Responsive GUI framework in Lua

Vlad Alexandru Draghici
Abstract

Responsive GUI framework in Lua

Vlad Alexandru Draghici

Responsive graphical user interfaces, which can adapt to changes in display size, improve the user experience and are easier to develop and maintain compared to ones relying on absolute (pixel) placement. Based on a survey of several responsive frameworks across desktop, mobile and web platforms, we design and develop a backwards-compatible responsive extension of an existing non-responsive Lua GUI framework. We also provide the specification that describes the features selected to be part of the new responsive framework, as well as some implementation details pertaining to those features.
Acknowledgements

I would like to thank the university coordinator Justin Pearson for assisting me with administrative matters throughout the whole duration of my thesis work, my university reviewer Johannes Borgström for helping me with drafting the report and giving me valuable insights, the developer supervisor Fadi Bitar and the team at XLN Audio for assisting in the practical portion of my thesis and for creating a wonderful work environment for me, and my parents and family for always supporting me.
## Contents

1 Introduction .......................... 1
   1.1 The problem .................................. 1
   1.2 Report structure ............................. 1
   1.3 General description .......................... 2
   1.4 Goals ........................................ 2
   1.5 Current technologies and architecture ............... 3
   1.6 Delimitations ............................... 4

2 Background .................................. 5

3 Pre-implementation .......................... 7
   3.1 Existing responsive GUI frameworks and technologies .... 7
      3.1.1 CSS Flexbox ................................ 7
      3.1.2 CSS Grid .................................. 9
      3.1.3 CSS Media Queries ........................... 10
      3.1.4 Android ................................... 11
      3.1.5 iOS ....................................... 12
      3.1.6 Qt ......................................... 13
      3.1.7 wxWidgets ................................ 14
      3.1.8 Others .................................... 14
   3.2 Comparison .................................. 14

4 Method of choosing features .................. 16

5 Specification and implementation ............ 18
   5.1 Containers .................................... 18
   5.2 Relative positioning .......................... 21
   5.3 Style files ................................... 23
   5.4 Properties based on device size .................. 24
   5.5 Testing ....................................... 25

6 Conclusion ................................... 26

7 Future work ................................ 27
1 Introduction

1.1 The problem
Responsive graphical user interfaces can adapt to changes in the shape and size of a device’s screen. They also enable easier maintenance and development of products by removing some of the need to manually specify most interface elements’ sizes and positions. Developing non-responsive interfaces, by contrast, can be more tedious than necessary, because of the need to specify each UI element’s position and size. In addition, such interfaces usually do not adapt well to screen configurations for which they have not been tested.

1.2 Report structure
The report structure is as follows:

- The introduction (section 1) describes the problem, goals, delimitations, and background of the project.
- Section 2 provides a background into the problem described by the report.
- In the pre-implementation (section 3) part, the research phase of the project is described, containing survey of existing responsive GUI frameworks. In addition, the feature specification for the implemented solution is presented here.
- The implementation, including various algorithms used, is described in section 5.
- The conclusion (section 6) analyzes the end product and tries to determine if the main goals of the thesis have been achieved.
- Future work (section 7) proposes possible improvements to the existing implementation, including features that have been left out during the current project.
1.3 General description

The project takes place at the Swedish music software development company called XLN Audio, which develops music production plugins used in digital audio workstations (DAWs). A DAW is an application allowing users to create, modify and record audio, including music. Most DAWs allow using music or audio plugins, which allow for different types of enhancement, manipulation of sound, including for specific musical instruments. XLN Audio develops such music plugins with the help of a technology stack that includes an upper layer developed in Lua (see section 1.5). This layer includes, among other things, a GUI framework for specifying the interface of the plugins. The goal of the thesis is to expand upon this GUI framework, adding responsive functionality to it.

1.4 Goals

The goals of this master’s thesis are to extend the current Lua GUI framework with responsive functionality and to provide other features for easier application development and maintenance. This will allow developers to assign general behaviours to a UI element or group of elements, to position and dimension elements in relation to other UI elements without the need for coordinates, to separate some UI element property declarations from the main layout files (similar to how HTML and CSS operate in that regard), and to specify different behaviours for different screen sizes. This will not only ease the development of products by eliminating the need for detailed manual positioning and size of elements, but will also add a responsive nature to the products, which will benefit end users.

