Irene - A Netball Visualisation System

Aden Kenny

I. Introduction

For the SWEN 422 course we were given a project to "design and implement a software system for an information visualisation application". This software system was required to allow "user to interactively explore a dataset containing the results of all matches in the first six seasons of the ANZ Championship.". Irene [1] is our implementation of the set specifications. Irene is a web application built with TypeScript, CSS, React.js, and D3.js. Irene was designed to be a personal assistant like Alexa [2] or Siri [3]. The major difference between our design of Irene and the former two, was that it was not designed to be a general purpose assistant, or have any natural language processing ability. Irene was designed purely to display data from the ANZ Championship. Irene allows the user to ask a limited number of questions and shows the user graphs that help to answer their question.

II. Key design decisions and justification

A. Overall Design

Our user interface was designed to be rather minimalistic, with a monochromatic colour palate of black and white. This colour palate was chosen as it complements the minimalistic design. The colour palate also has justification in colour theory. White is generally associated with "light, [and] goodness ... It usually has positive connotations and is seen as clean and safe" [4]. The goodness aspect of white was considered to be important as Irene is meant to be a person who you're almost having a conversation with, and this is one of the reasons why white plays such a large part in the chosen palate. Additionally, the clean aspect of white was important as we wanted our design to look clean, sleek, and minimalistic. Black is seen as "formal and elegant" which we felt were important for our design, therefore black was used, mainly through the header, and as a standard font colour. The the black and white colour complement each other well by providing a clean look, but the two colours contrast well, letting us present a strong but clean design. A good example of this is looking at the header, where the background is black, providing a strong and solid background, then both the logo and title are a white that contrasts well with the black background. The colours represent their associations well, and the contrast of white on black helps to deliver our design goals to the user.

We also considered whitespace to a key principle in our design. A good interface needs to have whitespace or it becomes difficult to navigate [5], and therefore to use. The white space of our interface contrasts to the information and elements that are presented. This is especially important when we display our graphs, and other data, as we want to draw the users attention to them, and not have the users distracted by other elements.

B. Prediction Bar

One major design decision was the inclusion of the "prediction bar". This is the bar at the bottom of the screen that user can type their questions into and have them answered. When a user enters in a query they are taken to the relevant page where a graph is presented. All navigation of the system is through the prediction bar. This means that if a user does know how to or cannot use the prediction bar, they cannot use the system. In order to make sure that the user knows to use the prediction bar, we made sure that we had a strong visual hierarchy that directed the user towards it. The bar also has an autocomplete feature, but it has an extremely limited number of questions. The prediction bar is less of a search bar and more of a way of displaying questions. I feel that it is a fairly effective way of displaying questions and "railroading" a user towards what we want them to ask. The prediction bar is also designed make the user feel like they're having a conversation with Irene. We wanted the system to have a friendly feeling, rather than feel like a simple application where you click and a graph is shown, and I feel that the prediction bar helps the system to have a more human feel.

It is open to debate if the prediction bar actually improves the user experience. I feel that if done well, and with enough time and resources, and in a large scale program, the prediction bar could prove to be a major selling point of the system. However, in this small, limited scale project it may have proved to not be a good design decision. This is primarily due to the large amount of development time it took to produce, and the fact that it feels slightly unpolished, primary due to the lack of questions.

Overall I feel that the prediction bar was good design but it was perhaps not suited to this scale of project. It is extremely easy to add new questions to it but the development of it took a large portion of the project. This development time could have been better spent upskilling or creating more graphs. It would also be more suited for a larger scale project where more questions were required to be answered, and the development cycle was longer.

C. Radar Chart

We decided to use a radar chart to plot a teams overall performance. This means that it uses performance variables of teams (points, wins, losses, goals for, goals against, and goals for). The overall area of each team on the chart is meant to show the overall performance of a team i.e. a large area covered, the better the performance of the team. We considered the area correlating to a team's performance to be extremely important as humans are drawn towards the area of the a chart, and we felt that having the area not represent anything would be a waste of the strengths of the radar chart over a different

type of graph. Correlating the area of the graph with the performance of a team did bring some downsides to focusing on the area. A major reason for this is that a team is performing better when they have less goals against, but with the standard data in a radar chart, more goals against would mean a greater area covered, and therefore better performance.In order to make a low goals against value be bad for the performance of a team I decided to "invert" the goals against on the graph. To do this I went through all the goals against and found the highest (the lowest number), and added that number to all goals against. This meant that team with the worst goal difference would have zero goals against on our chart. I also scaled losses with the same technique.

