Exploration of Interactive Visualization in the ELM Architecture

Monika Wissmann*
Supervised by: Harald Steinlechner†Andreas Walch ‡

VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH
Vienna / Austria

Abstract

Coupled visualizations and interactions in 2D and 3D represent a large part of the work in many visualization projects. By using the *ELM* architecture, these interactions are approachable, structured, composable and easy to test. *ELM*-style web applications are currently gaining importance and have already established themselves in web-development.

In this paper, we implement a ranking view using this architecture. *Aardvark.Media* is a functional programming interface to synchronize data and its visual representations while providing high-performance rendering. Based on the ranking view we demonstrate the power of an incremental rendering system. The incremental rendering system enables us to achieve drastic reductions in the creation of the complex 2D user interface.

Keywords: Interactive Visualization, Incremental Rendering, Functional Programming, Domain-Specific-Languages

1 Introduction

Interactive visualizations are ubiquitous in various fields ranging from industrial applications over websites and data-science. The development of visualizations is challenging particularly when working with dynamic data. Interactive visualizations are even more difficult since user inputs need to be handled and visualization has to be updated accordingly. Depending on the use case, developers face various challenges:

- Domain modeling of complex, potentially hierarchical data.
- The choice of appropriate visualization technique.
- Multiple views on the same data are needed (e.g. linked views) on which synchronization can be tricky.
- For large dynamic visualizations immediate user response is crucial, which again makes performance tuning necessary essentially polluting domain logic with non-functional concerns.

*monika.wiszmann@gmail.com
†steinlechner@vrvis.at
‡walch@vrvis.at

There is a recent trend towards functional programming concepts in various fields of computer science. In the field of web-development for example, approaches such as *react* and *redux* embrace a functional architecture. In this work we take this approach to the extreme by using purely functional programming and the *ELM* architecture [5] as the basis for our implementation of an interactive ranking view.

We explore the *ELM* architecture for visualization by investigating *LineUp* [7], a state-of-the-art visualization tool stressing interactivity. In the implementation section we represent a case study, based on the existing framework *Aardvark.Media* [16], which implements the *ELM* architecture and provides the necessary features such as dynamic user interfaces and efficient update mechanisms. The contributions can be summarized as such:

- We present a case study of a complex state-of-the-art visualization technique implemented in the *ELM* architecture, present reusable parts and show the applicability of the approach in a demo application.
- In our evaluation we analyse and improve the system performance by utilizing optimized data-structures enabling immediate feedback.

2 Background

As our work is based on functional and reactive programming paradigms we first give a brief introduction to functional programming followed by an overview of reactive programming techniques. Afterward, we explain the concept of incremental evaluation followed by an overview of well-established visual systems.

2.1 Functional Programming

The functional programming paradigm treats computation as the evaluation of mathematical functions. It avoids mutable data and changing-state, thus eliminating side-effects, which makes the program much easier to predict.

Features like higher-order functions and lazy evaluation contribute greatly to modularity, which is crucial for well-structured software. Programs become easy to write, easy

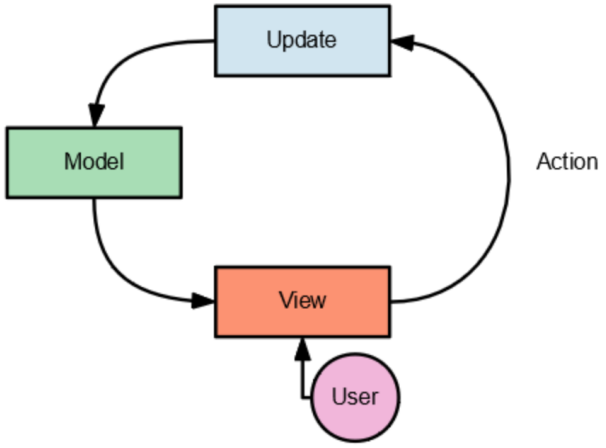


Figure 1: Visualization of the ELM architecture [16]

to debug, and it provides a collection of modules that can be re-used to reduce future programming efforts [10].

2.2 Reactive Programming

Reactive programming is a declarative programming paradigm that enables automatic propagation of change to dependent data. Because of this, it is well-suited for developing event-driven and interactive applications [1].