The most important questions that I have tried to answer during this project are the following:

- Which existing responsive features in other GUI frameworks would be relevant to XLN Audio’s music plugins and which wouldn’t be?
- What are the most significant features to add in a limited-scope project such as this one?
1.5 Current technologies and architecture

XLN Audio’s music plugins are developed using a technology stack with three main layers, with the actual application logic of each product making up the fourth and final layer:

- The lowest C++ layer consists of the third-party C++ application development framework JUCE, used mainly in the creation of standalone or plugin music production software. JUCE allows developers to target multiple platforms (in our current case, desktop ones - Windows and macOS) with not much modification to the code required from the application developer. Audio plugins communicate with DAWs through protocols such as AAX, AU and VST. The JUCE framework provides wrappers around these protocols to make plugin development easier. Other open-source libraries such as Boost, a C++ library containing a wide variety of features such as file system management and math functions, is also used.

- An intermediary, proprietary C++ layer consisting of a framework developed by the company on top of JUCE, nicknamed ”Cotton”. This layer includes functionality like low-level graphics, digital signal processing (DSP) and handling the file system. In addition, Cotton is designed to be as generic as possible and acts as a Lua interpreter for all the products developed by the company.

- A higher layer written in the Lua programming language, which includes the preset system and the GUI framework. The preset system allows saving user parameters and preferences and is designed to work with different preset lists from different products. The GUI framework is a set of components and functionality that enables developers to specify a product’s interface in the form of Lua tables. This layer calls a cotton API that handles the drawing of the specified UI elements.

- The final layer is the actual interface specification and application logic for a specific product, written in Lua.

This architectural division enables rapid prototyping of the company’s software products, since it allows changing most of the UI and logic of the application in Lua with ease (since it is an interpreted language), without the need to recompile the whole C++ platform. In addition, it enables the software
design team to conceive and experiment with a future product’s GUI with minimal involvement from the software developer team.

1.6 Delimitations

The thesis project only involves creating the responsive framework for specifying the GUI of the company’s software products, not actually creating any GUI or audio product. User studies are also out of scope. While the initial scope of the project stated that there would be some tangential work done on the lower-layer C++ platform, which would involve coding in that area only if strictly and directly related to the subject of the thesis, this hasn’t been a necessity at all.
2 Background

Modern web, mobile and desktop application development has changed considerably over the last few years, requiring interfaces to adapt to the wide variety of device screen sizes and shapes[1]. Considering the increased screen sizes and high-resolution screens increasingly used by professionals, software music plugins have the potential to benefit from a responsive GUI. In addition, an easier way to specify UI in products could result in a faster delivery of products to customers, as well as better subsequent maintenance.

A responsive GUI is one that adapts to different device and screen sizes and allows an easy way to rearrange UI elements when a user resizes the window. In addition, when the content inside a page changes dynamically at run-time, it performs a recalculation of the positions of some elements on the screen. This helps to keep an optimal and user-friendly positioning of UI elements regardless of screen size and helps prevent things like horizontal scrolling.

One more advantage of responsive GUIs is that they speed up development and maintenance[2]. Currently with some static interfaces in general, most UI elements are given a position by manually specifying absolute coordinates. This is also the case with the current practice of product interface specification inside XLN Audio’s products. This makes the creation, debugging and update of interfaces with this type of specification difficult and slow. If a UI element needs to have its position changed, an exhaustive and cascading manual verification needs to be performed for many interface elements that might be positioned with the previous UI element in mind. Afterwards, manual recalculation of the coordinates needs to be performed for UI element when necessary. This slows development and renders debugging and updating a UI tedious. Responsive and dynamic capabilities provide algorithms that greatly mitigate this problem by eliminating the need for most manual absolute positioning of elements. One example would be a dynamic container that automatically positions and aligns its children elements in a row, eliminating the need to manually declare the position of each row element one after another.

One of the sectors with most improvement in the area of responsive interface design has been web development. Here, different device sizes and user resizing of browser windows have both brought the need for a way to handle changes in web page size and shape. Current Web and other UI professionals generally use two distinct but related terms when referring to modern (Web)
user interface design[3]:

- Adaptive GUIs simply involve specifying multiple separate interfaces for each supported screen size range. This is the optimal solution when it comes to user experience and quality, since it allows each version of the interface to be tailored and finely adapted to its corresponding screen size and shape. This comes at the disadvantage of longer development times and more difficult maintenance and support, since it involves multiple interface versions instead of just one. As new device sizes are introduced to the market, an adaptive interface requires constant modification of the code to adapt the interface to the newer devices.

- Responsive GUIs involve having one responsive design for all or most screen sizes. With the use of CSS media queries (a feature that allows setting properties for specific device sizes), developers can specify a single interface version that changes its properties according to changes in the window size and shape. In addition, dynamic containers that automatically resize and reposition the elements inside them according to specific given parameters and outside interactions are also used.

Mobile platforms have also seen the introduction of responsive functionality. While most mobile apps are not generally resizeable by users, different screen sizes and shapes require similar features that are present in some responsive Web development frameworks.

