TU WIEN Informatics

A Modular Domain-Specific Language for Interactive 3D Visualization

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Dominik Scholz, BSc
Matrikelnummer 01527434

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Dipl.-Ing. Harald Steinlechner

Wien, 10. Mai 2021

_____   _____
Dominik Scholz   Eduard Gröller

TU WIEN Informatics

A Modular Domain-Specific Language for Interactive 3D Visualization

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Dominik Scholz, BSc

Registration Number 01527434

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Assistance: Dipl.-Ing. Harald Steinlechner

Vienna, 10th May, 2021

_____   _____
            Dominik Scholz   Eduard Gröller

Erklärung zur Verfassung der Arbeit

Dominik Scholz, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 10. Mai 2021

_____
Dominik Scholz

Acknowledgements

First and foremost, I want to thank Harald Steinlechner for giving me the opportunity to work on such an exciting project, as well as his guidance and support. I also want to thank Eduard Gröller for his valuable advice on my thesis.

This thesis was written during the COVID-19 pandemic, which tested everyone's resilience. Therefore, I am grateful for all the people that kept me sane during this time. I also thank my family, which made my whole university endeavour possible in the first place. And finally, special thanks to my friends who joined and supported me on this adventure.

Kurzfassung

Datenvisualisierung ist ein wichtiger Teil vieler Forschungsgebiete. Die Erstellung von 3D Visualisierungen ist meistens so komplex, dass dafür Visualisierungsexperten benötigt werden. In dieser Arbeit stellen wir ein System vor, mit dem Visualisierungen ohne Fachkentnisse in Computergraphik erstellt werden können. Stattdessen werden Visualisierungen mittels einer domänenspezifischen Sprache definiert. Anwender können mit dieser Sprache auf einfache Weise selbstständig 3D Visualisierungen erzeugen. Die Sprache ermöglicht es, alle Aspekte der Visualisierungs-Pipeline, von der Datentransformation bis zum Rendern, zu konfigurieren. Die Konfiguration der Visualisierungs-Pipeline wird dabei durch einen Mechanismus ermöglicht, der es erlaubt, Erweiterungsmodule zu definieren. Unser System enthält auch vordefinierte Module, welche gängige Visualisierungstechniken, wie etwa Volumen-, oder Punktwolken-Visualisierung, ermöglichen. Zusätzlich können Anwender neue Module erstellen um andere Visualisierungstechniken zu unterstützen.

Unsere Visualisierungsbeschreibung ist eine deklarative Sprache, welche in einem Klartextformat definiert wird. Dieses Format erlaubt einen einfachen Austausch sowie Speicherung von Visualisierungen. Zusätzlich erlaubt es den Einsatz unserer Sprache innerhalb "Computational Notebooks", welche Wissenschaftlern erlauben, Resultate ihrer Arbeit in replizierbarer Weise zu speichern. Das Format ermöglicht ist außerdem ein weitverbreitetes Austauschformat im Internet, wodurch unsere Visualisierungsbeschreibung leicht transferiert werden kann.

Interaktivität ist ein weiterer wichtiger Aspekt von 3D Visualisierungen, der auch von unserer Sprache unterstützt wird. Interaktionen in unserem System erzeugen Nachrichten, welche die Visualisierung nutzen kann, um sich selbst zu modifizieren. Das Nachrichtensystem ermöglicht es außerdem, dynamische Daten zu visualisieren. Modifikationen an der Visualisierung werden vom System automatisch analysiert, um die tatsächlichen Änderungen zu berechnen. Daraufhin wird nur der Teil an Grafikbefehlen ausgeführt, der benötigt wird, um die Änderungen zu bewirken. Mit diesem Ansatz werden Grafikkartenressourcen wie Shader, Buffer und Texturen nur dann aktualisiert, wenn es notwendig ist. Diese Vorgehensweise erhöht die Renderleistung und verbirgt Implementierungsdetails vom Nutzer wodurch unser System zugänglicher wird.

Abstract

Visualizing data is an integral part of many scientific disciplines. Due to the complexity of 3D visualizations, their creation usually requires visualization experts. This thesis proposes a system for specifying visualizations without requiring in-depth knowledge about computer graphics. Instead, visualizations are defined in a domain-specific language. This language enables users to easily create and edit 3D visualizations on their own. The language provides configuration capabilities for all aspects of the visualization pipeline, ranging from data transformation up to rendering. Configuring the visualization pipeline is facilitated through a versatile extension mechanism, allowing the definition of modules. Pre-defined modules provide a wide range of 3D visualization techniques, such as volumes or point clouds. Additionally, users can create new modules to support other visualization techniques.