As a variation of the well-known *Model-View-Controller* pattern, the *ELM* architecture (see Fig. 1) makes use of the reactive programming paradigm. It consists of an immutable *model* that holds the current state and data. The *view function* generates a graphical representation of the model as *HTML* code. The user can interact with this representation and trigger *update actions* on runtime. The *update function* receives the action and updates the model accordingly in a reactive manner and therefore completes the *unidirectional data flow*. Important to note, and in contrast to traditional approaches the data-flow is purely functional, i.e. all data is immutable. As a consequence, programmers do not need to care about updating the user interface, since, virtually the complete UI is recomputed after each iteration and optimization is taken care of in the implementation. Because of the well-organized code, a programmer can navigate and find the lines of code that are of interest with ease [8].

Following code highlights the *ELM* architecture by a simple example. The model holds a single integer value.

```
type Model = { value : int }
```

The `Message` describes the possible actions.

```
type Message =
  | Inc
```

Given the model and a message `Inc` the update method increases the value of the model by one and returns the new model.

```
let update (model : Model) (msg : Message) =
  match msg with
    Inc -> { model with value = model.value + 1
     ↪  }
```

The model gets visualized by the `view` function. Clicking the button results in the update of our model. The resulting changes in `model.value` are instantly displayed.

```
let view (model : MModel) =
  div [] [
    button [onClick (fun _ -> Inc)] [text "+"]
    text "my value:"
    Incremental.text (model.value |> Mod.map
     ↪  string)
  ]
```

Further examples can be found at the *Aardvark.Media* platform [16]

2.3 Incremental Evaluation

Incremental computation means that only structures that are dependent on changes are updated. This provides fine-grained control of the visualization design [9]. Incremental rendering leads to the re-computation of only affected pixels limiting the number of re-computations and therefore reduces computation time [15].

In their course, Steinlechner et al. [21] presented an incremental approach to scene graphs. Scene graphs are the standard approach for representing virtual scenes in memory. To make the approach practical concerning dynamic changes to those graphs, incremental evaluation is used to avoid repeated evaluation of unaffected elements in their application.

Another example is an incremental rendering *virtual machine* (VM) [9]. Incremental rendering is introduced as a layer on top of standard graphics APIs such as *OpenGL* or *DirectX* in the form of a VM. It provides an optimized compiled representation of an arbitrary high-level scene, leading to significant performance gain.

2.4 Visualization Systems

The following section gives an overview of two commonly used visualization systems *D3* [2] and *Vega* [22] as well as *Aardvark* [17], the platform used in our work.

D3 (data-driven-documents) is a lower-level tool and *JavaScript* library that allows to create data visualizations using a data-driven approach by leveraging existing web standards like *SVG*, *Canvas* or *HTLM* [23]. Data can be passed in the form of arrays or formats like *CSV*, *JSON* and *XML*. *D3* allows to bind those data to the DOM and to apply further transformations. This has the advantage that no other technology or plugins other than the browser are required. It can be smoothly embedded in other scripts and JS frameworks without disturbing the rest of the code. *D3* is a popular tool in the visualization community because of its ease of use. A downside is that DOM

manipulation can be extremely slow for large numbers of entries. *SVG* also has performance limitations when dealing with large quantities of elements.

Vega (visualization grammar) provides a higher-level visualization specification language on top of *D3*. It enables fine-grained control of the visualization design. Its specification is a *JSON* file that describes an interactive visualization design. It contains properties and definitions for the data to visualize axes, scale transforms, encoding algorithms and legends. Signals and predicates modify the visualization depending on the user interaction. These specifications are interpreted by a runtime system to dynamically create visualizations, or it can be cross-compiled to provide a reusable visualization component, in the form of editable code for a specific visualization framework (such as *D3*) [22]. *ELM* provides a *Vega* package [4] which generates *JSON* specifications that may be sent to the *Vega-Lite* runtime to create the output. The potential downside of *Vega* is that highly customized visualizations may be more difficult to achieve.