In the area of desktop software applications, developers have started using both traditional desktop development frameworks updated with new responsive functionality and hybrid application frameworks (such as Electron and other Chromium-based frameworks, used for applications Atom[4], Slack[5], VS Code[6], Spotify[7]) that use web views and Web technologies inside wrappers to create more responsive desktop applications.
3 Pre-implementation

The goal of the project is to design and implement a set of responsive features for the Lua GUI framework, optimal for the needs of the company’s products and developers. To achieve this, I have conducted a study of responsive features in other existing frameworks, as presented in this section. The study includes GUI frameworks and toolkits across mobile, desktop, and web platforms. These technologies/frameworks are Cascading Style Sheets (CSS) Flexbox and CSS Grids used in web development, responsive features that exist in iOS and Android mobile development, and the Qt and wxWidgets GUI toolkits used for desktop application development.

I have performed a comparison between each approach, looking at both their detailed feature specifications and their benefits and drawbacks in relation to the needs of the company. Studying a wider variety of GUI frameworks ensures knowledge about as many features as possible, allowing for a more informed decision about which of these features would be good for making XLN Audio’s music plugins, as well as realizing which desirable possible features that those platforms lack would be good for the type of products that the company is developing.

3.1 Existing responsive GUI frameworks and technologies

3.1.1 CSS Flexbox

Flexbox is a layout module used in Cascading Style Sheets (CSS). It is a one-dimensional model in which a container is used to handle different properties of its child UI elements automatically. The one dimension can be set to horizontal or vertical, meaning that flexbox can create both rows and columns of elements. When nesting flex containers inside other containers of different orientation, two-dimensional structures (similar to tables) can be created. The main features in the latest specification of Flexbox[8] are:

- Setting the orientation and direction of elements inside the container.
- Row wrapping: starting a new column/row if the current cannot fit all elements (overflowing).
• Aligning the content along the main axis, with options (figure 1) for managing the empty space between items: start, end, center, space-between, space-around, space-evenly. For example, space-evenly distributes the empty space evenly between container elements and between marginal elements and the container’s edges.

• Alignment of items along secondary axis: start, end, center, stretch, baseline.

• Handling the space between multiple rows/columns of elements: start, end, center, stretch, space-between, space-around.

• Item ordering, which defines the order in which the UI items appear on screen, other than the one defined by the order in which they are declared in HTML.

• Ability of items to grow to take up remaining empty space or to shrink, if required.

• Single-item secondary axis alignment, which is the same as the one described earlier but only applied to a single item instead of all.
3.1.2 CSS Grid

CSS Grid is a two-dimensional layout system in CSS. A grid container handles the positions and sizes of its children UI elements by placing them in automatic or user-defined areas of the grid. Here is an outline of the most relevant features at the time of writing[9]:

- Declaration of a variable number of rows and columns with different approaches to setting size: fixed pixel count, auto-sizing to fit content, filling a certain fraction of the grid.

- Naming of grid columns, rows and areas (an area being a rectangle consisting of multiple adjacent cells).

- Automatic creation of rows and columns as new content is added.

- Placement of elements in the grid based on grid coordinates or row/column names, spanning one, multiple or all rows/columns (can specify the
• Setting of gaps of variable size between rows and between columns.

• Aligning content in all grid cells along the row or column axis: start, end, center, or stretch (default).

• Aligning the entire created grid content along the row or column axis: start, end, center, stretch, space-around, space-between, space-evenly. This handles the empty space between rows and between columns.

• Aligning a single cell item along the row or column axis: start, end, center, stretch.

• When placing an item in a column/row that doesn’t exist, automatic creation of that column/row as well as intermediary rows/columns between the last previously existing row/column and the newly created row/column. The automatic intermediary rows and columns have size zero.

• Specifying the size of automatic intermediary rows/columns manually.

• Automatic placement of items in the grid when their coordinates are not specified, using several algorithms (fill rows, fill columns, fill more densely populated areas).

• Function for generating a fixed amount of empty rows/columns or until the grid size limit is reached.

3.1.3 CSS Media Queries

Media queries are a CSS feature[10] that allows declaring different element properties for different device types (most often based on screen resolution). A media query contains a media type and one or more expressions. If the device type matches the one declared in the media query and all expressions are evaluated to true, the style rules contained in that media query will be applied to the corresponding element.
3.1.4 Android

Android is a Linux-based operating system used in a variety of devices, including smartphones and tablets. Due to a such variety of screen sizes and shapes, responsive features are necessary and have been implemented. The layout is specified using Android’s XML vocabulary. At the time of writing, along other capabilities, Android has several available containers (RelativeLayout, GridLayout, etc.), each with different features:

- Setting widget size dynamically, dependent on its content or maximum parent size.
- Positioning widgets on the screen according to spatial relationships between child widgets or between a child widget and its container (RelativeLayout[11]):
  - Putting a child above/below or to the right/left of another child.
  - Aligning one of the child’s edges with one of the parent’s edges.
  - Centering a child view within its parent view (horizontal, vertical or both)
- Ability to load different layout files depending on screen size or orientation, similar to CSS media queries, and programmatic loading of different UI components based on screen size.
- LinearLayout[12] (rows/columns): places items in rows or columns. Can specify orientation, weight sum (individual items have weights which specify how much space they occupy along the main axis).
- GridLayout[9]: place items in a grid, specifying spacing, alignment, orientation, row and column span of each item.
- ConstraintLayout[13] - similar to RelativeLayout in terms of relative positioning, but in addition contradicting constraints can be weighted (meaning, two contradicting constraints can have weights other than 50%-50%).
  - Circular positioning (given angle, radius).
  - Hiding of UI elements, but they are still part of the layout in terms of layout computations (albeit with zero size).
- Use of chains (items bound sequentially) in a single axis (column/row) to handle item size and distribution of empty space between items, as well as item weights.
- Additional alignment options, compared to RelativeLayout.
- Guidelines - invisible UI elements that help with the positioning of other elements.
- Barriers - invisible helper UI elements that grow with the specified UI elements. This helps handling views with dynamic content, making the size of the view only known at runtime.

- Use of "gravity" to specify the position of a UI element inside its parent, as well as the content position inside the UI element.

3.1.5 iOS

iOS is a mobile and tablet operating system. As of iOS version 11, it has several features that enable UI responsiveness, the main ones being stack views and a constraint system. It also offers the option of having different layout files depending on factors like window size.

Stack views[14], which arrange a group of views in a row or column, are used for the simpler parts of the layout. They handle the alignment of items across the main axis of the stack view, with options to stretch all items equally to fill up remaining empty space, to stretch them proportionally to their size, to only stretch one item, or to only distribute the empty space equally around the items without changing their initial sizes. Stack views also handle the alignment of items relative to the cross axis of the stack view: top, center, bottom, baseline (which aligns UI elements by the lines of their text). Stack views can be nested, giving the possibility to create two-dimensional layouts.

A system of constraints[15] is used to achieve more complex UI structures and a responsive layout. It consists of declaring some traits such as the position and dimensions of an UI element in terms of the position and dimensions of other child or parent views. The constraints are mathematical equations that need to have a single solution. Ambiguous constraints cause errors. Some of the attributes included in the equations are the four edges, height, width, vertical and horizontal centers, and aspect ratio of the UI element. Constraints can have different priorities and are also divided into
optional and mandatory constraints. In addition, they can also be expressed as inequalities. Some UI elements have intrinsic size based on their content, which can be used inside constraints.

3.1.6 Qt

Qt is a cross-platform application development framework for most desktop, mobile and embedded platforms. Some current features[16] for enabling and responsive interfaces are:

- Anchors, or the ability to position a UI element relative to other UI elements or its parent, using anchor lines (left, right, vertical center, top, bottom, baseline, horizontal center). In addition, margins can be set for each element (top, right, bottom, left). Items can only be anchored to its siblings or parent UI element. Anchors can be dynamically changed at run-time.

- Positioners, which are one- and two-dimensional containers for placing items in rows, columns, grids or flows. Flows are similar to grids, but items in a flow will automatically start a new line when running out of space (row wrapping), and items do not necessarily align. Positioners also include properties for orientation, direction, spacing between items, padding, etc.

- Layouts (grid, row, column) are like positioners, but can also resize the items inside them. Each item has the option to fill the height/width of the container, as well as have minimum, preferred and maximum height/width. An item’s grid coordinates can be declared, as well as row span or column span (meaning, how many rows or columns are occupied by the UI element, starting from the coordinates). Items can also be inserted without specified coordinates, being places automatically based on the declared flow.

- Property bindings[17]: relations between different properties of UI elements. When the dependencies change, the property is updated automatically.
3.1.7 wxWidgets

wxWidgets (as of version 3.1[18]) is a cross-platform GUI toolkit that also has a few features related to responsiveness. One of these features is the box sizer, which is a one-dimensional container that can be nested inside other box sizers. It has weights that regulate item size along the main axis, or how much an item can grow beyond its initial size in relation to other items, as well as item margins (left, right, bottom, top) and alignment.

In addition, it includes two-dimensional containers like the grid sizes and the flex grid sizer. The grid sizer lays UI elements in a 2D table, in which cells have the same size, the number of rows and columns as well as the vertical and horizontal spacing between cells can be specified. The flex grid sizer is similar to the grid sizer. The difference is that while it requires the same height for all items in a row and the same width for all items in a column, there doesn’t have to be a single size for all items in the grid.

