be noted that this spike can relate to the Java languages overheads since this measurement was snapshotted with GUI which may caused the spike or garbage collection system that Java provides.

TABLE 6.2. Execution Time of Get Ranges algorithm (ms)

	10 Components	31 Components	51 Components	84 Components
With Memoization	0.3	0.6	0.7	1
No Memoization	128	2816	36415	43340

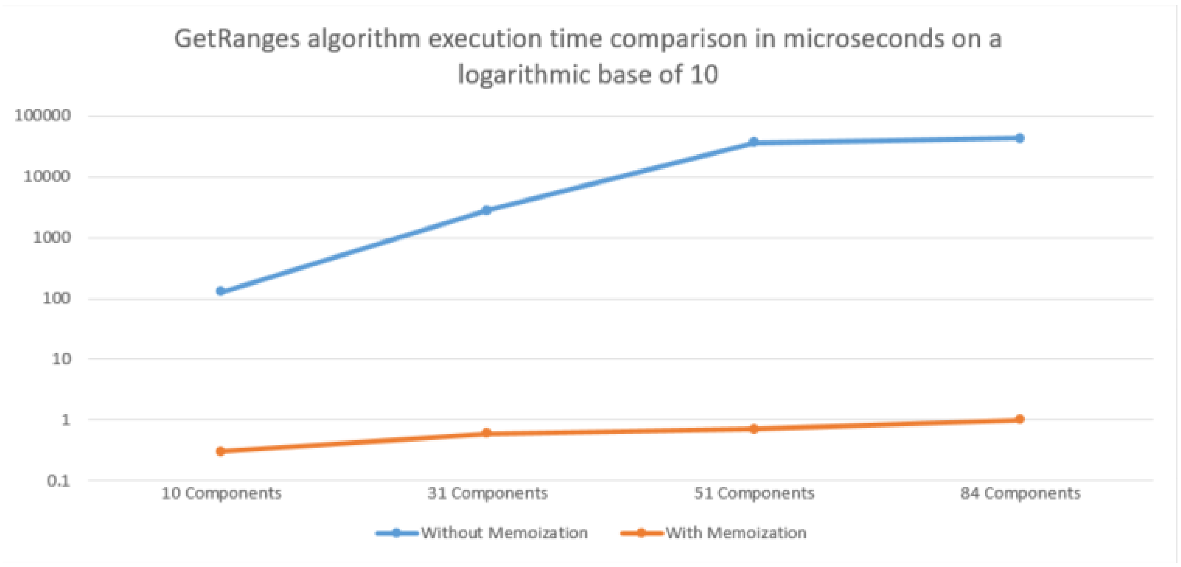


FIGURE 6.19. Execution time of creating relational tree on a logarithmic base of 10 for all layouts and resolutions experimented in microseconds.

If we look at the timing of our GetRanges function, which checks whether a layout is feasible for a certain resolution according to the tree structure we have, here in Table-6.2. and Figure 6.19., we can see the comparison of our results using the memoization structure to prune the tree and the results measured with a normal tree traversal method without using memoization. Since the GetRanges function works only once for a specified tree, no more data is obtained from the measurements, but as the data clearly shows, more is not needed. As Table-6.2. and Figure 6.19. show, memoization has saved a significant exponential increase in time and pruned the tree, avoiding many unnecessary branches.

TABLE 6.3. Aesthetic Scores and Execution Times, Feasible Layout Execution Times and Counts

	Aesthetics		Feasible Layout	
	Score	Execution time (ms)	Find all (μs)	Count
10 Comp - 600,100	5.9734	7.79	3096	8
10 Comp - 200,200	6.6462	5.47	2067	7
10 Comp - 200,300	6.6224	4.25	1347	8
31 Comp - 400,200	7.3758	5.15	317	52
31 Comp - 300,300	6.4426	6.99	363	68
31 Comp - 200,400	6.1904	1.27	694	20
51 Comp - 650,250	6.4012	17.12	264	348
51 Comp - 400,400	6.8584	10.28	296	242
51 Comp - 300,600	6.1524	153.56	498	3077
84 Comp - 1000,270	5.7745	93.4	3597	1074
84 Comp - 600,600	6.0551	581.95	10615	6225
84 Comp - 400,600	5.6897	48.87	341	580

If we look at the time it takes to find all feasible layouts of a tree, the values in both Figure 6.20. and Table-6.3. show that the number of elements in the interface, the number of feasable layouts found and the time of finding feasible layouts are directly proportional.

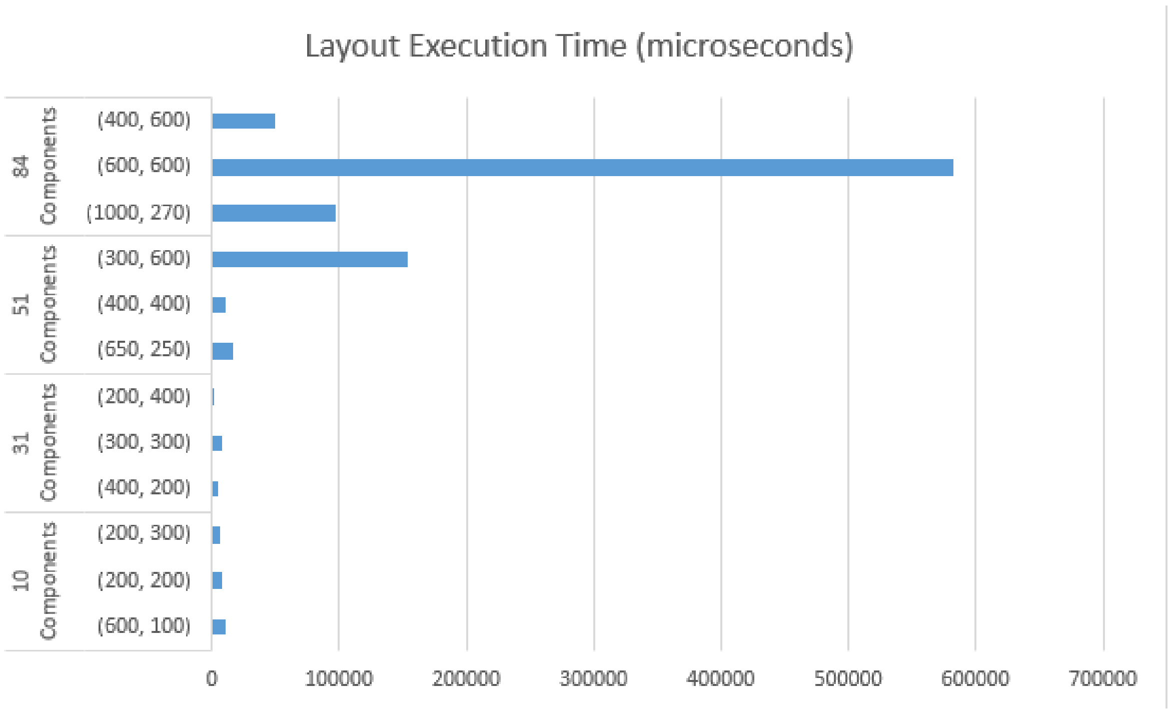


FIGURE 6.20. Execution time to create a list of every feasible layout, for all layouts and resolutions experimented in microseconds.

6.2.3. Aesthetics

Table-6.3. shows that as the interface grows, the aesthetic scores decrease slowly. It would be wrong to make this assumption for each aesthetic parameter, but this was what we expected to see; the overall aesthetic scores decreased. As the interface grows, it will be harder to score because some of our parameters will require fewer elements for simplicity, and it will be harder to fulfill the remaining parameters' conditions. If we need to look at each aesthetic parameter one by one, Table-6.4. and each parameters' individual result figures will help us.

