W^2 : Synthesizing Responsive Webpage from Wireframe

Shangyin Tan* and Tiger Tang*

Purdue University
{tan279,tang426}@purdue.edu

Abstract. We present a novel approach to synthesize and generate a nested and responsive UI layout from a wireframe, a common tool used by UI/UX designers to draft their design of a client application. Given the wireframe design of an application's layout in absolute coordinates, $\mathbf{W}^2$ synthesizes an HTML file that when rendered by any modern browser, not only places the components in their original positions, but also automatically adapts to changes in the viewer's screen size using modern HTML and CSS techniques. Furthermore, the resulting HTML has a clearly defined parent-child hierarchy, enabling the application developer to iterate and detail it quickly. Preliminary benchmark results show that W2 is able to synthesize an absolute wireframe layout consisting of 200+ rectangles into an HTML hierarchy tree of depth 5 in under 2ms.

Keywords: Program Synthesis · Web Development · User Interface

1 Introduction

UI/UX design plays an important role in most software applications. It allows software users to interact with the software functions and perform the designated tasks in an intuitive and effective way. The usual client application development cycle is as below: the UI/UX designer first designs the application layout, then provides it in image format to the application developer, who then writes code that when rendered resembles the design. However, the latter task is often time-consuming and error-prone. For instance, the spacings between the layout elements, referred to as margins, are often not accurately implemented by the application programmer, either because they are not explicitly stated in the design or such process simply takes too much time, so that the programmer uses estimated values instead.

Furthermore, as the popularity of mobile devices grows, a single fixed layout is no longer sufficient. In the context of web application development, most websites today use a responsive layout, which refers to a flexible, fluid layout that resizes itself to fit the size of the viewer's display. For instance, consider the layout of YouTube [Figure 1], a popular video website.

* equal contribution

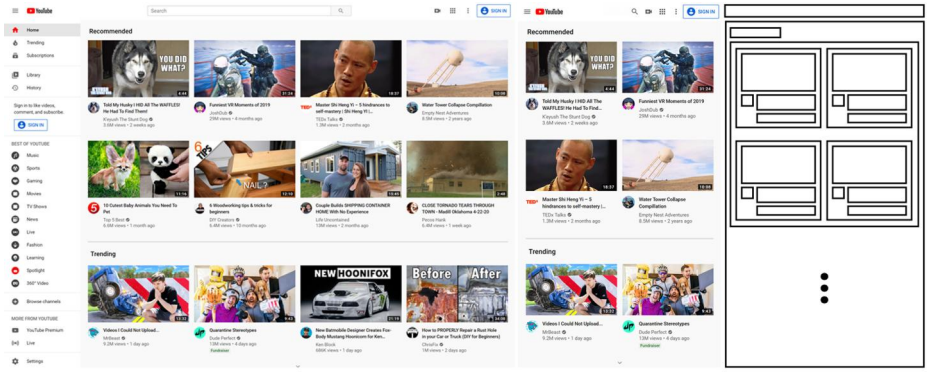

Fig. 1. Difference in Youtube Layout

When the viewer is using a desktop computer whose resolution width is higher than a threshold in pixels, the website shows the videos in 4 columns; when the viewer is using a mobile device with a much smaller width in pixels, however, the website shows the videos in 2 columns. To achieve this result, both the UI/UX designer and the application programmer have to do extra amount of work: the UI/UX designer must design multiple versions of the layout in different screen sizes, and the application programmer must write code that incorporates these layouts such that it renders correctly across all supported screen sizes. Specifically, the application programmer has to work out the parent-child relationships between the elements, known as the layout hierarchy. Unfortunately, this task is also time-consuming and error-prone, as the layout relationships are not immediately clear. For example, [Figure 1] shows a simplified layout hierarchy of YouTube when rendered on a mobile device, in which the innermost element has a depth of 5. Along with even more complicated margins, this task poses a considerable challenge to the application programmer.

Our Work In this paper, we present a proof-of-concept framework, named $\mathbf{W}^2$ (Synthesizing responsive Webpage from Wireframe), to address the stated challenges. Specifically, we present the following technical contributions:

1. $\mathbf{W}^2$ introduces a fast sectionalization algorithm to synthesize any absolute layout with hundreds of elements, specified by their absolute screen coordinates (x and y) and sizes (width and height), into a nested layout with hierarchical information inferred under a few milliseconds.
2. $\mathbf{W}^2$ introduces a direct mapping of nested layouts to an HTML DOM (Document Object Model), exploiting the latest HTML5 features: flexboxes and grids. The resulting HTML can be rendered by any modern browser on its compatible devices.