3.1.8 Others

There are many other application frameworks that haven’t been included, most of them being on the desktop platform. These include frameworks like JUCE (which is being used for the lower-level layer of the music plugins but not their GUI, as detailed in section 1.5), which has implemented the CSS Flexbox specification. There are many hybrid desktop application frameworks that use the same technologies from Web development (HTML, CSS) for their GUI development, making an analysis redundant. In addition, there are some lesser utilized desktop and mobile frameworks and platforms that haven’t been studied in more depth.

3.2 Comparison

During the study of different responsive frameworks and platforms, I’ve come across a variety of features that could be grouped into the following three general categories:

- Relative positioning, which is the placement and sizing of UI elements according to other elements, like aligning an UI view’s edge to another view’s edge or setting a particular view’s size as a percentage of another view’s size.
• Containers, which are UI elements that automatically set the positions and sizes of their child UI elements. They can be either one-dimensional rows and columns or two-dimensional grids. They handle multiple aspects like the orientation of the flow of elements, element sizes, positions, and alignment.

• Layout files or styles based on device size: this includes the possibility of declaring specific UI element properties or even entire separate layouts for different screen versions.

Relative positioning generally allows for more low-level control of item positioning than containers do, as it provides an easy way to set the location of a specific item. This gives the possibility of creating highly customized GUI layouts that are also adaptable to changes in screen size automatically, since UI elements are positioned according to other elements and are not given absolute coordinates. On the other hand, using relative positioning features can be unnecessarily complicated for simpler layouts where containers usually suffice. In addition, maintaining and developing layouts using relative positioning can be more difficult than with containers: the developer has to keep track of all relationships between elements and changing the layout by deleting or modifying a view can cascade to other views.

Containers allow a much easier declaration of UIs with relatively simple layouts, where structures like rows, columns, and features like item alignment are enough to satisfy the design needs. They can also be dynamic, adapting to screen changes by rearranging and resizing their child elements. Containers also provide two-dimensional capabilities through the use of grid containers or nesting of line containers. Layouts using containers are easier to maintain than those using relative positioning because there is no need to set any additional properties regarding size or other parameters to individual elements due to the fact that this functionality is handled by the container for all its child elements. On the other hand, making complex UIs with more detailed customization is harder with containers because there are few tools available to affect an individual element inside a container.

Size-based layouts/styles can differ a lot in their scope and implementation, from having separate layouts altogether to just assigning some properties separately for some elements according to screen size. The creation of separate UI layouts for different sizes is part of adaptive GUIs, as opposed to responsive ones (as detailed in section 2), while just having some different size-dependent properties for the same layout (like CSS Media Queries) can
be used in conjunction with other features, when relative positioning and containers are not enough to adapt to different sizes.

When it comes to the difference between one- and two-dimensional containers, we can compare CSS Flexbox and CSS Grid. The advantages of using Flexbox are its simplicity and the easier alignment and rearrangement of items in a particular row or column. With CSS Grid, it is easier to specify uniform gaps between columns and rows (the grid gap). It is also easier to position elements in two dimensions by having the possibility to assign both the row and the column where an item is to be placed, while cross-axis manipulation in one-dimensional containers is limited to alignment on a single line/column. The grid can also more easily align items across multiple rows/columns while having more limited ability to control items in a particular row/column. Thus, grids are better for constructing a template of an entire web page by assigning different areas in the grid to specific web page components like the header, footer and main body.

Since many features are shared between platforms like iOS, Android, and Qt when it comes to relative positioning or containers, it is hard to make an objective evaluation of which set of features is better in a particular situation. The differences in semantics between the features of those different platforms come down to design decisions made by their creators.

4 Method of choosing features

During the design stage I compiled existing and new techniques into a single specification that achieve the goals of creating a responsive framework and ensuring backwards compatibility with the old version of the GUI framework. Ensuring backward compatibility is an important requirement for XLN Audio, as it will also provide developers at the company the possibility to upgrade existing products to the newer framework version. In addition, this stage involved research into the functioning of the current Lua framework and required developing a deep and detailed understanding of all its components, like specific UI widgets used across products, existing systems that might help me when coding the new features, as well as the general work-flow of the code and life cycle of UI elements.

When choosing the features that I’ve implemented in the GUI framework, it was hard to give a scientific basis to these choices. Being a very limited time project, I had to prioritize those features that would be the most useful
to the company’s products. The features were selected through participation in workshops with developers and software designers at the company by analyzing which features would have the biggest impact on both existing future products. One important aspect was to determine if existing products have specific UIs or features that could become more maintainable or more responsive if converted to the new framework, as well as to see if existing UI widgets could be improved by incorporating the new responsive functionality into them or replaced altogether. In addition, we studied specific use cases of these new features for products that might be implemented in the future.