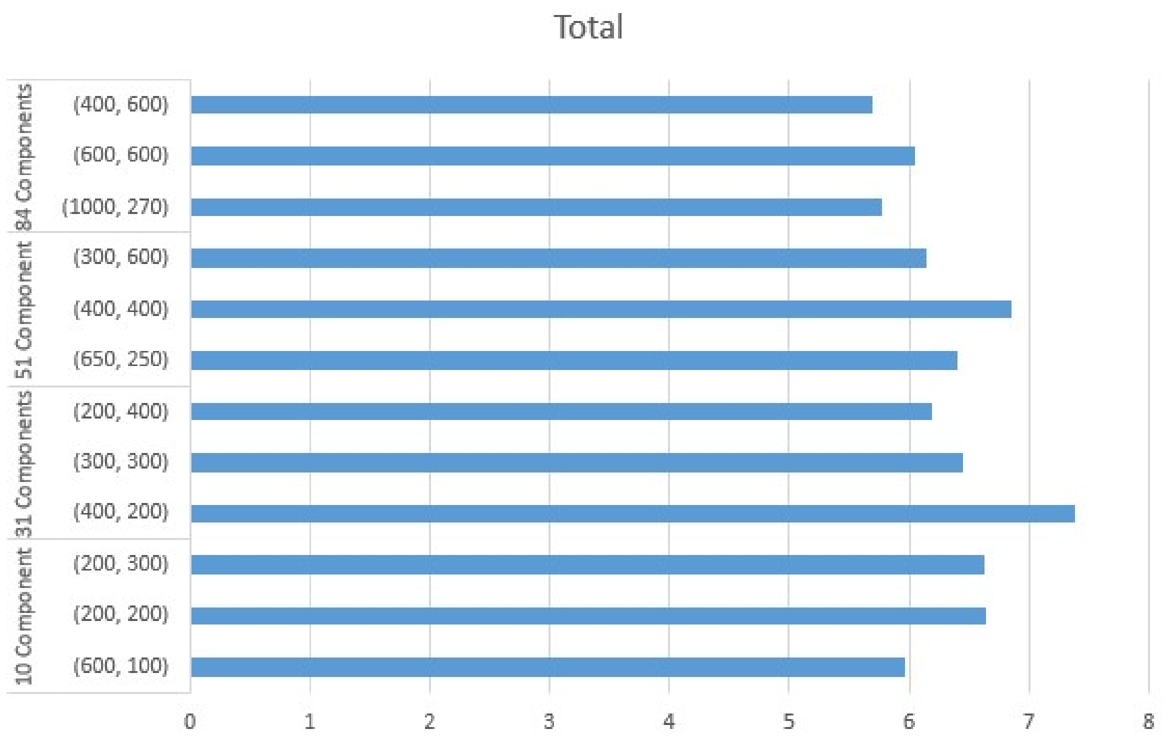


FIGURE 6.21. Total aesthetic results for each resolution.

TABLE 6.4. Aesthetic scores

	10 Components			31 Components			51 Components			84 Components		
Parameter	600,100	200,200	200,300	400,200	300,300	200,400	650,250	400,400	300,600	1000,270	600,600	400,600
Balance	1	0.9925	1	1	0.8248	1	0.7962	1	0.8902	0.8436	0.8095	0.4091
Symmetry	0	0.3333	0	0.4335	0.457	0	0.4377	0.4511	0.4579	0.4205	0.4571	0.4563
Sequence	0.15	0.15	0.125	0.175	0.2	0.15	0.2	0.2	0.2	0.2	0.2	0.2
Cohesion	0.4815	0.6146	0.5208	0.8092	0.5484	0.2318	0.7621	0.4813	0.2741	0.5654	0.5426	0.3449
Unity	1	0.8333	1	0.9412	0.7647	0.9412	0.6786	0.8929	0.8214	0.75	0.75	0.8125
Proportion	0.4166	0.9612	0.9595	0.8647	0.9430	0.8647	0.7758	0.8158	0.8698	0.5539	0.8471	0.8198
Simplicity	0.2308	0.2308	0.2308	0.1071	0.0909	0.0968	0.0612	0.0652	0.0612	0.0411	0.0370	0.0395
Regularity	0.6629	0.6174	0.7083	0.8231	0.6132	0.779	0.6943	0.8211	0.758	0.8119	0.7702	0.8173
Economy	1	0.5	1	0.5	0.2	0.5	0.1	0.25	0.1667	0.0769	0.0769	0.1
Homogenity	0.6049	0.6049	0.6049	0.7786	0.8858	0.7786	1	0.9070	0.7347	0.6945	0.6945	0.892
Rhythm	0	0.3333	0	0.4166	0.4545	0.4062	0.4382	0.4841	0.479	0.4042	0.4376	0.3919
Order and Complexity	0.4267	0.4747	0.473	0.5269	0.4602	0.4421	0.4572	0.4899	0.4395	0.4125	0.4325	0.4064
Total	5.9734	6.6462	6.6224	7.3758	6.4426	6.1904	6.4012	6.8584	6.1524	5.7745	6.0551	5.6897

6.2.3.1. Balance

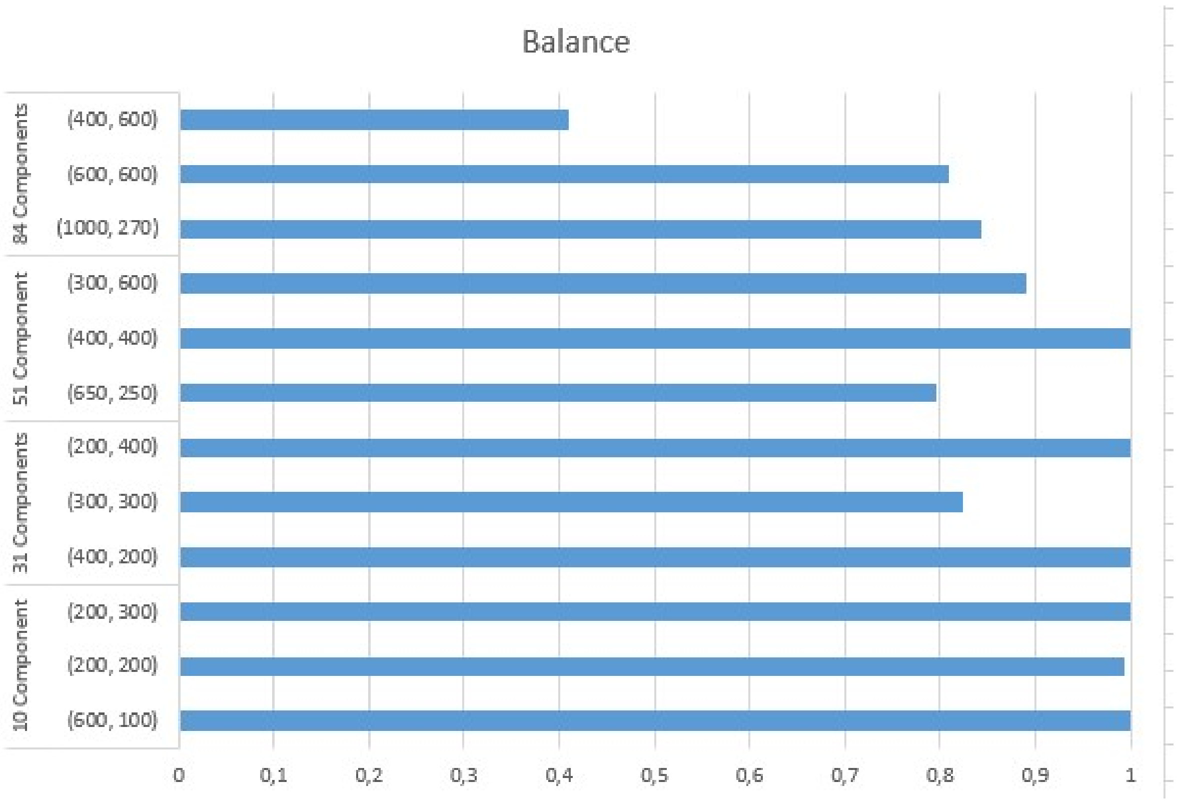


FIGURE 6.22. Balance parameter results for each resolution.