2 Related work

There has been similar research in the area of generating layout code from images/rectangles. However; we did not find any work that comes too close to $\mathbf{W}^2$: there is a machine-learning based approach to generate a web page from a mockup image [Huang et al. 2016], but it does not have a hierarchical layout specification; another approach aims to synthesize a relational layout[1] for Android applications from a set of rectangles [Bielik et al. 2018], but it produces a flat hierarchy that relies on ConstraintLayout, a newly introduced feature in the newer versions of Android SDK, which is not as portable as HTML5 which is universal and cross-platform.

There is also work on generating UI layout for alternative screen orientation [Zeidler et al. 2017] and vectorizing UI screenshots [Swearngin et al. 2018]; but, once again, they did not target the web platform like $\mathbf{W}^2$, which leverages HTML5 features to enable the resulting layout's responsiveness.

3 Illustrative example

When the wireframe is designed, we are given a static layout graph [Figure 2].

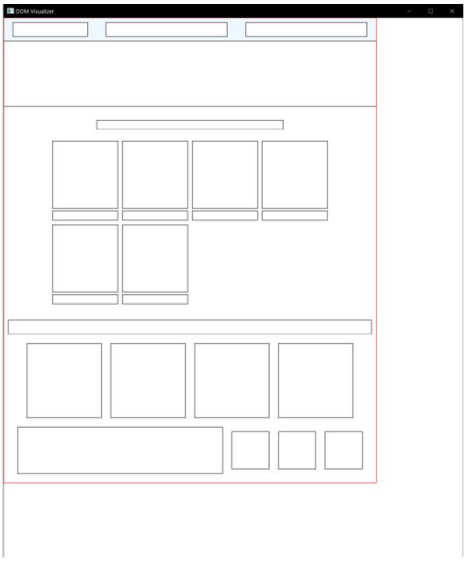

Fig. 2. Static Wireframe Graph

[1] Note that "relational layout" is not the same as "hierarchical layout." In their work, the term "relational" refers to how the elements are relatively positioned against each other, i.e. the inter-element spacings.

We aim to transform this static layout to HTML script can directly run in browser. With the inference of the hierarchical structure by $\mathbf{W}^2$, the website generated can adjust and adapt to various width. [Figure 3]

Fig. 3. Responsive in Different Width

4 Formal problem definition

In this work, we are given a list L, consisting of rectangles given their absolute coordinate with respective to origin. In addition, user provides the default size of the layout. All rectangles should be bounded by default size. Our objective is to find an equivalent HTML script which is:

1. **Precise -** Render the exact same layout when opened in default size.
2. **Responsive -** Automatically adjust the elements rendering if the browser is resized.

5 Preliminaries

5.1 Workflow

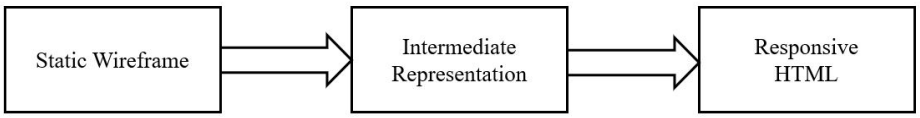

Fig. 4. Workflow

We define our workflow as a two-staged program. In the first stage, we synthesize the input Wireframe rectangles to IR. Then IR is compiled into universal HTML compatible with any browser in the second stage.

5.2 Input

The input is defined as a list of rectangles, given their coordinates related to $(0, 0)$, the upper-left corner of the canvas. For the consistency, we record the coordinates of the upper-left corner of a rectangle, as well as its width and height.

5.3 IR Definition

As discussed before, the intermediate presentation captures the nested hierarchy of raw rectangles. To maintain the hierarchical structure, we defined the data structure below.

```
size = <Int, Int>
margin = <Int, Int, Int, Int>
Div = <size>
HDiv = <size, margin, VDiv*, Grid*, Div*>
VDiv = <size, margin, HDiv*, Grid*, Div*>
Grid = <size, margin, VDiv*, HDiv*, Div*>
Layout = Div | Grid | VDiv | HDiv
```

Fig. 5. Definition of IR

We define `Div` to be basic component of the hierarchy, which is equivalent to a raw rectangle in the input. All solid input rectangles are transformed into `Div`s in different level of the tree in separate time during synthesizing. The three virtual containers for hierarchy are `HDiv`, `VDiv`, `GDiv`. As the name suggests, child containers of `HDiv` and `VDiv` are rendered in horizontal and vertical direction

respectively. We define `Grid` to behave like an HTML grid: two-dimensional container with auto row-wrapping. It contains homogenous elements, all rendered with same margin.