Deciding the semantic details of the features came down to taking inspiration from other frameworks while at the same time making sure that the created feature specification is coherent and the features are compatible with the existing Lua framework. The choice for a particular semantic over another sometimes came from the technical properties of the Lua framework, by deciding which choice would be both easier to implement and better for the framework from an architectural or performance standpoint.

The implementation stage was the phase of the thesis project where I’ve created the actual responsive GUI framework, based on the feature specification. The selected features were one-dimensional (row/column) containers, a system of relative positioning, and a system for separating properties into CSS-like stylesheets with media query functionality. Section 5 presents the feature specification along with some implementation details and demonstrations.
5 Specification and implementation

5.1 Containers

A dynamic container that can have multiple child views has been implemented. It can be either a row or a column. This container places child elements sequentially in an horizontal/vertical manner inside the container, according to the type of the container. Containers can have both fixed and adaptive sizes.

The container can have either a fixed or adaptive (default) size along the main axis:

- A fixed size specifies the exact size in pixels the container will fill along the main axis and is not influenced by the items inside the container.

- The “adaptive” value given to the size attribute signifies that the container will adapt to encompass all child views, up to a maximum size. If the maximum size is reached, the container will no longer grow and new items will be either hidden or be inserted into a new row/column (wrapping). If there are no elements inside the container, its size along this axis will be zero.

The container can have a fixed or adaptive (default) size along the secondary axis:

- The “adaptive” version will mean that the container will have the secondary axis size equal to the largest on the secondary axis container item.

- The fixed secondary axis size will specify the exact size that the container will fill. The container will not adapt if a child item overflows the container along the secondary axis, but the UI item will still be drawn.

For adaptive main axis size containers, there is the optional specification of minimum and maximum size. When reaching maximum size, the container will behave as a fixed size one in terms of wrapping elements, but other functionality that fixed size containers have will not be available.

Containers have dynamic size, meaning that they watch for changes in size for any of their children elements (which can also be other nested containers)
and recalculate child element positions and sizes accordingly, in addition to modifying their own sizes.

Container items can have margins, which are empty spaces around the items that are counted as being part of the item for positioning and sizing purposes. Margins can be set both container-wide and for an individual element inside the container. The different margin properties are top, right, bottom, left, as well as a shorthand variant for setting all margins simultaneously.

For fixed main axis size containers, we can specify the alignment of items across the main axis (start (default), center, end, space-between, space-around). This is the handling of empty space left after all the items have been placed:

- Start: Empty space is located at the end of the container (as determined by the type attribute, either the end of the row or bottom of a column).
- End: Empty space is located at the beginning of the container.
- Center: Empty space is divided equally between the start and end of the container.
- Space-between: Empty space is distributed equally between every two container items.
- Space-around: Empty space is distributed equally between every two container items as well as between the first item’s start edge and the parent container’s start edge and the last item’s end edge and the parent container’s end edge.

The reason that containers with adaptive main axis size cannot have alignment is that they by definition take up as much space as needed to contain elements, with no empty space created.

The items in a container can also be aligned according to the secondary axis. The options are start, center, end and stretch and can be both container-wide and specific to a single UI item.

For each container item, we specify base absolute size or percentage on the main axis. Base absolute size will be the size of the space that the item will occupy along the main axis, before any growing takes place. This is mandatory for adaptive main axis sized containers, since you cannot declare an item’s size as a percentage of its parent’s size if that size is adaptive.
and not fixed. For fixed main axis size containers, there is the alternative of giving percentage values to the item’s main axis size. This will specify the percentage of the total container size along the main axis that a specific item will occupy. Items have the ability to grow. For fixed main axis size containers, individual items can have a property “grow” which specifies if and by how much can the item expand to fill the remaining empty space in the container, after all items have been placed. This value can be any number and the item’s growth share will be its value divided by the sum of all growth values of items inside a single row or column, so items that have been sent to another row or column (wrapping) will not count towards this calculation.

Containers can wrap elements, which means starting a new row or column adjacent along the secondary axis to the previous one, according to the direction of the container. This happens if the container doesn’t have enough space to fit everything in a line or a column. For adaptive main axis size containers, wrap happens when the container reaches the end of the window or its maximum size. For fixed main axis size containers, the first case above is valid, and in addition wrap happens when the items’ combined sizes along the main axis are greater than the container’s declared size.

Containers can have a row/column gap: it specifies in pixels the width of the gap between rows/columns of a container when it is wrapped at least once.

There is also the functionality to hide overflowing items. When items exceed a particular amount that would normally trigger wrapping, there is the option to replace that behaviour with element hiding. All elements after a certain point will be hidden from view and other functionality like item size growing or alignment will not take them into account. If the container increases in size, it will verify if one or more hidden elements fit after this change and display them if appropriate.