The balance parameter was able to maintain its highest values comfortably in 10 components, and this is because there are few elements in the interface, it can be equally distributed to all four quadrants. However, as the number of components increases, the value has started to decrease, and as it can be predicted in more mobile-like views or horizontally extended views, since the distributions gain horizontal or vertical priority, it is not possible to talk about a balance between the groups as the majority are grouped one below the other or side by side. This deterioration is most clearly seen in the interface with 84 components. As filling the rows in groups is usually left to one or two elements, aesthetic concern has ceased to be the primary priority and we can see from the numbers that the results are reflected in the balance parameter.

6.2.3.2. Symmetry and Rhythm

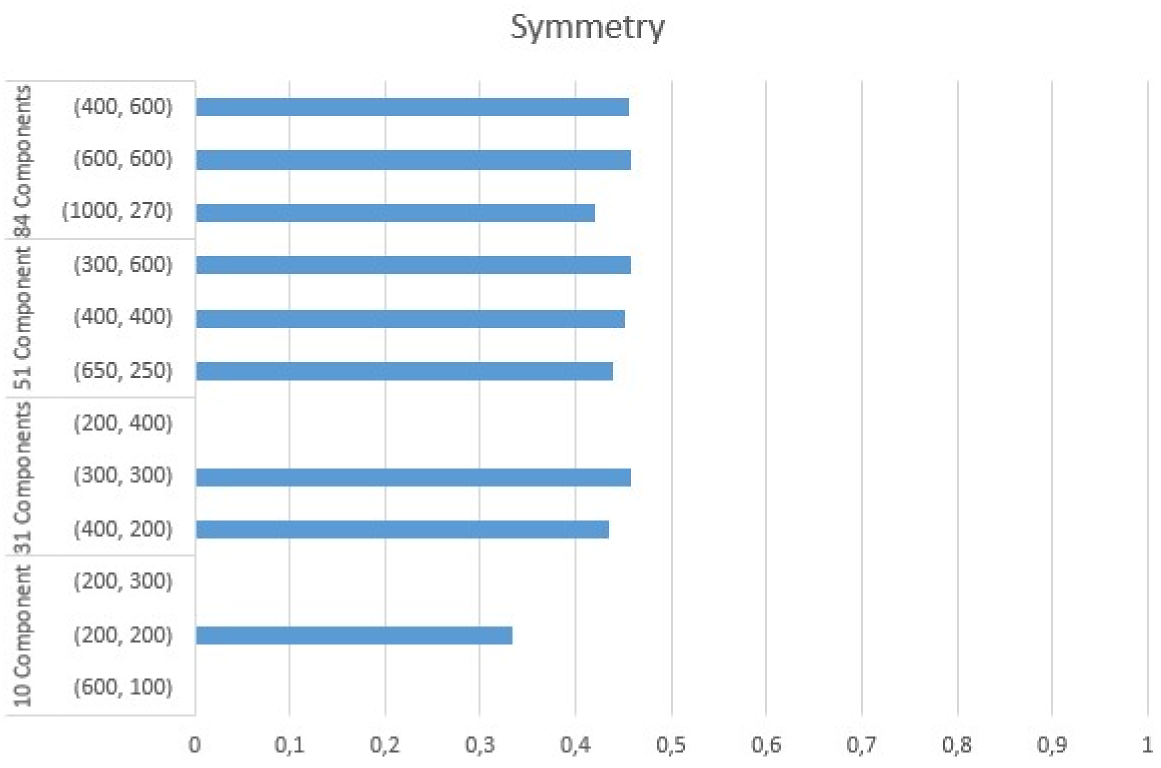


FIGURE 6.23. Symmetry parameter results for each resolution.

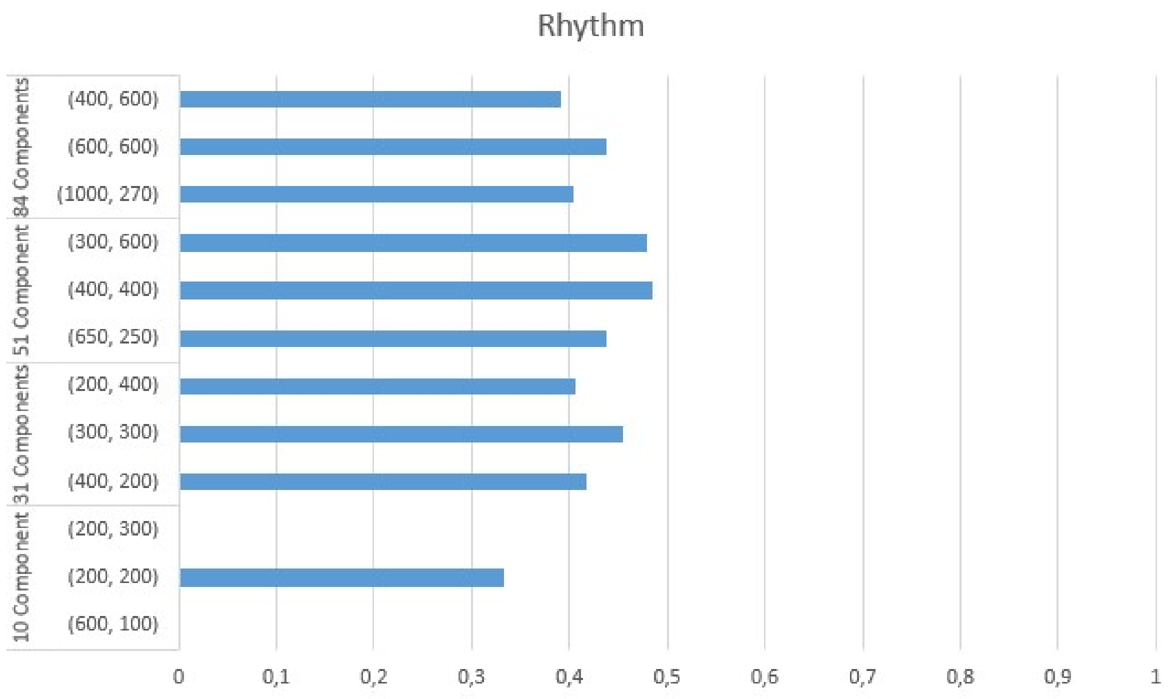


FIGURE 6.24. Rhythm parameter results for each resolution.

Since the equations of symmetry Equation 9 and rhythm Equation 47 are so similar, the values are very much similar too. While rhythm Equation 50 looks at the size difference between the areas of the elements that belong to each quadrant for the four quadrants, symmetry Equation 12 looks at a radial symmetry calculation. Like other parameters, symmetry and rhythm parameters give extreme values in calculations because there is not enough data in 10 component trees. The only significant difference in Figure 6.23. and Figure 6.24. is the interface in Figure 6.13. with 31 components (200, 400) resolution. Although it is impossible to find a visible radial symmetry, it can be seen without calculation that the area differences for each quadrant are similar in the rhythm parameter. This explains us the difference.