6 Algorithm

In this section, we explain the procedures we used synthesizing the list of rectangles to IR with an estimate order of time complexity. We also present the compilation process from IR to final HTML result.

6.1 IR Synthesis

Recall that the input is a list of rectangles. This is the starting point of the synthesis, and we target the nested hierarchical IR.

Sectionalize

Definition 1. *`section` We define a `section` to be a set of rectangles which bounded by an inclusive lower bound B_{low} and an inclusive upper bound B_{up} in either x-direction or y-direction. In range $[B_{low}, B_{up}]$, we cannot find another bound B_{mid} such that either $[B_{mid}, B_{up}]$ or $[B_{low}, B_{mid}]$ can form a `section`.*

Given the list of rectangles, we divide such list into different `section`s. To optimize the future operations, we want to find the max number of `section`s. We propose the following sectionalize method.

```
/*
Input - a list L of rectangle
Output - a list S of section
*/

sectionalize(L: List[Rectangle]): List[section] = {
  val upperBounds = L.map(rectangles => rectangles.getUpperBound())
  val sectionBounds = upperBounds.filter(
    line => doesNotCutOtherRectangle(line))
  var S = List()
  for (sectionLine <- sectionBounds) {
    S = S + getRectangleBetween(sectionLine, lastSectionLine)
  }
  return S
}
```

Fig. 6. Sectionalize

In the above method, we first filter all upper-bounds of rectangles that does not go through other rectangles. Then, we argue that these lines are the bounds

of that maximize the number of `section`s. The proof is trivial and omitted. In each `section`, we iterate through list L and find all rectangles belong to this `section`. Note that no specified direction is used in the above method, so it applies to both horizontal and vertical direction: we can sectionalize L based on x or y coordinate.

Note that every `section` can be easily converted into a `VDiv` or `HDiv`. We calculate the margin of each child rectangle of the `section` by calculating the difference between that rectangle's coordinate and `section` origin, which can be obtained by finding intersection of B_{low} and outer coordinate.

Before the sectionalize operation, every rectangle is in the same level and container: every individual rectangle will be rendered with same priority and order. After we sectionalize a set of rectangles, we increase the depth of the hierarchy tree by 1. Now each rectangle must belong to one `section`. If we view each `section` as an individual component, there will be another virtual *master* container that contains all the `section`s, which is visualized in the following graph,

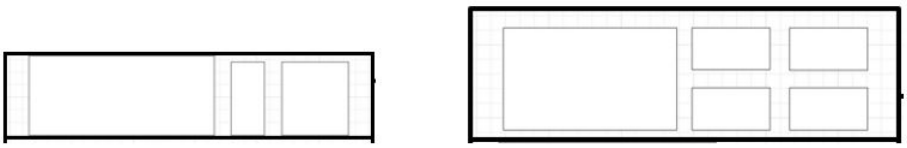

Fig. 7. Simple vs. Complex `section`

FormGrid The central part of forming and optimizing the hierarchy relation is to identify `Grid` from `section`s. We see that in the previous sectionalize method, we generate a suboptimized hierarchical strucuture as we can put every rectangle into different `VDiv` and `HDiv` after sectionalizing. To further optimize the previous result, we want to merge different `section`s into one `Grid`.

For every `section`, we classify the `section` into *simple* and *complex* lists. We define *simple* `section` to be the section that contains only similar rectangles and are able to form `Grid`. *Complex* `section` is cannot merge with other `section` to form `Grid` but can contain `Grid`.

Then we iterate through the list of `section`s and enumerates all possible of combination of `section`s. We group larger `Grid` first and then process smaller ones. For the `section` fail to merge with other `section`, we put them into the *complex* `section` for future processing.

Complex Section *Complex* `section` indicates the need for further processing. The nature of web-page rendering suggests that elements are either arranged horizontally or vertically. Therefore, if the *complex* `section` was sectionalized based on x-coordinate, we now sectionalize it based on y-coordinate and vice

versa. Within each *complex* `section`, we perform the above sectionalize and formGrid operation recursively until we meet max-depth or a single `Div`.

In each execution of sectionalize and formGrid operation, we increase the depth of the hierarchy tree by 1. The result tree after termination is the IR we get and then feed into the IR-HTML compiler. There are rare cases where the hierarchy tree produced fail to be optimal layout. We will discuss in future sections.