Containers have been integrated into some of the previously existing features of the Lua framework, like being able to be placed into various UI elements like scroll lists and being able to dynamically add or remove UI elements at run-time, with the proper computations performed to ensure that all the features described above are working correctly, according to the container’s given parameters.

In broad terms, the algorithm for accomplishing the features described above includes several computational steps. First, the entire UI children list is traversed, calculating their preliminary positions and sizes before all other
parameters (like growing or alignment) are taken into account. The items are then assigned to their rows/columns (wrapping can create multiple lines in a container). For each row, growth factors and alignment options are computed and preliminary item size and positions are adjusted accordingly, and then the items are resized and repositioned according to those final values. If an item changes its size dynamically at run-time, the parent container is notified and computations are performed for the row/column that contains that element, and a check is also performed to see if the overall structure or number of lines has changed and recalculations done accordingly. In addition, if the size of the entire container changes, the whole algorithm is run again.

5.2 Relative positioning

I have also implemented a relative positioning system for placing UI elements without the use of containers or absolute fixed positions. This system includes a set of properties for aligning a UI element’s edge to another’s, which we will call constraints. In addition, we can set the size of an element as a percentage size and to set margins similar to those in containers.

An item has four edges: top, right, bottom and left. The relative positioning system contains the following attributes that define constraints:

- left_toLeftOf
- left_toRightOf
- right_toLeftOf
- right_toRightOf
- top_toTopOf
- top_toBottomOf
- bottom_toTopOf
- bottom_toBottomOf

For example, the first attribute on the list aligns an item’s left edge to another item’s left edge, meaning that the X coordinate of the first item will become equal to the X coordinate of the second item, as measured from the beginning of both UI elements. The second attribute would align the
first item’s left edge to the second’s item’s right edge, meaning that the X coordinate of the first item would become the sum of the second item’s X coordinate plus its length along the X axis, and so on.

An item can have multiple constraints, even on the same axis. If this is the case, a calculation is performed, taking into account all constraints and coming up with an average. For example, let’s say we have three Ui elements (figure 2): A, B, and C. B is attached to A through the left_toRightOf constraints and is also attached to C through the right_toLeftOf constraint, meaning it has apparently contradicting constraints which are on the same axis. In such a case, the left and right constraints are averaged and B ends up being centered between A and C.

Figure 2: Same-axis left and right constraints resulting in centering

```
A    B    C
```

There is the case of impossible constraints. This happens when there are multiple same axis constraints, and the behaviour described in the previous paragraph cannot happen due to various reasons, like for example lack of space. In this case, an assert is thrown informing the developer of the error. Asserts are disabled in released products.

Information about constraints is stored in a directed graph-like structure. Each node contains information about whether it has been visited and a list of neighbours which symbolizes that element’s dependencies. All elements are initially marked unvisited. When computing the relative positioning of elements, the first elements that are visited are the ones that have no unvisited dependencies. An item is only visited if it either has no incoming dependencies or all of them are marked visited. This is iterated until there are no unvisited elements left in that graph. In addition, the algorithm searches for circular dependencies, which are forbidden and are met with an assert.

If a UI element changes in size or position, its dependencies or the UI elements that the changed item depends on do not need to be modified. What changes are the incoming dependencies (items that in turn depend on the changed item). The changed item and all of its incoming dependencies are marked unvisited again, cascading through the entire graph, as necessary.
Also, an optimization has been made so that if the system detects a change along an axis, only calculations on that axis are performed (if only the width or the vertical position of the element have changed, there is no need to check constraints concerning the top axis, like top_toTopOf).

There is also a feature permitting the specification of an element size not as an absolute value, but as a percentage of another element’s size. This is achieved through the help of the ID system, an already implemented system which is also necessary for relative positioning.

Lastly, there is the possibility of declaring margins for an element to supplement the relative positioning constraints. Those margins correspond to the four edges of the element and also contains a shorthand for all four edges, like the margins in containers.

5.3 Style files

Style files are files that contain certain properties of UI elements not included in the main specification file. XLN Audio’s Lua framework specifies GUI layouts in Lua tables. Lua tables are similar to the hashes or dictionaries in other programming languages. A key in the Lua table is a string or index, while a value can be either a primitive or another Lua table, thus allowing the nesting of Lua tables. This is a good solution for specifying UIs similar to how HTML works, with multiple levels of depth and multiple attributes for each element. If an element has many attributes (which happens even more now due to all the features that have been implemented during the project), the main GUI specification can get visually cluttered and hard to write, maintain, and scroll. For this reason I have created a system that divides the main UI element declarations from some of their attributes, similar with how the division between HTML and CSS works.