6.2.3.3. Sequence

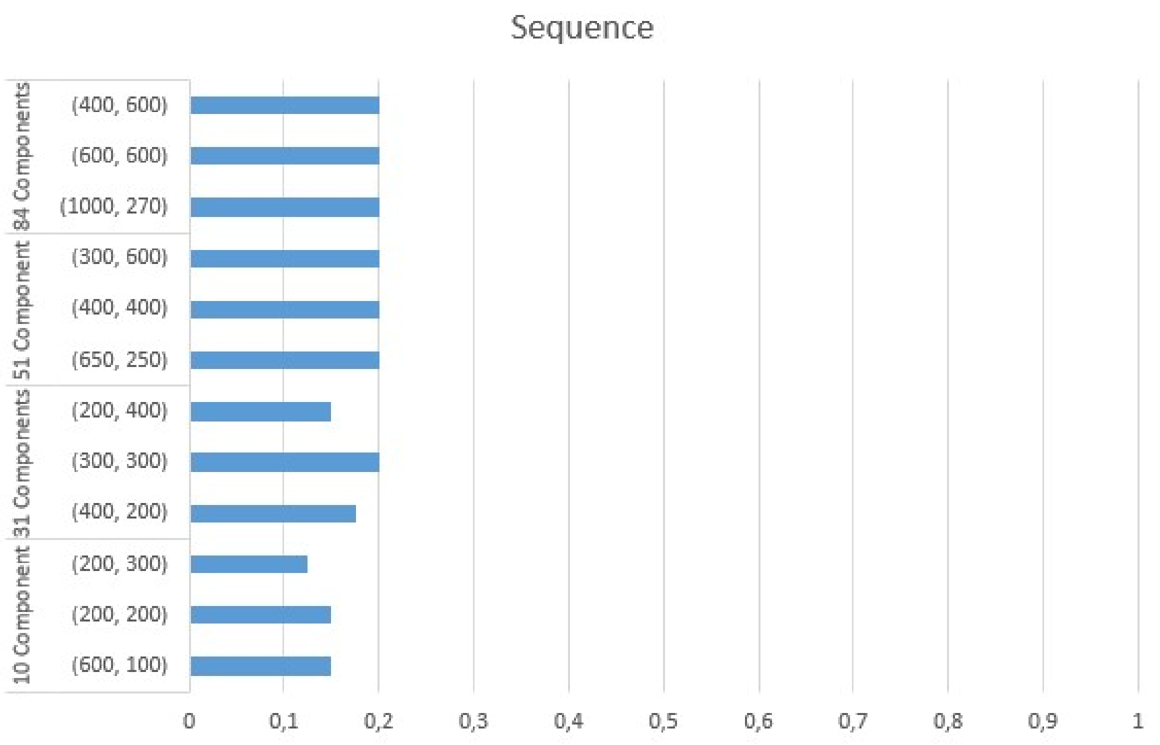


FIGURE 6.25. Sequence parameter results for each resolution.

For the sequence parameter, while the areas in each quadrant are not evenly distributed in the interfaces with less components, as the number of components increases, the sequence value often returns the full value after a certain number of components, since less area overflows from the quadrants. For this reason, its weight was reduced to 0.2 in order not to affect the results too much.

6.2.3.4. Cohesion

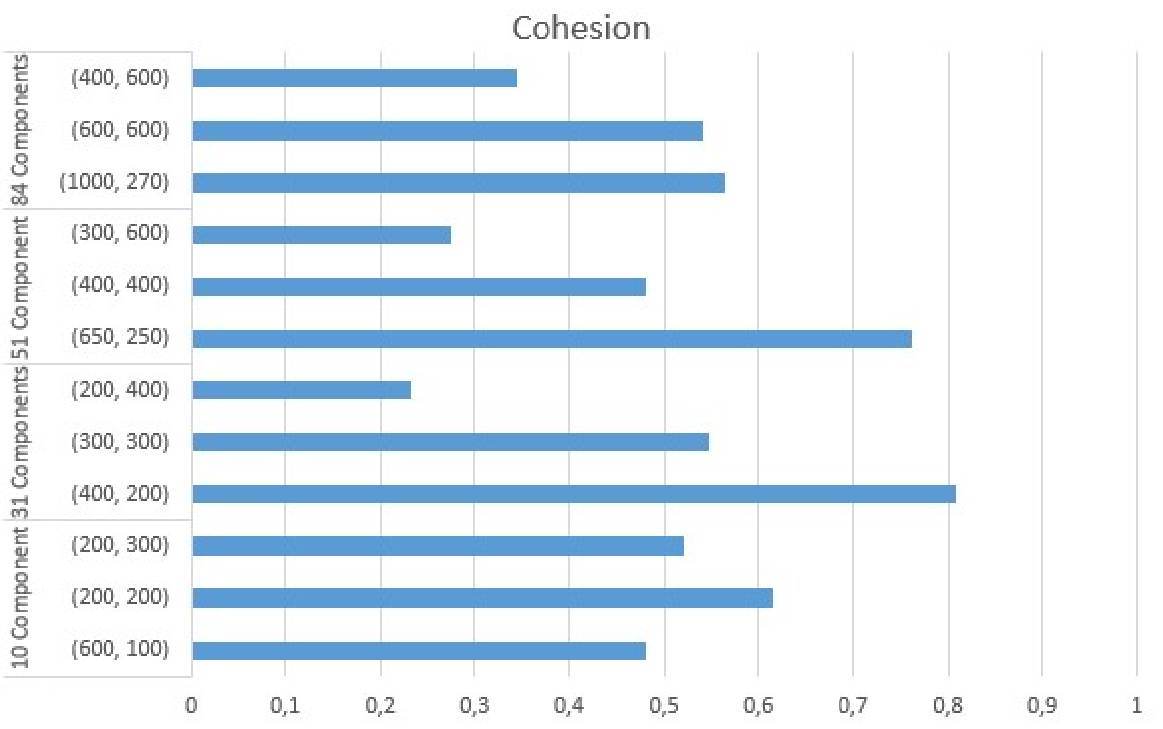


FIGURE 6.26. Cohesion parameter results for each resolution.

Since the cohesion value looks at the ratio of the width and height of the components to the ratio of the resolution to the width and height, it allows us to avoid unnecessary long or wide components. The values it receives can be interpreted even with the naked eye.

6.2.3.5. Unity

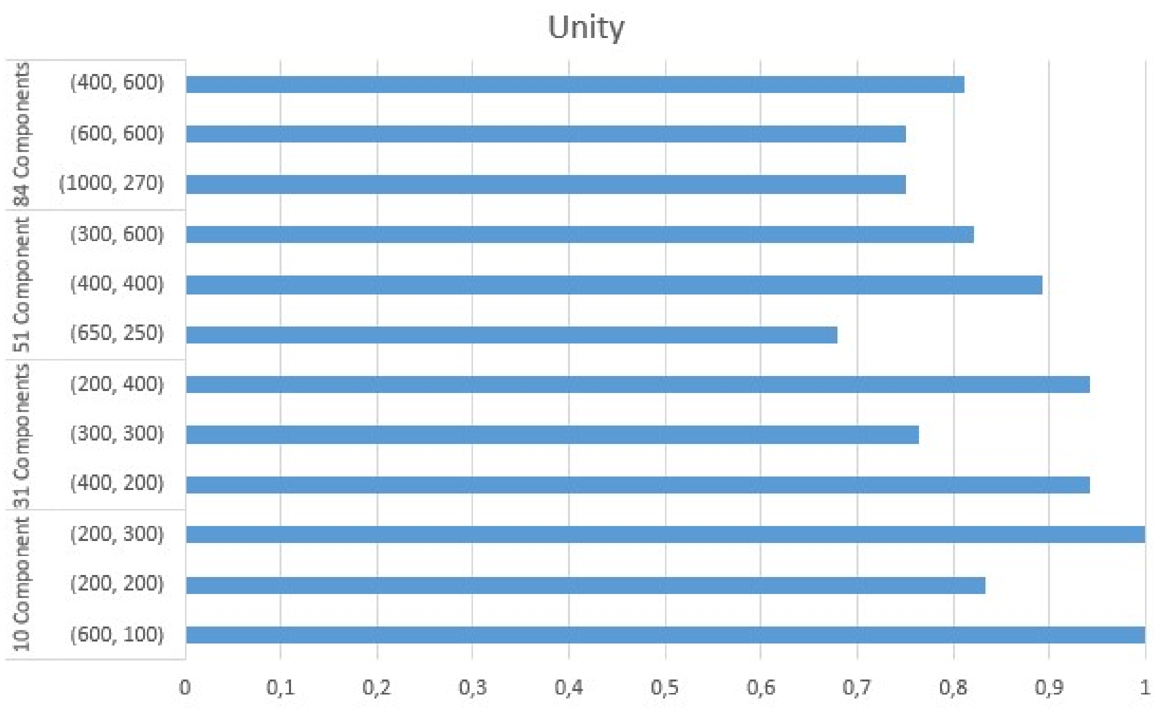


FIGURE 6.27. Unity parameter results for each resolution.