After this, we still had the problem of some data points being significantly larger than others, such as wins and goals for. This meant that it was impossible to tell the difference between teams in terms of points and other statistics that had a lower mean value. In order to solve I decided that I would get a baseline value from the chart and scale values based upon that. This meant that all values fit nicely on the chart. The downside of this, along with the scaling of goals against and losses is that we have traded the area of the graph being significant for a lack of data values being accurately reflected in the graph. The only values that are not scaled in some way are the points and wins of each team, all other values are scaled but they are scaled so that the relationships between each teams variables are not impacted as every team is scaled by the same scale factors.

Overall I feel that the changes made in the data for the radar chart were worth it. It did mean that some data had to be scaled and inverted, but it preserved the meaning of the area of the chart. Without the area of the chart being meaningful, there was no real point of using a radar chart, therefore, in my opinion we made the right choice by choosing a radar chart, and scaling the data so it worked for the chart.

D. Bar Chart

III. Alternative design decisions

A. Overall Design

B. Prediction Bar

C. Radar Chart

IV. Critique of development tools

A. D3.js

As required for this project we used D3.js [6] to construct graphs. I found the usage of D3 to be rather difficult in some cases especially with our usage of React for building the other, non graph, UI elements. This is because both React and D3 want to control the DOM, and common usage patterns of D3 are "in direct conflict with React and its virtual DOM" [7]. Examples and tutorials were also harder to find for using D3 with React as opposed to not using it. This made it more difficult to learn than it could have been otherwise. In the end, we found it much easier to use libraries that wrapped D3 charts that were designed to be used with a React program. These wrapped turned a D3 graph object (generally an svg) and turned them into a React component which made it extremely easy to use them with our program.

Another major issue we found was our combination of D3 and TypeScript. Significant amounts of online examples of D3 graphs would not work in our codebase. This was because they were coded to take full advantage of the "laxness" of JavaScript and its syntax. Examples of this included variables being declared without a variable declaration statement [8] e.g.

```
dataValues = [];
```

compared to

```
let dataValues = [];
```

TypeScript will not allow the first snippet of code as it will create a global variable (assuming *dataValues* had not been previously declared), and is bad practice as it is important to keep the scope of variables as small as possible especially if they are mutable (as they are in this case).

Our issues with poor quality of code in D3 examples and tutorials online did not result in TypeScript problems. We found significant amounts of examples that did not include semicolons. JavaScript without semicolons is technically syntactically correct code, and the JavaScript parser will insert semicolons where it thinks they should be, but this can result in unexpected and undesired behaviour. We found an example without semicolons that caused unexpected behaviour but when we attempted to fix it by inserting semicolons, we realised that the example relied on the compiler inserting semicolons in unexpected places.While this is not directly a fault with D3, but an issue with the ecosystem (JavaScript and some the of developers), it still impacted how we used and viewed D3.

Another issue we faced with D3 was the lack of backwards compatibility. Major D3 updates (v3 to v4, or v4 to v5) seemed to remove old functions and features and replace them. This caused problems with older examples and tutorials that no longer worked. One solution would have been to downgrade to an older version, but some other examples that we found used some of the newer features that did not exist in the older versions. Additionally the newer versions fixed some bugs that were present in older versions. A lack of backwards compatibility is always hard to work with in a software context, and our experience was no exception to this rule.

Although D3 did present many problems throughout our project, once we had React wrappers for D3 graphs, it became much easier to implement our graphs. Overall I found that when working properly, D3 can be a powerful, and easy to use tool, but unfortunately there are some major downsides to it which can make it extremely frustrating and hard to use in some cases.

B. React

We used React.js [9] for building our user interface. I found React to be easy to use, and it made the sometimes difficult process of creating HTML and CSS interfaces much easier.