Our visualization description is a declarative language specified in a human-readable text format. Due to the text format, visualizations created with our language are simple to store and share. The format also aids the use of our language inside notebooks, which enables scientists to persist their computational results in a replicable fashion. Furthermore, using a text-based format is a natural method for communicating information over the web. Hence, it makes our visualization description easily transferable.

Interactivity is another crucial aspect of 3D visualization that our language supports. User events emit messages that the visualization can then use to modify itself. The message system also makes it possible to visualize dynamic data. Modifications of the visualizations pass through our system to automatically analyze the changes. Then just the needed subset of rendering calls to achieve the change execute. With this approach, GPU resources like shaders, buffers, and textures only update when required. This strategy enhances rendering performance and hides implementation detail from the user making the system easier approachable.

Contents

Introduction

Visualizations - visual representations of data - are vital tools for conveying information. They create insight and help in detecting otherwise occluded patterns in raw data. Moreover, visualizations are an excellent method for sharing knowledge. Therefore, they are a popular choice whenever data is communicated, like in newspapers or presentations. Classical media often forces visualizations into static views. Computers, on the other hand, extend visualizations by adding the capability to interact with them. Interactivity aids in several use cases, for instance, in data exploration, particularly when visualizing three- or higher-dimensional data. In such cases, no single viewpoint might educate the viewer about the complete dataset. Solving this problem requires interactive interfaces that let users manipulate the visualization, for example, by changing camera perspectives or switching aggregations.

Sharing knowledge is an essential task that interactive visualizations are well-suited for accomplishing. Therefore, our scientific question is:

What is the easiest way to share an interactive (3D) visualization?

For sharing insight via interactive visualizations between non-computer scientists, low threshold accessibility is critical. Arguably in today's age, sharing ideas is prominently facilitated through the internet. Therefore, this work aims to provide an easy method for sharing interactive visualizations in web browsers as a platform-independent format.

Numerous visualization libraries already exist in the web ecosystem. A modern family of languages, comprising Protovis [BH09], D3 [BOH11], and Vega [SRHH15] (Figure 1.1), showcased the benefits of specifying visualizations for the web declaratively (Chapter 2). Specifying visualizations in that manner allows users to focus on how visualizations should look and behave rather than on their particular execution on the computer. Consequently, such languages enable people without in-depth knowledge of computer graphics to plot interactive visualizations quickly. Furthermore, Vega demonstrates the advantage of describing visualizations inside self-contained files. Vega descriptions provide the full

information needed to display an interactive visualization inside a human-readable text file (JSON). As a result, the file operates as a standardized communication medium for sharing visualizations.
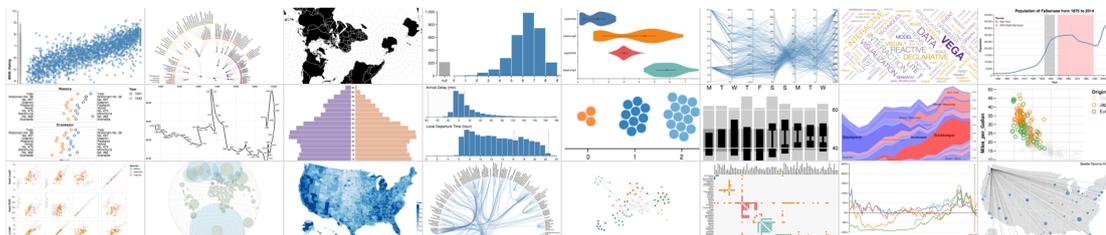

Figure 1.1: Various charts created with Vega, which uses D3 for rendering.
Source: https://vega.github.io/vega/

Another motivation for our system is the emerging trend of publishing research interactively. For example, researchers from CERN publish papers using their SWAN system [CER20], and machine learning researchers use the interactive Distill journal [Dis20] for publications. The idea of combining explanatory text and computation instructions traces back to the concept of *literate programming*. Literate programming is a term coined by Donald Knuth describing the idea of handling computer programs as works of literature [Knu84]. In that sense, programs should resemble essays of the creator's thought process consisting of descriptions in natural language as formatted text annotating the actual source code. The recently popularized notebook environment [KRKP+16] extended Knuth's idea to form literate computing, whose idea is to form *"documents integrating prose, code, and results"* [KRKP+16]. These notebook documents then achieve a so-called *computational narrative* by combining human language, live code, and its results. In this regard, notebooks take the original idea of literate programming a step further by including the computations' results. Jupyter notebooks focus on the scientific concepts of collaboration and reproducibility. These concepts are not only essential concepts among academics, but also in the much broader community of data analysts ranging from journalists to policymakers [PG15]. Hence, the notebook format has immense potential for influencing those disciplines. However, in all of those groups, the need of novice-friendly methods for describing visualizations in textual format arises. Declarative languages such as Vega-lite can fill this need for 2D visualizations.