Unity represents how similar the components are to each other, which means; the more components of different sizes, the lower the results. For this reason, the fact that we can find a lot of components with different sizes in an interface with 84 components, while all elements have equal size in an interface with 10 components, shows the reason for the decrease in this parameter.

6.2.3.6. Proportion

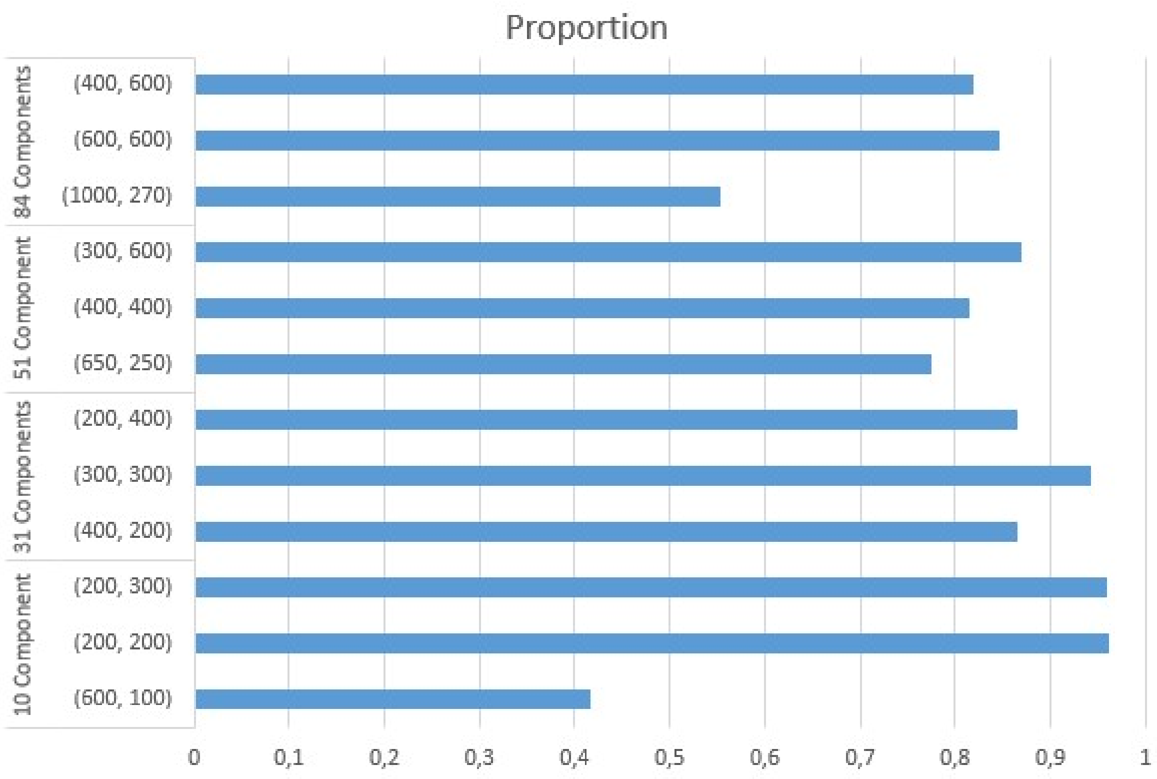


FIGURE 6.28. Proportion parameter results for each resolution.

Proportion measures the similarity of components to some specified proportions (square, square root of two, golden rectangle, square root of three, double square). This helps us to eliminate unnecessarily long or wide components and allows us to keep more eye-catching components.

6.2.3.7. Simplicity

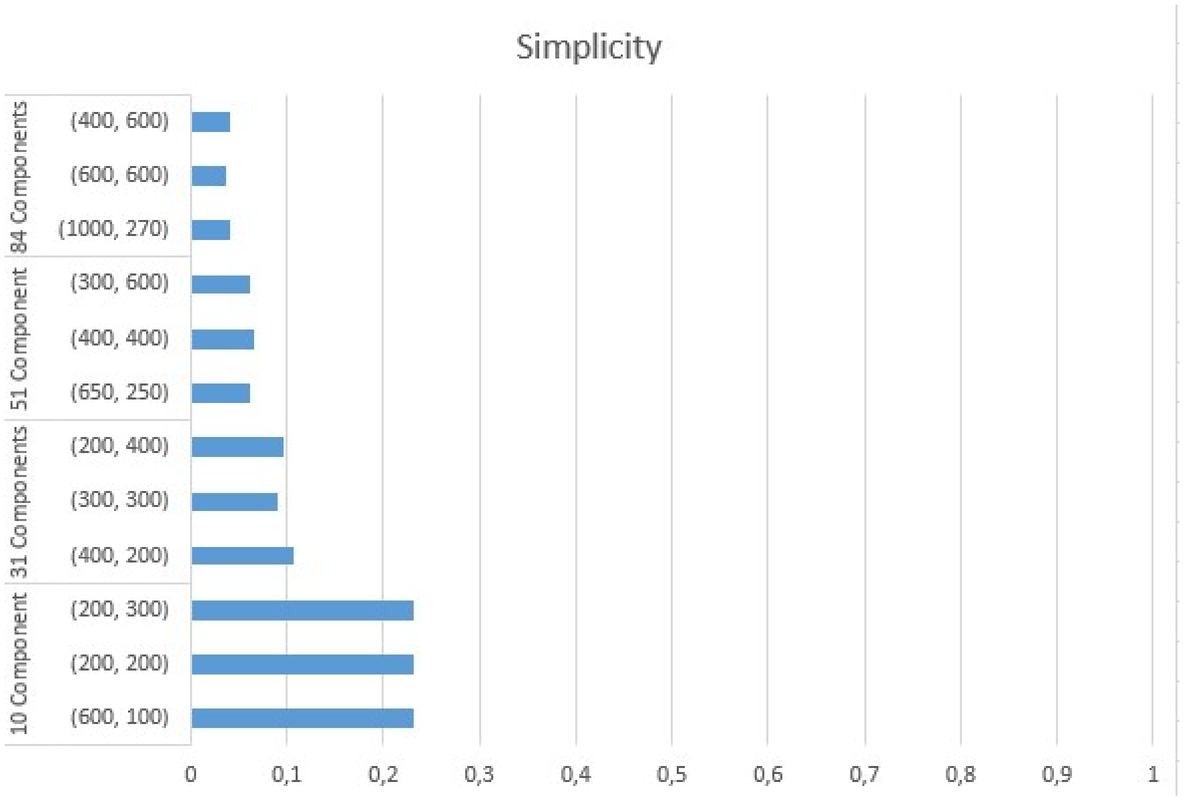


FIGURE 6.29. Simplicity parameter results for each resolution.

In the simplicity parameter, attention is paid to the horizontal and vertical alignment points and the number of elements. Thus, structures similar to a grid structure are preferred. Gestalt's Common Region principle is used by obtaining a grid structure between groups. The more aligned an interface is, better results it gets, but as the interface grows, its is harder to achieve fewer alignment points which results in lower points from this parameter.