Aardvark is an open-source platform for visual computing, real-time graphics, and visualization [14]. It is completely implemented on top of incremental computation primitives. This means all projects built on top of the *Aardvark platform* are incremental by default [15]. The platform contains a fully incremental visualization library [13] which is based on incremental rendering.
Another tool on the *Aardvark platform* is *Aardvark.Media*. It provides front-end and UI for *Aardvark* and is an interface to build applications without additional complexity for handling change like synchronizing data and its visual representations while providing high performance through incremental *HTML* generation (similar to *react*). Data changes update the DOM tree in an incremental manner so that only affected parts are modified [16]. This allows efficient computation of customized elements.

3 Case study

Decision-making is a complex task when considering multiple parameters of the available choices. In the following section we present the ranking view as a tool that may be helpful in decision making. We show examples of visualization techniques that have been applied to show ranked data. Based on this we define requirements for a user-friendly, interactive interface.

3.1 Ranking view

Ranking views help to decide whether movie to watch next or which GPU performs best for your budget. Some ranking views are based on a single attribute like number of sold copies for a bestseller list, others are composed of a variety of attributes. An example for such a multi-attribute ranking is a ranking view for smartphones that is composed of performance, features, battery life, quality of display and price.

In the case of a single attribute, ranking visualization is straight forward as the overall score corresponds to the value of this attribute. Multiple attributes contribute in various degrees to the overall score and therefore require visualization of the composition.

Because ranking views should help in decision making, the focus is on the interests of the user. Users can have different preferences as to which attribute has the most value for them. Hence the user should be able to explore different preference settings. In order to interpret and modify the result of the ranking view, advanced visual tools are required. An important aspect of interactive visualization is that the changes by the user should be immediately visible, therefore performance and efficient rendering play a crucial role.

3.2 Domain and user interface

Due to the broad application of rankings, a wide variety of visualization techniques are available. The most basic way to display ranked data is a spreadsheet. It presents a set of ordered data together with a label for identification and allows sorting according to an arbitrary column. A well-known purpose tool for this technique is *Microsoft Excel*. It provides a scripting interface that allows for great flexibility but is only mastered by experienced users. Another drawback of spreadsheets is the lack of interactive visualization. A detailed discussion of the design of spreadsheets was published by Few [6].

To enhance the readability of data it is more efficient to encode values in a graphical representation. Length is a commonly used visual variable, as it is used in histograms (see Fig. 2b) and other bar charts [11]. To make the values comparable, the bars are usually aligned to a common baseline. Bars can represent a single attribute or encode the sum of multiple attributes in the form of a stacked bar (see Fig. 2d). An example for an implementation is the *table lens* technique [18]. The visualization uses a focus-plus-context technique that works effectively on tabular information as it supports categorical data and allows the user to focus on multiple areas.

Our user interface and requirements are inspired by *LineUp* [7] which is a part of *Caleydo*, an open-source data visualization framework. It is implemented in *Java* and uses *OplenGL/JOGL* for rendering. *LineUp* makes use of the a variety of visualization techniques as described above.

3.3 Design goals and requirements

To design a user-friendly interface that fits the needs of an interactive, multi-attribute ranking view following require-

Figure 2: Overview of our implementation. (a) The ranking view displays data of cameras. (b) Histograms visualize the distribution of values per attribute. (c) Weight and preferred optimum per attribute. (d) Stacked bars visualize the overall score composed of the sub scores per attribute. (e) Weighting bar for interactive use. (f) Buttons to filter attributes. (g) Further actions.

ments have to be met:

R1: Encode rank The user should be able to quickly evaluate the rank of an individual item.

R2: Encode cause of rank The contribution of each attribute to the total score is shown, to understand how ranks are determined. As scores are not uniformly distributed, there can be gaps between ranks that could be of relevance. Hence, the user must be able to evaluate the relative difference between multiple ranks.

R3: Support multiple attributes The user should be able to grasp the total score of the items and comprehend how the single attributes contribute to it. For further refinement, the user should be able to weight the attributes by personal preference. The attributes need to be normalized to be comparable.

R4: Interactive refinement and visual feedback The contribution of a single attribute should be visible to the user and changes of preference have to be instantly visible. To make this possible, each attribute is assigned a weight to compute the combined score.