Although Protovis, D3, and Vega offer excellent tools for creating 2D visualizations, they do not support 3D visualizations. Compared to 2D visualizations, 3D visualizations are even harder to create. Efficient 3D visualizations demand a great focus on optimizations and low-level concerns such as GPU memory management or state changes. These concepts require considerable knowledge about computer graphics making the creation of visualizations for people from other scientific fields challenging.

Many currently used 3D visualization libraries lack custom interaction possibilities. In general, they mostly only support a basic, predefined set of options for navigating or interacting with the visualizations. Also declarative languages like VRML or its successor

X3D [Con20] only offer a limited set of interaction primitives without extension possibilities. 3D visualizations encompass a broad range of different visualization techniques like volumes, point clouds, or flows. These techniques already rely on distinct interaction methods. Additionally, specific use cases or visualizations outside of these basic categories often require custom interaction methods. Due to these application-specific needs, it would be challenging to fit all possible interaction methods inside a single monolithic library. Instead, a versatile visualization toolbox should offer a mechanism for extending interaction methods easily. To sum up, in this theses we have the following contributions:

- We formulate a declarative, domain-specific language (DSL) for interactive 3D visualizations (Chapter 3). Moreover, the language can configure all stages of the visualization pipeline. The language defines visualization descriptions inside a human-readable text format. Therefore, users can easily configure the visualization and create custom interaction capabilities without requiring computer graphics knowledge (Section 6.1.1).

- We propose a modular system for computing visualizations using this language. Just as our DSL, our system also leverages declarative design principles, efficient functional datastructures and enables interfacing with imperative programming web-APIs. With this approach, the system computes updates to the visualization using resources sparsely.

- Combining the system and the language, we develop a visualization-creation environment (see Figure 1.3). This web-based application allows visualization creators to quickly prototype visualizations and change the parameters of shared visualizations on the fly.

- Showcasing the language's expressiveness, we present different demo-visualizations that highlight various aspects of the system, such as the creation of custom interactions (Section 6.1.1), volume rendering (Section 6.1.2), or the integration of external libraries (Section 6.1.3). These demos do not require any knowledge of low-level graphics programming from the user so that people outside the computer-graphics community can create and configure them.

- Lastly, we show the run-time overhead introduced by our approach compared to traditional solutions and analyze the performance characteristics (Section 6.2).

```
In [4]:  TCanvas c1;
         f2.SetLineWidth(1);
         f2.SetLineColor(kBlue - 5);
         f2.Draw("Surf1");
         auto Xaxis = f2.GetXaxis();
         auto Yaxis = f2.GetYaxis();
         auto Zaxis = f2.GetZaxis();
         Xaxis->SetTitle("X Title"); Xaxis->SetTitleOffset(1.5);
         Yaxis->SetTitle("Y Title"); Yaxis->SetTitleOffset(1.5);
         Zaxis->SetTitle("Z Title"); Zaxis->SetTitleOffset(1.5);
         dte.Draw("P0 Same");
```

Display the 2D graph in the notebook.

```
In [5]:  c1.Draw();
```
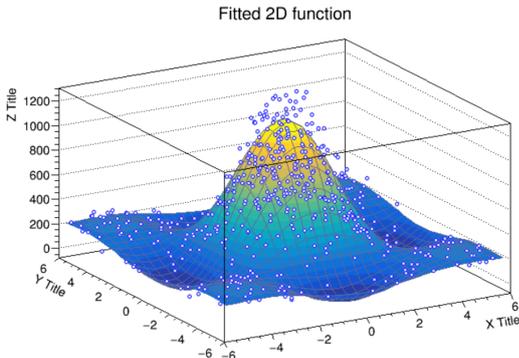

Make the x and y projections.

```
In [6]:  gStyle->SetPalette(kBird);
         TCanvas c_p("ProjCan", "The Projections", 1000, 400);
         c_p.Divide(2,1);
         c_p.cd(1);
         dte.Project("x")->Draw();
         c_p.cd(2);
         dte.Project("y")->Draw();
```