6.2.3.8. Regularity

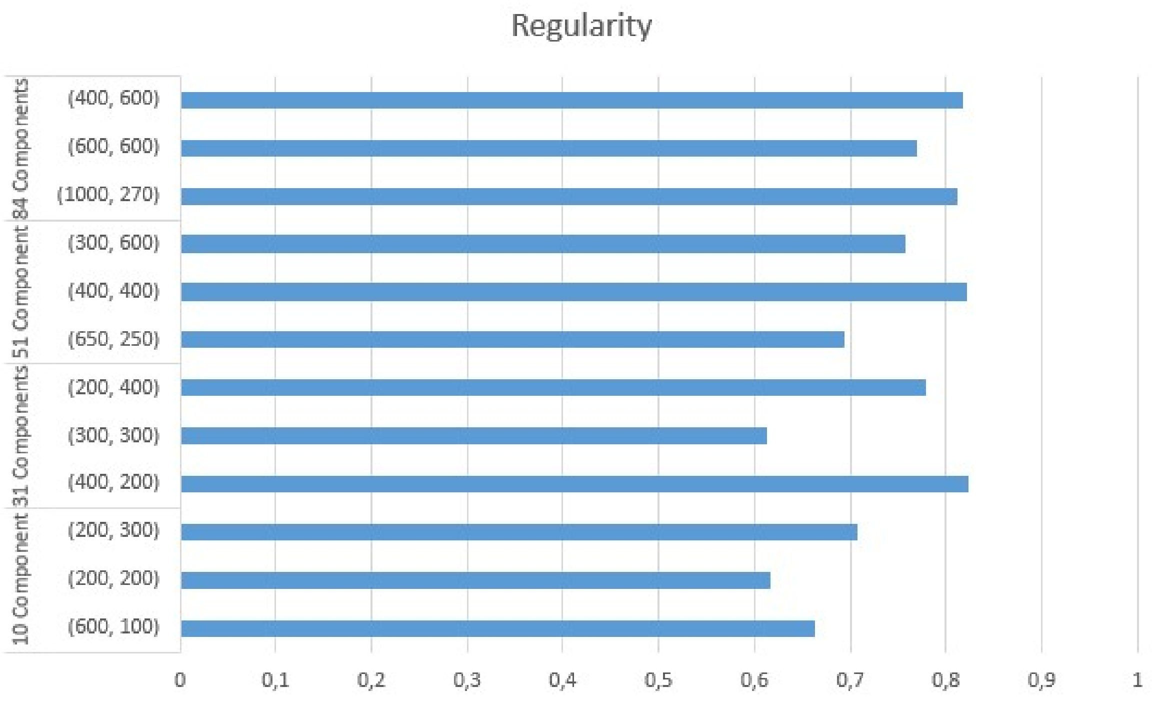


FIGURE 6.30. Regularity parameter results for each resolution.

Regularity is obtained by minimizing both the alignment and spacing differences. This parameter compliments Simplicity and Economy parameters and tries to apply both of them in its' own way. The aim is to take advantage of both Gestalt's Similarity and Continuity principles and create the self-oriented guidance feeling to the user.

6.2.3.9. Economy

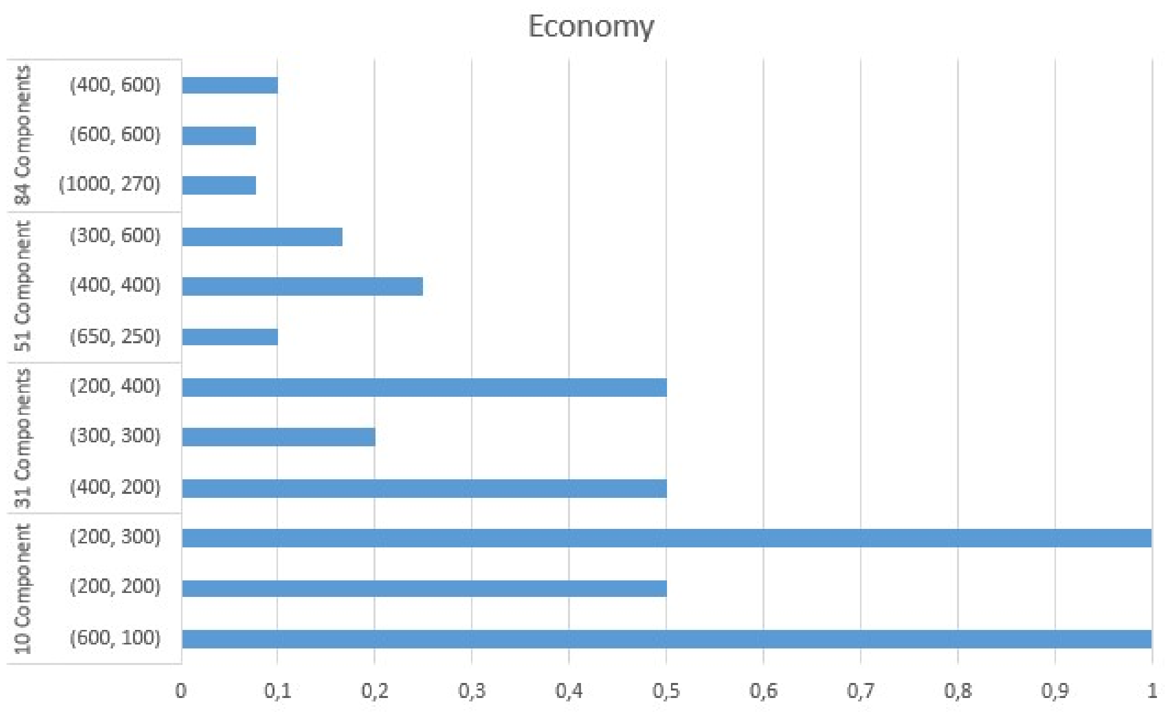


FIGURE 6.31. Economy parameter results for each resolution.

The economy parameter measures how many different dimensions are used in the components, and the more similar dimensions are used by utilizing Gestalt's similarity principle, the higher results are given. As the interfaces grow and become more complex, the values in the graph decrease as it will be difficult to find components of similar sizes. It has also been observed that as the number of components increases, the difference between the values of the same tree at different resolutions decreases.