R5: Enable filtering and sorting The user can exclude attributes of little interest and to sort by a specific attribute.

The requirements are inspired by the work of *LineUp* [7]. To demonstrate our use case, we used a dataset of cameras with 13 properties such as weight, focal length, etc.[12]

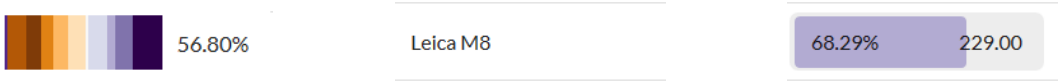
4 Implementation

Our implementation is based on the *ELM* architecture, as concrete implementation we use *Aardvark*. In the following section, we give an introduction to the software components used in our implementation:

4.1 F#

All components of *Aardvark* are mainly programmed in *F#*. It is a powerful language that spans multiple paradigms of development. It is functional by heart and as such, it focuses heavily on functions, expressions, algebraic types for creating domains and pattern matching for control of flow. Values are per default immutable to avoid side effects [20]. *F#* also supports object-oriented aspects like classes, inheritance and interfaces. The main advantages [19] of *F#* are:

- **Conciseness** through avoiding code "noise" trough brackets and semicolons and the powerful type inference system. It usually takes fewer lines of code compared with *C#*.



(a) Score with stacked bar (b) Textual attribute (c) Numerical attribute

Figure 3: Types of columns

- **Convenience** with functions as first-class objects that can hold another function as parameter. This enables easy creation of reusable, powerful code.
- **Correctness** trough a powerful type system, which helps to avoid incorrect code. Values are immutable by default to prevent side effects.
- **Concurrency** works great with the help of built in libraries. Because data structures are immutable sharing state and avoiding locks is easy.
- **Completeness** as *F#* has access to all third party *.NET* tools and libraries. Because *F#* is designed as a hybrid functional/object-oriented language it can do virtually everything *C#* can do.

4.2 Aardvark.Media

Aardvark.Media builds on top of the *ELM* architecture, which allows building large and complex apps trough the composition of smaller apps. It is a functional programming interface based on *F#* and provides incremental *HTML* rendering in addition to *CSS*, *SVG* and *JavaScript* usage. Several optimization mechanisms in the background are employed that allow programmers to focus on actual functionality instead of performance details leading to a cleaner, more robust and scalable code.

For some types like maps and lists, *Aardvark* provides an incremental implementation. This allows a faster and more efficient calculation of changes in the interface. In our work we will compare the implementation with the basic collection types (`plist<'a>` for immutable ordered lists, `hmap<'k, 'v>` for immutable maps with key `'k` mapping to values of type `'v`) and the automatically synthesized incremental datastructures (`alist<'a>` for incremalized lists, `amap<'k, 'v>` and incrementalized maps which both automatically track changes opposed to the non-incremental counterpars) to demonstrate the performance of an incremental rendering approach (see Section 4.4.2). More information about these types can be found in the *Aardvark.Media* documentation [16].

4.3 Basic design and interaction

Our ranking view is implemented as a table (see Fig. 2a), where the user can interactively sort and filter attributes in their corresponding column. We select a well distinguishable color scheme of *Colorbrewer 2.0* [3], which is also color-blind friendly to encode the different attributes.

Columns The columns correspond to one of the three types:

- **Score** is always visible and is represented by the leftmost column in the table. It shows a percentage value and a stacked bar as seen in Fig. 2d and Fig. 3a. The percentage is calculated depending on the optimum, as described in the header options. The stacked bar shows a serial combination of the attributes, whereby the width of each element represents the contribution of the attribute to the overall score. The attributes are encoded by color and the elements within the stacked bar are highlighted when hovering over the header label of the corresponding attribute.
- **Textual attribute** represents a string (Fig. 3b). An example in our dataset is the field `Model` holding the camera model name. Attributes of this type have no effect on the score or rank, hence neither the options nor the histogram in the header are available as seen in Fig. 4.
- **Numerical attribute** represents either an integer or a float value. It contributes to the calculation of the score. The length of the colored bar represents the percentage compared to the other values of this attribute (Fig. 3c).