(a) An example of the SWAN system, which uses the ROOT framework [BRC$^+$19] in conjunction with Jupyter notebooks.
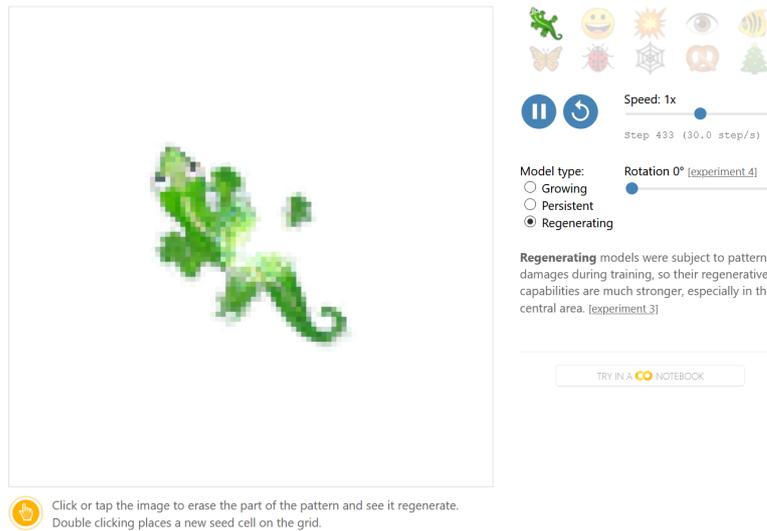
(b) An interactive Distill publication from Mordvintsev et al. [MRNL20].

Figure 1.2: Different uses of the notebook environment.

3D Vis

example: volume

visualization description

```
1  {
2    "volume": {
3      "loadPath": "../data/stagbeetle832x832x494.dat",
4      "colorRange": [0, 1],
5      "renderStyle": "iso",
6      "threshold": 0.2,
7      "texture": "color"
8    },
9    "texturePaths": {
10     "color": "../data/cm_viridis.png",
11     "gray": "../data/cm_gray.png"
12   },
13   "camera": {
14     "position": [14, 54, -20],
15     "rotation": [-110, 13, 147],
16     "zoom": 4
17   },
18   "scene": {
19     "grid": true,
20     "backgroundColor": [0.1, 0.1, 0.1]
21   },
22   "pipeline": [
23     "scene",
24     "volume-loader",
25     "texture",
26     "orthographic-camera",
27     "volume",
28     "render",
29     "orbit-controls",
30     "source"
31   ]
32 }
```
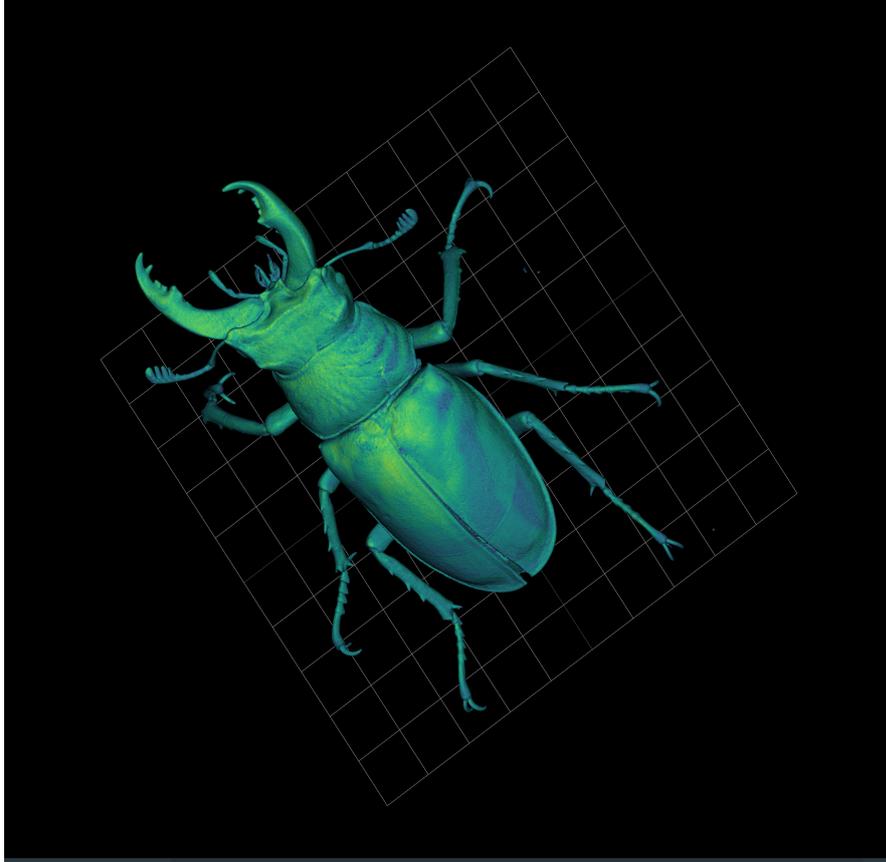
Figure 1.3: The visualization-creation environment. The DSL on the left describes the volume visualization on the right, which shows the stag beetle dataset from Gröller et al. [GGK05]. Textual changes in the DSL (left) are reflected in the visualization (right) and interactions with the visualization automatically update the DSL.