6.2.3.10. Homogeneity

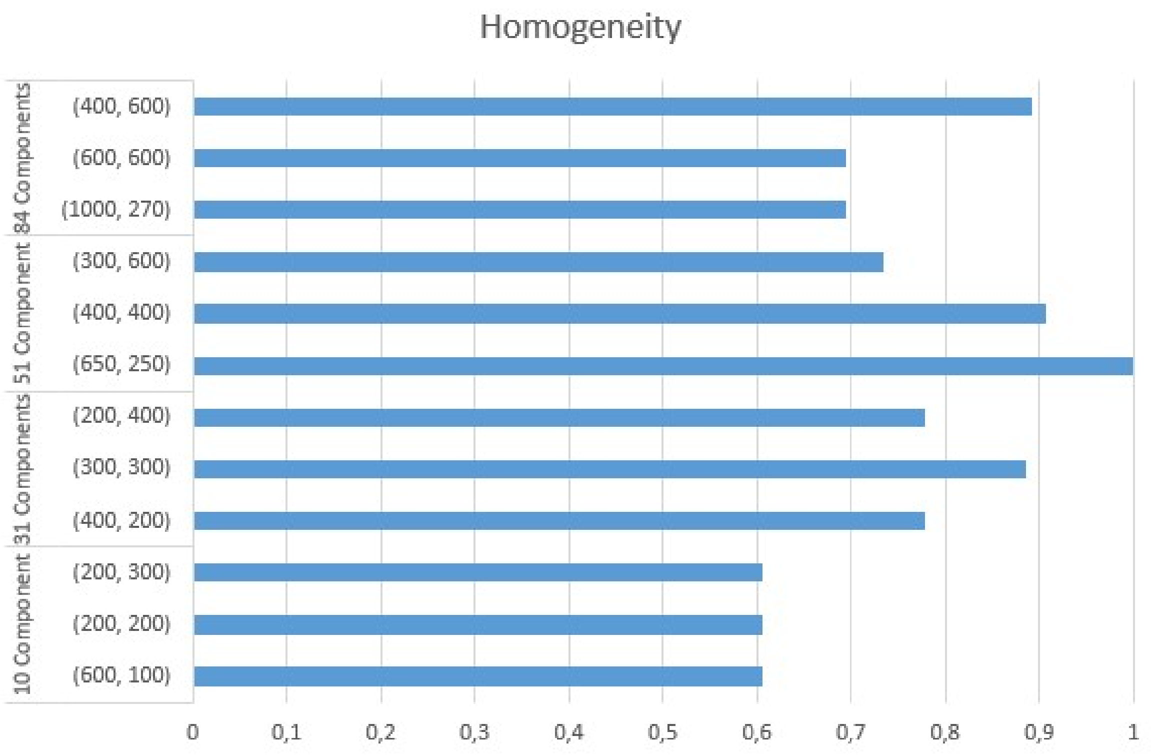


FIGURE 6.32. Homogeneity parameter results for each resolution.

Homogeneity pays attention to how the elements are distributed over the four quadrants. If there are no equal number of elements in each quadrant, homogeneity is not achieved. When the imaginary quadrant lines are drawn, it is seen that the more equal distribution there is within these lines, the higher the scores. However, as in Figure 6.7., this parameter has lower values in interfaces that are grouped unbalanced because there are more alignment points.

6.2.3.11. Order and Complexity

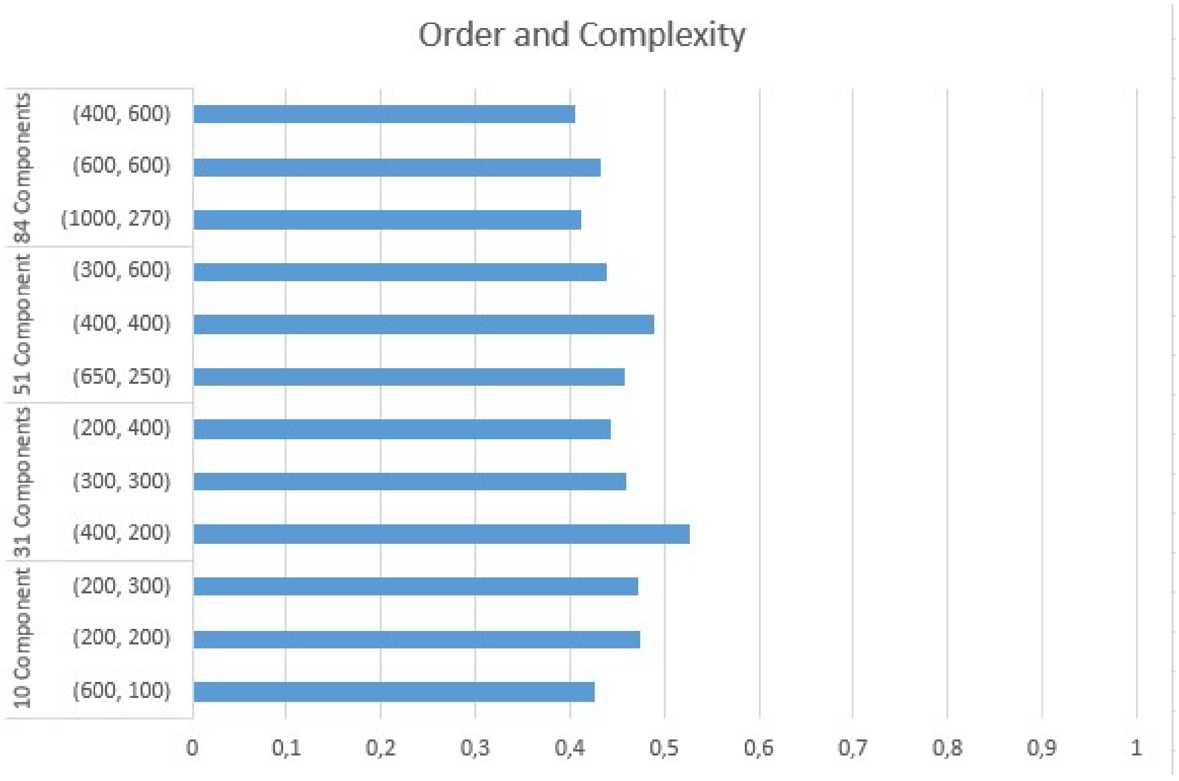


FIGURE 6.33. Order and Complexity parameter results for each resolution.

Since Order and Complexity is the average of all these parameters, it creates another layer to eliminate complex interfaces. Other parameters aim to eliminate complexity and irregularities so this parameter can be seen as a support parameter to the others.

7. CONCLUSION

7.1. Concluding Remarks

Ensuring that the interfaces, which play the most important role in the interaction between human and machine, are automatically presented to people in the most proper way has been one of the most valuable skills of today. Although automating interfaces has been tried by many studies and institutions, they all require human support. What we are trying to achieve is to remove the human factor and present the given information in the most automatic and effective way possible.

Shopping sites, social media, communication platforms, games, the common point of all these interaction tools are that they have an interface and that this interface can be displayed on phones, computers, kiosks or giant screens. Normally, these interfaces, which are prepared separately for each platform shown, are exactly the same, and the only difference being the presentation shows us a great loss of effort. In order to prevent this loss of effort, we established a geometric and topological structure between the elements in the interface, and using this structure, we developed an algorithm that will select the most preferable presentation for each platform, even, for each resolution.

Similar to our study, the fact that we found few results when we searched for other studies in the literature shows the originality of our study. Only Zeidler and Ngo have succeeded in doing a study close to what we aimed for, and these studies are only a small part of our scope and only a measurement tool, although they do not exactly solve the problem we want to solve. We had a hard time finding similar studies both in the early stages of the study, in the article we published in the middle of the study, and in the literature review we made afterwards.

Our main technique in doing this was to extract a topological relationship tree and find feasible layouts by pruning on this tree. But finding feasible layouts is only half of the job, and these layouts are actually nothing but guidelines with infinite arrays of possibilities. By trying methods to adjust and balance the arrays of these infinitely possible layouts, we finally turned all these layouts into realizable data with a balanced distribution method. At this point, we could now implement any interface with the chosen layout, but a method was needed to make the best choice. The mathematical representations of the aesthetic values we found in our literature review attracted our attention and we wanted to make something abstract concrete. That's why we left the final decision to aesthetic evaluations and subjected the layouts to a final test. The interface with the highest score from the test was selected as the most suitable interface for that resolution to be displayed and drawn on the page to be shown to the users.

In our study, we tried 4 layouts and we subjected each layout to 3 different resolutions. Each layout was tested with different resolution values such as wide, square and long values, and the results were noted. Our aim was to make the automatic layouts usable in daily life, so real-time adaptation had to be provided and that could only be achieved with results that are under a second. In order to combine our sample layouts on a common denominator, we chose the same interface and used only different component numbers, and to add realism to the work, we chose a part of a real-world visa application form. First layout was composed of 48 UI components and 36 container components, second layout was composed of 28 UI components and 23 container components, third layout was composed of 17 UI components and 14 container components, and lastly the fourth layout was composed of 6 UI components and 4 container components. Each tree was tested at wide, square and long resolutions and all values were logged during the whole process. While navigating the topological relationship tree, we saw that a lot of time was lost, especially in large interfaces, and with the memoization method, we got rid of all the calculations that were calculated over and over and gave the same results, and we obtained feasible layouts in less than a millisecond. During the distribution phase, we saw how choosing a balanced distribution that creates the least visual errors helped us apply Gestalt principles. We achieved handcrafted-like results on interfaces selected by aesthetic scoring. We have seen that aesthetic formulas[22][23] overlap with Gestalt principles[21] and make choices according to these principles. Even in an interface with 84 components, it took a maximum of 10 ms for us to only find feasible layouts and make them ready for drawing by distributing all of these layouts according to the given resolution, without coming to aesthetic scoring. This was a result that exceeded our expectations. Despite heavy and expensive aesthetic formulas, we were able to test thousands of feasible layouts in less than a second and draw the interface with the best results. We managed to esthetically score all 6225 feasible layouts of our tree with 84 components in 581.954 ms, which is the biggest experiment we have done. The slowest drawing of a layout we could achieve in these experiments took 0.582 seconds which fulfills our goal of drawing layouts under a second.