Header The header consists of three rows (Fig. 4). The uppermost row shows a histogram of the distribution of the corresponding attribute. In the second row are the labels. A mouse click on an element sorts the view by the values of the corresponding attribute. The third row represents additional options to manually set the weight of the attribute and to switch the optimum between `Min` and `Max`. `Max` means that the higher the value is the better it is rated and vice versa for `Min`. This row is optional and can be masked out. If the weight of an attribute is changed, the rest of the weights are scaled, so that the sum of all weights equals 1. Only weights between 0 and 1 are accepted.

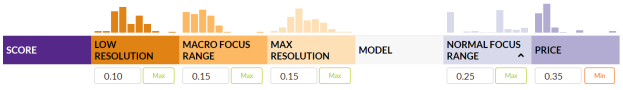


Figure 4: The header of the ranking view includes the histogram, label with sorting function and additional options for weighting or setting the optimum (min, max)

Weighting bar Another way to change the weights is the weighting bar (Fig. 2e, Fig. 5) underneath the table. The user can interactively alter the weight of an attribute by dragging the small red bar on the right of the corresponding attribute, which triggers an updated ranking of the solution. The width of the sub bars encodes the relative weight of the attributes. Each attribute of the type number that is currently not filtered gets included and is encoded by color and label.

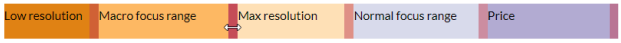


Figure 5: The weighting bar is used to interactively change the weights of the attributes by dragging.

Filtering To filter certain attributes the user has to click on one of the red buttons underneath the weighting bar (Fig. 2f). The corresponding attribute is removed from the table and the scores are re-calculated excluding this property. Filtered elements become visible in the form of green buttons. Clicking this button adds the attribute back into the ranking view. Due to the color encoding user can quickly grasp interrelations of the attribute and its contribution in the stacked bar of the scores or the weighting bar.

4.4 Code samples

In this chapter, we want to present the code of our implementation.

4.4.1 ELM Architecture

The following lines describe the model of our ranking view.

```
type Table =
    {
      header : Map<string, Attribute>
      weights : hmap<string, float>
      rows : array<Row>
      visibleOrder : List<string> //holds
       ↪ attribute names that should be
       ↪ displayed in corresponding order
      showOptions: bool
      colors : Map<string, C4b>
      dragedAttribute: Option<string>
      weightingFunction : WeightingFunction
  }
```

Possible actions are described as `Messages`.

```
type Message =
  | SetWeight
  | NormalizeWeight
  | CalculateScore
  //...
```

The following code gives an overview of our `update` function. For brevity, we only show a subset of possible messages. Please note, that all messages create a new immutable model based on the input model.

```
let update (model : Table) (msg : Message) =
    match msg with
        | SetWeight (name, value) ->
            let newWeights = model.weights |>
             ↪ HMap.add name value
            CalculateScore {model with weights =
             ↪ newWeights}
        | NormalizeWeight ->
            let visibleAttributes =
             ↪ getVisibleAttributes
             ↪ model.header model.visibleOrder
            let weights =
                visibleAttributes
                  |> List.map (fun attribute  ->
                  (attribute.name, (1.0 / float
                   ↪ (visibleAttributes).Length)))
                  |> Map.ofList
            CalculateScore { model with weights
             ↪ = HMap.ofMap weights}
        | CalculateScore ->
          CalculateScore model
```

Our view is implemented as an incremental *HTML* table. The header contains three rows, holding the histograms, the title of the corresponding attribute and the optional weighting-options including the optimum preference. The visualization of the table rows depends on the data type of the attribute.
The following code shows the creation of the header.

```
let view (model : MTable) =
    Incremental.table (*styling*) <|
        // adaptive list builder enables
         ↪ incremental rendering
        alist {
            // initialize model attributes
            yield thead [] [
                yield drawHistogams headers
                 ↪ visibleOrder rows colors
                yield drawHeaderWithSorting
                 ↪ headers visibleOrder colors
                match showOptions with
                    | false -> ()
                    | true -> yield drawOptions
                     ↪ headers visibleOrder
                     ↪ model.weights
                    ]
        }
```

Again, please note that, for each generated model, the view function needs to be re-executed from scratch. This results in clear but also inefficient code.