Related Work

This chapter presents works that directly influenced our visualization design and methods solving similar problems as our architecture.

In general, there exists a vast number of visualization tools. Hence we first focus on tools that aim to provide visualization capabilities from a language-centric viewpoint, thus constitute domain-specific languages.

Next, we highlight a family of languages that focus on the ease of creating declarative languages for the web, which are therefore easily shareable. Subsequently, we look at systems for the web that integrate performant recalculations and graphical interfaces.

Lastly, we take a look at systems proposing a solution for similar problem spaces as our work. Although there are many visualization systems for domain experts. This sections focuses on two particular visualization systems that are similar to our approach and also enable visualization designers to create custom applications for domain experts without computer graphics or visualization knowledge.

2.1 Domain-Specific Languages for Visualization

Domain-Specific Languages (DSLs) are programming languages constructed around a specific domain. In contrast to general-purpose languages, the aim of DSLs is not to offer a language that can be universally applied for a broad range of problems, but to focus on a small, well-defined set of problems and handle those in an expressive manner. Rautek et al. identifies a hierarchy of possibilities interfacing with applications (see Figure 2.1) [RBGH14]. The interfaces in the hierarchy are ordered by increasing flexibility and required experience. The range starts with *turn-key* systems, which are applications exposing only some parameters to the user, often through graphical interfaces. They allow the use and configuration with minor user experience. At the end of the range, direct modifications in source code and the use of libraries are located, requiring

expert programmers. In this hierarchy, DSLs bridge the gap between the two extremes: little experience/low flexibility and high experience/maximum flexibility. Hence, the general application area for DSLs lies in providing an interface for high-level application programmers (instead of rendering experts) as well as domain experts and scientists with basic programming knowledge.
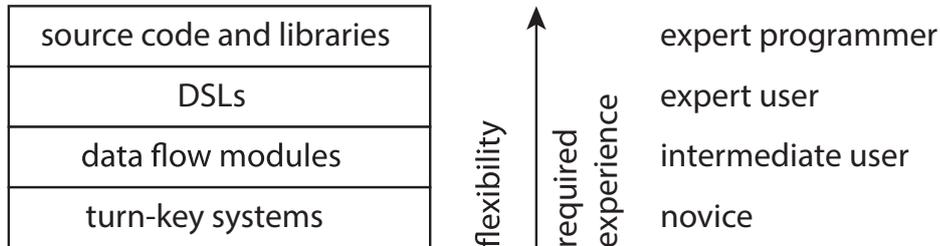

Figure 2.1: Experience hierarchy of configurable software.
Source: [RBGH14]

DSLs for visualization already exist for quite some time, with Elliot and Hudak [EH97] as well as Stolte et al. [STH02] being noteworthy mentions. The following sections cover modern DSLs, that draw inspiration from these early approaches.

2.1.1 VisLang

VisLang is a system for integrating multiple DSLs into a common visualization application. Supporting multiple DSLs enables the use of different programming paradigms in the same environment. Therefore, the best-suited programming paradigm for the current task can be selected. Whereas declarative languages, like D3 [BOH11] and Vega [SRHH15], are popular for specifying layouts and structure, imperative languages might be better suited for rendering applications or functional languages for streamlined signal processing.
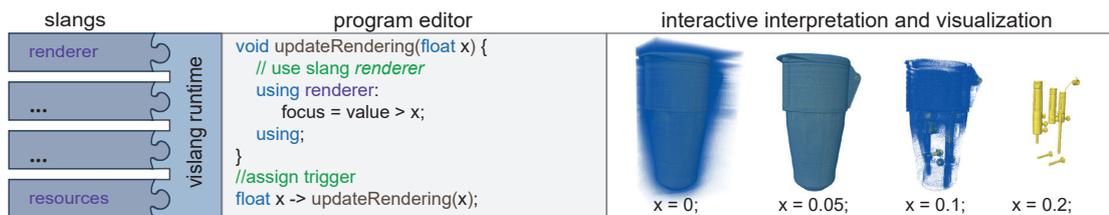