React encourages the use of embedded HTML (in the form of JSX or TSX) to create components which can be reused as objects for creating a user interface. These components allow for easy reuse of code which means there is less code, and the code is generally "cleaner". A smaller amount of "cleaner" code is considered easier to reason about, which helps with preventing bugs.

The component-object style of user interfaces felt far more natural to use when compared to the standard HTML, CSS, and JavaScript style. This was probably due to most of my previous experiences creating user interfaces being in an object oriented style (Swing, Qt, JavaFX, Android).

React also proved to be easy to interoperate with TypeScript, which is slightly surprising given that TypeScript is maintained by Microsoft, and React by Facebook. The only major difference is that JSX (the HTML like syntax for building React components) can be put into .js files, but TSX (the TypeScript version of JSX) cannot be put into .ts files, it must be put in .tsx files.

C. TypeScript

We used TypeScript [10] in this project as a direct replacement for JavaScript. Since TypeScript is a strict superset of JavaScript, the TypeScript compiler complies our TypeScript code to plain JavaScript so, from a code behaviour perspective, there is no difference than if we were using JavaScript. I found TypeScript to be extremely useful in this project. The added static typing proved very useful when contrasted with JavaScript's dynamic, weak typing. TypeScript allows the usage of type signatures which makes reasoning about the code significantly easier in my experience. The usage of TypeScript also allowed us to catch a significant number of bugs much earlier than would have been possible if we were using JavaScript. This is primarily due to the fact that TypeScript is compiled to JavaScript therefore has a compilation step where some compile time bugs are caught. This is contrast to JavaScript, which is interpreted, and therefore bugs cannot be caught at compile time. The static nature of TypeScript also gave us autocompletion and far better refactoring support than if we were to use JavaScript. This was especially helpful as we are relatively new to web development and refactoring and autocomplete made our lives significantly easier while learning, especially when combined with a powerful IDE like Visual Studio Code [11].

Overall I found TypeScript to be pleasant to use in this project, especially in contrast to if we had to use JavaScript. The main reason for this is due to TypeScript's static typing. Using TypeScript made our program feel significantly less like "a house of cards" compared to previous experiences of programming in JavaScript. Unfortunately due to being a strict superset of JavaScript, TypeScript still has many of the warts of JavaScript such as the equality operator coercing the values. Overall, while I feel that TypeScript is a significantly more pleasant language to program in, and is easier to use than JavaScript, and it still has some major flaws, most of them are inherited from JavaScript.

V. Conclusion

References

[1] "Irene by SWAG." https://adenkenny.github.io/Irene/, 2018. [Online; accessed 26-August-2018].
[2] "Amazon Alexa." https://developer.amazon.com/alexa, 2018. [Online; accessed 26-August-2018].
[3] "IOS - Siri - Apple." https://www.apple.com/nz/ios/siri/, 2018. [Online; accessed 26-August-2018].
[4] "How To Use Color To Enhance Your Designs - Vanseo Design." http://vanseodesign.com/web-design/color-meaning/, 2010. [Online; accessed 26-August-2018].
[5] "Whitespace: Less Is More In Web Design - Vanseo Design." https://vanseodesign.com/web-design/whitespace/, 2009. [Online; accessed 26-August-2018].
[6] "D3.js - Data-Driven Documents." https://d3js.org/, 2018. [Online; accessed 25-August-2018].
[7] "Interactive Applications with React & D3 - Elijah Meeks - Medium." https://medium.com/@Elijah_Meeks/interactive-applications-with-react-d3-f76f7b3ebc71, 2017. [Online; accessed 26-August-2018].
[8] "D3.js - Radar Chart or Spider Chart." http://bl.ocks.org/nbremer/6506614, 2018. [Online; accessed 26-August-2018].
[9] "React - A JavaScript library for building user interfaces." https://reactjs.org/, 2018. [Online; accessed 25-August-2018].
[10] "Typescript - JavaScript that scales.." https://www.typescriptlang.org/, 2018. [Online; accessed 25-August-2018].
[11] "Visual Studio Code - Code editing. Redefined.." https://code.visualstudio.com/, 2018. [Online; accessed 26-August-2018].