4.4.2 Incremental Rendering

A solution for an efficient and clean, purely functional rendering is the incremental approach. To enable incremental rendering we replaced basic structures with adaptive ones. An example is the creation of a *div* and its attributes. In our first implementation, we hand over the attributes as a list into a simple *div*.

```
let attributes =
  [
      clazz selected;
      onMouseMove (fun _ -> Highlight
       ↪ (key, true));
```

```
      onMouseLeave (fun _ -> Highlight
       ↪  (key, false))
      style (sprintf "height: 100%%;
       ↪  width: %.2f%%; background: %s;
       ↪  float: left" width color);
  ]
div [attributes][]
```

For the incremental approach we use an `amap` and hand it over to the `Incremental.div`.

```
let attributes =
  amap {
      yield clazz selected;
      yield onMouseMove (fun _ -> Highlight
       ↪  (key, true));
      yield onMouseLeave (fun _ -> Highlight
       ↪  (key, false))
      yield style (sprintf "height: 100%%;
       ↪  width: %.2f%%; background: %s; float:
       ↪  left" width color);
  } |> AttributeMap.ofAMap
Incremental.div attributes AList.empty
```

As we can see the code stays at the same syntactical level and changes do not require extensive refactorization.

5 Evaluation

Our implementation allows users to analyze and compare ranking data in a user-friendly way. We optimize the rendering performance of the ranking view to support high-frequent user interactions.

5.1 Design requirements

In our tabular visualization, the user can easily grasp the rank of an individual item by sorting according to the score. This fulfills R1. The stacked bar visualizes the relative contribution of each attribute, which solves the problem of R2 and R3. Highlighting of the bar while hovering over the associated header clarifies this. The length of the stacked bar represents the overall score and makes it easy to compare.
To visualize the distribution of an attribute among the items, which is part of R2, we incorporate a histogram into the header. Weights can be set by the user in two ways: manually, by typing the relative proportion of the attribute into the field in the header (R3), or interactively, by dragging the weighting bar (R3 and R4). Using the `normalize weight` button underneath the weighting bar, the weights can be reset to uniform weightings. Every weight adjustment triggers an instant adaption of the ranking view. The user gets immediate visual feedback on how the changes of his or her preferences affect the ranking, as required in R4.
To further refine weights we enabled filtering of the attributes per mouse click on the corresponding buttons. By clicking on the label in the header, the user can sort scores or the values of an attribute in ascending or descending way, illustrated by a small down- or upward pointing arrow (see the score label in Fig. 2 or Fig. 4). By enabling filtering and sorting we also met R5.

5.2 Performance optimization

The computer used for the performance measurements has following specifications:

- CPU: Intel® Core™ i5-4690K CPU @ 3.50GHz, 3501 Mhz, 4 Cores
- GPU: GeForce GTX 1070 8GB GDDR5
- RAM: 16.0 GB
- Operation system: Windows 10

Our first implementation uses non-incremental structures like the map `hmap<'k, 'v>`. When an element in this map is changed, the whole structure of the `hmap` is updated. As a consequence, the dependent UI elements are also updated. This results in a distinctly negative impact on the performance of the visualization. To illustrate the benefit of incremental rendering we provide the same implementation with the adaptive structures like `amap<'k, 'v>`. This requires only small changes as shown in our code samples in 4.4.2. We automated the dragging of our weighting bar for repeatability and measured the time how long the updating and rendering of the ranking view takes. Our results are as follows:
Our first implementation took on average 0.2912 ± 0.0647 seconds per update. During usage, the lag is clearly noticeable. After swapping the map for its incremental version the average update time decreased to 0.0590 ± 0.0224 seconds. With minor changes, we were able to improve our performance by a factor of 5.

6 Conclusions

In this work, we provided an interactive solution for a multi-attribute ranking view. We implemented the user interface using *Aardvark.Media* which is based on the *ELM* architecture and uses a functional approach with F#.