Figure 2.2: Example of how slangs (left) interact through the vislang runtime with the program editor (middle) to produce an interative visualization (right).
Source: [RBGH14]

To the user, the VisLang runtime appears as a unified programming environment for all the integrated DSLs. These DSLs, which are extending VisLang, are called *slangs*. Users can access these slangs by stating the `using` keyword (see Figure 2.2) followed by

the name of the slang. Inside this `using` scope, the slang's language is active. In the example shown in Figure 2.2, the slang *renderer* allows changing visualization parameters. The encapsulating function `updateRendering` is defined, and assigned as a trigger to the variable `x`. In that way, when the user changes `x`, the `updateRendering` function is executing changing the `focus` value inside the *renderer* slang. This mechanism lets the user tune the visualization by exploring different values for `x` and observing the visualization's live updates (Figure 2.2 right).

Rautek et al. showcase the flexibility of VisLang by presenting three different slangs with distinctive programming paradigms. The *Volume Predicate* slang is a **procedural** language, that can specify logical queries on a 3D voxel data structure resulting in a set of binary voxels. *Vlabel Visualization* is a **declarative** slang, which gets compiled to an OpenCL program. It takes a volume and a label as input with which the user can specify the appearance of the volume declaratively. Lastly, the *Map-Reduce* slang is a **functional** language providing the user with a way to efficiently perform volumetric queries on the GPU, like the counting of voxels emitted from the *Volume Predicate* slang.

Usually, the process of creating novel DSLs is rather intricate. It requires lexing, parsing, generation of an abstract syntax tree, and finally interpretation of the DSL. Therefore, VisLang offers an extension mechanism using a meta-language. This language makes it possible to specify new slangs by stating parameters and grammar rules. Additionally, the meta-language also features an inheritance/template mechanism.

2.2 Visualization Systems for the Web

2.2.1 ProtoVis

ProtoVis [BH09][HB10] is a tool for streamlining the creation of visualizations. Modern charting software allows users to import data and quickly create diagrams for it from a predefined set via user interfaces. These predefined chart types, however, are highly limited. When custom visualizations, outside of this predefined set, are needed, users often resort to creating them manually in vector-graphics drawing applications. This process is time-consuming, and the static visualization prohibits interactivity as well as the ability to update it with live data. ProtoVis presents a solution that lies between those two extremes by enabling quick charting while providing maximum flexibility and retaining interactivity. ProtoVis is a declarative DSL for interactive visualization. Thus it describes what visualizations should look like instead of defining how they should be computed. This declarative approach splits the specification from the execution, permitting various optimizations in the needed translation step. Heer and Bostock note that previous declarative DSLs like ggplot2 [Wil12][Wic10] or VizQL [STH02] feature high levels of abstraction, enabling fast data analysis, but do not support precise control over graphics and interaction. ProtoVis is an *embedded* DSL. Therefore, it is not a standalone language but programmed and used from within a host language. The original ProtoVis implementation was done in JavaScript [BH09], followed by an implementation in Java by the same authors [HB10]. ProtoVis constructs visualizations by combining

graphical marks hierarchically. Graphical marks are simple, visual primitives like lines, bars, or labels (see Figure 2.3). Optionally, marks can specify anchors, which are named positions that can be referred to by related marks. Marks are placed inside *panels* that can be positioned and replicated. The marks encode data through dynamic properties such as color, position, or size. The data is provided to the marks and then delegated to the properties by calling the `data()` function.
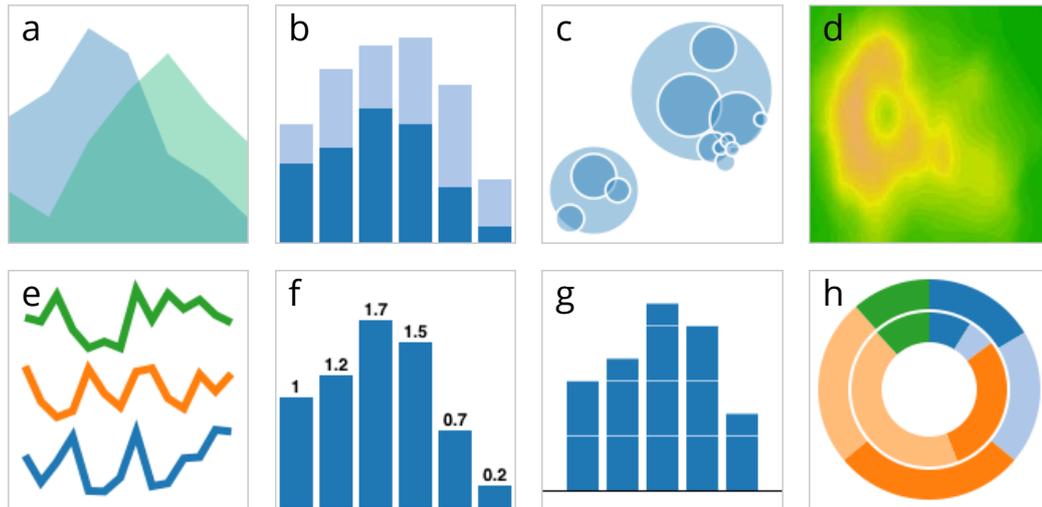