Since the experiments are written in Java programming language, the variety of interface elements in the examples depends on the capabilities of the swing library. Although we want to measure Memory and CPU values in experiments, other asynchronous tasks of the JVM prevent these measurements from giving stable results. Memory and CPU samples taken repeatedly in the experiments are not included in the results section of the experiment, as they give different results each time. We used JProfiler 10.1.5 to take the samples but with each snapshot, the less sense the data made. As long as the intervals in the given topological tree do not exceed 300 pixels, or a tree larger than 100 components is not used, our study can analyze between 7000 to 8000 max feasible layouts and score them aesthetically. If we get rid of the dependencies created by the Java language, we can work with much larger numbers.

7.2. Future Work

For now, our work only processes areas and fills the screen with no space left. Spaces can be added between components and scroll bar logic can be implemented. While implementing scroll bars and spacers, an error margin can be considered as a decision maker parameter. The area that overflows the screen, and the shape and proportion of the overflowing area can be used to prioritize which component gets a spacer or a scroll bar.

If we include different shapes and colors besides the fields, we can obtain a different aesthetic scoring system. Apart from these, the weights of all parameters are currently considered equally in aesthetic scoring, and these weights may give different results at different values. Optimizing parameter weights can be a work in itself.

While distributing feasible layouts according to resolution, a single distribution method is used. Different distribution methods can be added or it can be preferred to use these methods in different places at the same time. Also with weighted distribution, we can provide a logic which determines when to use maximum or minimum values to distribute or even come up with a custom weight calculation formula. Depending on the situation, much more accurate results can be obtained with different distribution methods. Which brings us to the next future work, which is to look at the interaction between branch nodes. We consider the interaction within the component groups, but if we change the distribution system by looking at the interaction between the branch nodes while distributing, we can get a system that sees the bigger picture. Also in a future work, component groups can be highlighted more and give a feeling of belonging. For the time being, our results emphasize on groups but does not highlight it.

REFERENCES

[1] Ali Roudaki, Jun Kong, and Nan Yu. A classification of web browsing on mobile devices. *Journal of Visual Languages & Computing*, 26:82–98, **2015**.

[2] Jean-Sébastien Sottet, Gaëlle Calvary, and Jean-Marie Favre. Models at runtime for sustaining user interface plasticity. In *Models@ run. time workshop (in conjunction with MoDELS/UML 2006 conference)*. **2006**.

[3] Krzysztof Gajos and Daniel S Weld. Supple: automatically generating user interfaces. In *Proceedings of the 9th international conference on Intelligent user interfaces*, pages 93–100. **2004**.

[4] Dirk Roscher, Grzegorz Lehmann, Veit Schwartze, Marco Blumendorf, and Sahin Albayrak. Dynamic distribution and layouting of model-based user interfaces in smart environments. In *Model-Driven Development of Advanced User Interfaces*, pages 171–197. Springer, **2011**.

[5] Greg J Badros, Alan Borning, and Peter J Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, **2001**.

[6] Hiroshi Hosobe. A scalable linear constraint solver for user interface construction. In *International Conference on Principles and Practice of Constraint Programming*, pages 218–233. Springer, **2000**.

[7] Hiroshi Hosobe. A simplex-based scalable linear constraint solver for user interface applications. In *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, pages 793–798. IEEE, **2011**.

[8] Noreen Jamil, Deanna Needell, Johannes Muller, Christof Lutteroth, and Gerald Weber. Kaczmarz algorithm with soft constraints for user interface layout. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 818–824. IEEE, **2013**.

[9] Noreen Jamil. Constraint solvers for user interface layout. *arXiv preprint arXiv:1401.1031*, **2014**.

[10] Noreen Jamil, Johannes Müller, M Asif Naeem, Christof Lutteroth, and Gerald Weber. Extending linear relaxation for non-square matrices and soft constraints. *Journal of Computational and Applied Mathematics*, 308:346–360, **2016**.

[11] Alan Borning, Richard Lin, and Kim Marriott. Constraints for the web. In *Proceedings of the fifth ACM international conference on Multimedia*, pages 173–182. **1997**.

[12] Charles Jacobs, Wilmot Li, Evan Schrier, David Bargeron, and David Salesin. Adaptive grid-based document layout. *ACM transactions on graphics (TOG)*, 22(3):838–847, **2003**.

[13] Jakob Nielsen. Usability inspection methods. In *Conference companion on Human factors in computing systems*, pages 413–414. **1994**.

[14] Yue Jiang, Ruofei Du, Christof Lutteroth, and Wolfgang Stuerzlinger. Orc layout: Adaptive gui layout with or-constraints. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12. **2019**.

[15] Clemens Zeidler, Christof Lutteroth, and Gerald Weber. Constraint solving for beautiful user interfaces: how solving strategies support layout aesthetics. In *Proceedings of the 13th International Conference of the NZ Chapter of the ACM's Special Interest Group on Human-Computer Interaction*, pages 72–79. **2012**.

[16] Christof Lutteroth, Robert Strandh, and Gerald Weber. Domain specific high-level constraints for user interface layout. *Constraints*, 13(3):307–342, **2008**.

[17] Clemens Zeidler, Christof Lutteroth, Wolfgang Sturzlinger, and Gerald Weber. The auckland layout editor: An improved gui layout specification process. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 343–352. **2013**.

[18] Clemens Zeidler, Gerald Weber, Wolfgang Stuerzlinger, and Christof Lutteroth. Automatic generation of user interface layouts for alternative screen orientations. In *IFIP Conference on Human-Computer Interaction*, pages 13–35. Springer, **2017**.

[19] M Pajusalu, R Torres, and D Lamas. The evaluation of user interface aesthetics. *Masters), Tallinn University*, **2012**.

[20] Patrick Mbenza Buanga. Automated evaluation of graphical user interface metrics. *Universite Catholique de Louvain, Master Thesis*, **2011**.

[21] Dejan Todorovic. Gestalt principles. *Scholarpedia*, 3(12):5345, **2008**.

[22] David Chek Ling Ngo, Lian Seng Teo, and John G Byrne. A mathematical theory of interface aesthetics. In *Visual mathematics*, volume 2. Mathematical Institute SASA, **2000**.

[23] David Chek Ling Ngo and John G Byrne. Another look at a model for evaluating interface aesthetics. *International Journal of Applied Mathematics and Computer Science*, 11:515–535, **2001**.

[24] Mathieu Zen and Jean Vanderdonckt. Assessing user interface aesthetics based on the inter-subjectivity of judgment. In *Proceedings of the 30th International BCS Human Computer Interaction Conference: Fusion!*, page 25. BCS Learning & Development Ltd., **2016**.

[25] John Zukowski. *Java AWT reference*. O'Reilly Media, **1997**.

[26] Shihui Han, Glyn W Humphreys, and Lin Chen. Uniform connectedness and classical gestalt principles of perceptual grouping. *Perception & psychophysics*, 61(4):661–674, **1999**.