We evaluated requirements, which had to be met to enable an user to quickly and easily adapt the rankings to her or his needs. The ranking depends on the values and the weights of the single attributes. An interactive approach in changing those weights has been implemented and visual feedback of those changes is immediately reflected by an updated ranking view.

The performance could be significantly improved by using the incremental data structures provided by the *Aardvark platform*.

References

[1] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, August 2013.

[2] Mike Bostock. D3 - data-driven documents. https://d3js.org, 2017. [Online; accessed 26-March-2020].

[3] Mark Harrower Cynthia Brewer. Colorbrewer 2.0. http://colorbrewer2.org/#type=diverging&scheme=PuOr&n=11, 2017. [Online; accessed 07-February-2020].

[4] Evan Czaplicki. Elm vegalite. https://package.elm-lang.org/packages/gicentre/elm-vegalite/latest/VegaLite#2-specifying-the-data-to-visualize. [Online; accessed 07-February-2020].

[5] ELM-lang.org. The elm architecture. https://guide.elm-lang.org/architecture. [Online; accessed 07-February-2020].

[6] Stephen Few. Show me the numbers. *Analytics Pres*, 2004.

[7] Samuel Gratzl, Alexander Lex, Nils Gehlenborg, Hanspeter Pfister, and Marc Streit. Lineup: Visual analysis of multi-attribute rankings. *IEEE transactions on visualization and computer graphics*, 19(12):2277–2286, 2013.

[8] Matthew Griffith. Why elm? https://www.oreilly.com/library/view/why-elm/9781491990728/, 2017. [Online; accessed 07-February-2020].

[9] Georg Haaser, Harald Steinlechner, Stefan Maierhofer, and Robert F. Tobler. An incremental rendering vm. In *Proceedings of the 7th Conference on High-Performance Graphics*, HPG '15, pages 51–60, New York, NY, USA, 2015. ACM.

[10] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[11] Jock Mackinlay. Automating the design of graphical presentations of relational information. *Acm Transactions On Graphics (Tog)*, 5(2):110–141, 1986.

[12] Yvonne Jansen Petra Isenberg, Pierre Dragicevic. Project datasets. https://perso.telecom-paristech.fr/eagan/class/igr204/datasets, 2018. [Online; accessed 07-February-2020].

[13] Aardvark Platform. aardvark-platform/aardvark.rendering. https://github.com/aardvark-platform/aardvark.rendering, 2015. [Online; accessed 07-February-2020].

[14] Aardvark Platform. Aardvark - an advanced rapid development visualization and rendering kernel. https://www.vrvis.at/research/projects/aardvark/, 2017. [Online; accessed 07-February-2020].

[15] Aardvark Platform. Aardvark.media - incremental rendering. https://github.com/aardvark-platform/aardvark.docs/wiki/Incremental-Rendering, 2017. [Online; accessed 07-February-2020].

[16] Aardvark Platform. Aardvark.media - documentation. https://github.com/aardvark-platform/aardvark.docs/wiki/Aardvark.Media, 2018. [Online; accessed 2-February-2020].

[17] Aardvark Platform. Aardvark.media - documentation. https://github.com/aardvark-platform, 2018. [Online; accessed 26-March-2020].

[18] Ramana Rao and Stuart K Card. The table lens: merging graphical and symbolic representations in an interactive focus+ context visualization for tabular information. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 318–322. Citeseer, 1994.

[19] W. Scott. Why use f#? https://fsharpforfunandprofit.com/why-use-fsharp, 2018. [Online; accessed 05-February-2020].

[20] Chris Smith. *Programming F#: A comprehensive guide for writing simple code to solve complex problems*. " O'Reilly Media, Inc.", 2009.

[21] Harald Steinlechner, Georg Haaser, Stefan Maierhofer, and Robert F. Tobler. Attribute grammars for incremental scene graph rendering. In *Proceedings of the 14th International Conference on Computer Graphics Theory and Applications*, GRAPP 2019, pages 77–88, 2019.

[22] vega.github.io. Vega 2 - documentation. https://github.com/vega/vega/wiki/Documentation. [Online; accessed 07-February-2020].

[23] Nick Qi Zhu. *Data visualization with D3. js cookbook*. Packt Publishing Ltd, 2013.