Figure 2.3: Examples of graphical marks from ProtoVis: (a-h) Area, Bar, Dot, Image, Line, Label (and Bar), Rule (and Bar), Wedge.
Source: [HB10]

Encoding is provided by anonymous functions (or static variables) that determine the properties' value — describing these properties in such a way results in a call chain, which gets translated into a scene graph. This chaining results in the capability of properties to depend on each other's value. Property functions can access information through various built-in variables as well as global and bound variables inside the current scope. Important built-in variables are: The `this` object, referring to the scene graph in general, the `parent` object referencing the panel containing the current mark, and the `sibling` object, referencing the previous panel. Event handlers can be specified to include interactivity. These event handlers listen to user interactions and execute a function that can change properties of the interacted mark by using the `this` keyword as displayed in Figure 2.4. New mark types can be created from existing mark types by using prototypical inheritance. This mechanism allows new mark types to retain the general functionality of the base type and only override or add the new functions.

The Java implementation of ProtoVis differs from the JavaScript implementation in that it does not support anonymous functions directly. For that reason, every property type

```
var vis = new pv.Panel().width(w).height(h);
vis.add(pv.Bar)
  .data(someData)
  .left(() => this.index * 5)
  .bottom(0)
  .width(3)
  .height(d => d.y * 50)
  .def("i", -1)
  .fillStyle(() => i == this.index ? "0xff0000" : "0x1f77b4")
  .event("mouseover", () => i(this.index))
  .event("mouseout",  () => i(-1))
vis.render();
```

(a)

```
Scene vis = new Scene().width(w).height(h).scene()
vis.add("Bar")
  .data(someData).datatype(Point2D.class)
  .left("{{index * 5}}")
  .bottom(0)
  .width(3)
  .height("{{data.getY() * 50}}")
  .def("i", -1)
  .fill("{{Fill.solid(i == index ? 0xff0000 : 0x1f77b4)}}")
  .mouseover("{{i(index).update()}}")
  .mouseout("{{i(-1).update()}}");
vis.update();
```

(b)
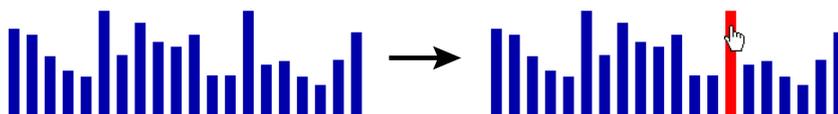
Figure 2.4: Example definition of a simple, interactive bar chart using ProtoVis JavaScript (a) and Java (b).
Sources: [BH09][HB10]

requires the creation of a class implementing the `Property` interface. To provide the same ease of writing property functions as in the JavaScript implementation, ProtoVis Java offers users a simpler alternative: Anonymous property functions are passed as strings to the property function call. The string contains regular Java code enclosed by double curly brackets. This code permits omitting the return statement. Additionally, variables like `data` and `index` are in scope, with the former referencing the complete

provided data object and the latter referring to the index of the current graphical mark. This data object does not have to be a static array but can be an arbitrary object implementing the *Iterable* interface, allowing high flexibility in the choice and structure of the provided data.