6.2 Mapping IR to responsive HTML

Finally, we compile the nested layout we synthesized to an HTML DOM by mapping. We use `flexbox` and `grid`, two layout elements in the latest HTML5 specification, to resemble the behaviors we defined for `HDiv`/`VDiv` and `Grid`. Specifically,

1. All `HDiv`/`VDiv` containers in the IR are mapped to `flexbox` in HTML.
2. All `Grid` containers in the IR are mapped to `grid` in HTML.

However, this simple mapping is not sufficient. For instance, `div` wrappers are required for `grid` elements for proper alignment with their parents; extra CSS (Cascading Style Sheets) also has to be added to ensure the element margins are correct. For implementation details, please visit the project source repository.

7 Evaluation

We ran $\mathbf{W}^2$ on a small set of representative test cases ($|s|=5$). Test case 1 to 3 are simple layouts; test case 4 is a design that is close to that in a real-world scenario (most rectangles are irregularly aligned, multiple layers of depth), and test case 5 is for stress testing (> 200 rectangles). The results are as below:

Test case #	1	2	3	4	5
# of input rectangles	3	3	9	26	210
Time taken	0.10ms	0.10ms	0.18ms	0.29ms	1.53ms

Fig. 8. Benchmark

As our algorithm has a very limited search space and most UI layouts have fewer than 100 elements, it runs extremely fast-much faster than most similar work. Also, unlike neural-network based methods that often generate code that is non-interpretable or syntactically incorrect, $\mathbf{W}^2$ produces a clean and robust DOM tree that is ready for display in the browser and easy to modify.

However, there are some limitations with $\mathbf{W}^2$. For example, for a layout shown in [Figure 8], there exists two possible ways to sectionalize the layout, and $\mathbf{W}^2$ always chooses the latter, due to its nature of returning the hierarchical tree of maximized depth. In the practical sense, though, the former sectionalization is more common as most websites adapt a multi-column layout. We have yet to clearly define a user-customizable parameter to measure the optimality of our synthesized hierarchy; hence, the resulting layout may not match the user's intent.

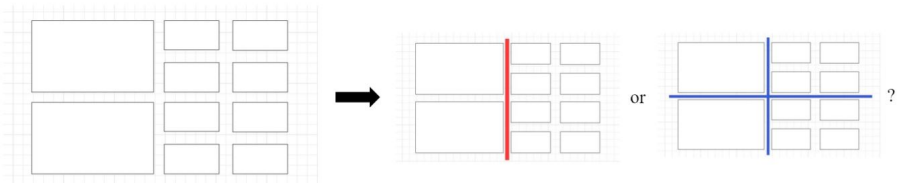

Fig. 9. Limitation

8 Conclusion

We presented a lightweight framework to produce a human-readable, hierarchical and responsive UI layout in HTML5 from a list of rectangles provided in absolute coordinates. Our technical contributions include a fast algorithm to infer the implied hierarchy in the wireframe and a direct mapping of nested layout to an HTML DOM tree with the apt use of HTML5 flexboxes and grids.

Because $\mathbf{W}^2$ is created as a proof-of-concept, we did not include many useful features that are trivial to implement, such as importing the rectangles from SVG, a popular vector graphic format, or exporting the resulting HTML to image formats of different screen resolutions. Other future work may include extending the input specification, such as allowing the user to decide which elements should be responsive to screen width changes and which should be fixed-size (currently, all elements are responsive), improving the algorithm so that it can form grids of elements of different sizes, and formalizing the definition of the optimality of a nested hierarchy.

Reference

1. Ellis, Kevin, Daniel Ritchie, Armando Solar-Lezama and Joshua B. Tenenbaum. *"Learning to Infer Graphics Programs from Hand-Drawn Images."* NeurIPS (2018).

2. Huang, Ruozi, Yonghao Long and Xiangping Chen. *"Automaticly Generating Web Page From A Mockup." SEKE* (2016).

3. BielikPavol, FischerMarc and VechevMartin. *"Robust relational layout synthesis from examples for Android."* (2018).

4. Beltramelli, Tony. *"pix2code: Generating Code from a Graphical User Interface Screenshot." ArXiv abs/1705.07962* (2017): n. pag.

5. Zeidler, Clemens, Gerald Weber, Wolfgang Stuerzlinger and Christof Lutteroth. *"Automatic Generation of User Interface Layouts for Alternative Screen Orientations." INTERACT* (2017).

6. Swearngin, Amanda, Mira Dontcheva, Wilmot Li, Joel Brandt, Morgan Dixon and A. J. Ko. *"Rewire: Interface Design Assistance from Examples." CHI '18* (2018).