First, a simple division based on element ID was performed, meaning that the attributes in the style file could only be assigned to one specific element. Then, inspired by CSS, more specifiers were added, which made possible assigning a group of attributes not just to one element. The specifiers are the following, in the descending order of their priority:

- The ID of the element, indicated by the "#" sign in the style name, can only refer to a single element. If, by accident, two or more elements share an ID, an error occurs. The attributes in the style file attached to an ID will only apply to the element in question.
• The style class, which is indicated by the "." sign in the style name, can refer to multiple elements that have this class. This style class is attached to elements through an attribute in their specification. All elements that share this class will benefit from the attributes declared in the style file under this name.

• The UI component class, which is indicated by the name of the class with no special symbol needed (like "cButton"). This refers to all elements of that component type. This will automatically apply to all elements that are of this class.

In addition to the above, specifying attributes in the main layout specification file will overwrite all style file specifiers, making this the highest priority attributes that can be declared. This semantic choice is an obvious choice for preserving backwards compatibility with existing products that do not use style files and keep all attribute declarations in the GUI specification files.

All of the above is accomplished by intercepting the creation of UI elements, parsing their attribute fields, and checking if they have declared any style files or style classes. Then, the style tables in memory are analyzed to search for the relevant styles, and UI element attributes are changed according to the priorities declared above.

In the above snippet, we have the following example for style classes:

```json
{
    [#acceptButton] = { size = {140, 70}, pos = {0, 0} },
    [".containerButtons"] = {size = {50, 50}, pos = {100, 300} },
    ["cButton"] = { size = {85, 60} },
}
```

**5.4 Properties based on device size**

This allows declaring some element properties for specific screen sizes. It complements other features in the responsive GUI framework and can be used when relative positioning or containers are not the optimal choice, or it can be used in conjunction with them (for example, transforming a row container to a column container when the device width decreases below a certain developer-specified value).
This was achieved by extending the style file specifiers described in section 5.3 to also include screen size ranges, beyond the initial ID, style class and component class specifiers. The screen width can be given in a certain range, and styles will only apply for a certain element only if its width is within that range. This is a more restrained version of adaptive design (described in section 2) and only includes separating certain UI element attributes according to size, not entire layouts.

Figure 4: The container with that ID will be a column when having a width of between 400 and 600, and a row when having a width between 601 and 800.

```json
{
    [\{400, 600\}, {
        [".container1"] = \{ cType = "column" \}
    }],
    [\{601, 800\}, {
        [".container1"] = \{ cType = "row" \}
    }],
}
```

5.5 Testing

All features were tested both manually and through the use of unit tests provided by a testing framework inside the Lua platform. Unit tests were created for all features as they were implemented. These tests help with future code maintenance in particular, since they make it easier to verify that previous parts aren’t broken by new ones. Without unit tests, every new feature would require manual testing of each and every previous one.
6 Conclusion

When it comes to answering the main questions of the thesis, whether features in existing frameworks are relevant or not for our framework and which of them are of highest priority in the project’s limited time frame, the answer to both came during the feature specification phase. Thus, the choice of features can be justified through practical decision-making with the software design and development teams.

I implemented the main features decided during the early planning meetings and included in the feature specification. Tests show that the new responsive GUI respects the specification, and that the new features are compatible with the old framework.

In conclusion, I created a working solution for responsive layouts inside the company’s products. This initial solution can be enriched with more features and even extended in the future, as recommended in section 7.
7 Future work

There were a multitude of features that were excluded from the feature specification due to the very limited time of the thesis, and in all probability there might be features that I have missed during my research phase. Some of them could prove useful to the responsive Lua framework, even though they were regarded as lower priority during this project.

For containers, a direction feature could be implemented. The direction attribute would specify the direction the elements will be placed:

- Normal means from left to right or from top to bottom for rows and columns, respectively (default).

- Reversed means from right to left or from bottom to top for rows and columns, respectively

In addition, item shrinking in containers could accompany the current item growth feature. This would give the possibility of item to shrink if not enough empty space is left, to prevent either wrapping or hiding of UI elements inside the container. As an alternative to wrapping or hiding, scrolling could be implemented so containers would now fit all elements on the main axis.

Right now, containers wrap/hide elements when either reaching their maximum capacity or window limit. This could be extended to include parent containers or even other sibling UI elements.

As for relative positioning, a few omitted elements where the guidelines and barriers from Android, which could help with declaring dynamic layouts.

There could also be improvements in existing code, by making optimizations in the various algorithms used by both relative positioning and containers. One example could be to minimize the amount of computations performed in the container when an item changes by pruning out even more rows, columns and items before running the algorithms.
References