From an implementation standpoint, the creation of visualizations with ProtoVis follows a pipeline containing six successive stages: **Bind**, **Build**, **Evaluate**, **Interpolate**, **Render**, and **Event**. At first, the **Bind** stage traverses the specifications of the graphical marks. In this step, property definitions of each mark are resolved. Three different types of properties are distinguished: *constant* properties just have a static property value, *compiled* properties refer to predefined compiled functions, and *dynamic* properties, which use the string specification of anonymous functions and have to be either interpreted or compiled before the property evaluation. Similar to the prototype inheritance in the JavaScript version, ProtoVis Java traverses the inheritance hierarchy of the marks to find inherited property definitions. If the encountered mark is the first one of its type found by the build process, ProtoVis constructs an *Evaluator* for the mark. This evaluator is capable of compiling dynamic properties on-the-fly if the platform running ProtoVis offers compiler access or interpretation if it does not. Next is the **Build** phase. Its purpose is the creation of the scene graph. For this, ProtoVis iterates through the provided data and emits an associated scene graph node for every data point. Subsequently, the **Evaluate** stage starts. In this stage, the properties' values for every scene graph node is calculated by executing the property function. The **Interpolate** stage comes next. Here, ProtoVis interpolates property values for scene graph nodes that are part of animations. Afterwards comes the **Render** stage. In this stage, ProtoVis renders the scene graph with all of its nodes to a display. The actual rendering code is accessed through an interface, which lets ProtoVis provide different platform-specific renderers. The final stage is the **Event** stage, where interaction events for registered callback functions are triggered so that the visualization is able to react to user interaction. To speed up the pipeline processing, ProtoVis utilized parallelization for the build and evaluate stages.

Summarizing, ProtoVis offers a declarative approach of specifying visualizations. This gives advantages such as an intuitive and concise specification but results in higher pre-processing work than direct draw calls in conventional visualization programming.

2.2.2 Data-Driven Documents

Data-Driven Documents (D3) is a popular tool for 2D visualization on the web. It is based on its predecessor ProtoVis [BH09][HB10], a declarative DSL. As mentioned in Section 2.2.1, ProtoVis uses abstraction concepts called graphical *marks*, like a `Line` or `Wedge`. These marks describe the visual appearance of diagrams. They can be adapted using inheritance in a prototype-pattern fashion, resulting in highly reusable components. D3 does not follow this design idiom. Instead, D3 works much like document transformation tools, like jQuery [Fjc21] or XSLT [W3C17], working directly on the underlying document primitives. As D3 is a JavaScript library for the usual web stack, these primitives are DOM nodes. In practice, these DOM nodes are usually either

common HTML nodes directly or SVG nodes, which are also XML based and can be integrated into HTML documents and, therefore, into the DOM. Utilizing this native representation directly, instead of an abstractly defined scene graph, allows the use of basic, already existing tooling and debugging capabilities like the developer tools of modern browsers, as well as preexisting documentation of the underlying primitives—in the case of D3 HTML and SVG elements.

D3's main entry point is the *selection* with either the `select` or `selectAll` functions called on the library's namespace `d3`. This command takes a selection predicate using the W3C Selection API as input. CSS and jQuery also use the same API. The selection call returns an object encapsulating the specified nodes. *Operators* like `style`, `attr`, or `html` are called on this encapsulating object to change the appropriate property of the selected nodes. The selection object also enables the execution of an integral part of D3: *Data joins.* Data joins bind arbitrary data, bundled in an array, with the selected nodes. The bound data is then passed through functional operators with the first parameter being the whole data array, and the second being the index of the node. With this setup, it is easy to define expressive access functions to the dataset that, for example, transform the data to another range. These access functions are then used in the before mentioned operator calls that modify the properties of the nodes. If the underlying data changes, the nodes are updated dynamically to reflect the new state. For this, three sub-selections are created for which handlers are executed: `update` is called on data-node pairs, for which the data is changed, `enter` is called on incoming data, which does not yet have a corresponding node, and `exit` is called on nodes, which do not have corresponding data items after the update. The `enter` callback emits new placeholder nodes, which can be added using the `insert` or `append` function, while the `exit` callback returns a list of the nodes that now do not have related data so they can be removed from the DOM. If a node selection is supplied with data—that is, after a data join—it can be further refined using the `filter` or `sort` calls, which update the selection according to the node's corresponding data values.

For the interactivity of the visualization, D3 exposes the typical native interaction events on the DOM nodes through the `on` callback. When a user interacts with a DOM node, the corresponding functions are executed for every node. D3 passes the event type as well as the complete data array and the node's index to these callbacks, allowing visualization or data changes. It is possible to define animations that act on such data changes using the `transition` operator. This operator enables the definition of new attributes or style values for the nodes. Additionally, delay, duration, and easing functions, similar to the specification of animations in CSS, are configurable. When transitioning between values, so-called *interpolators* can change how the numbers are interpolated. For example, an interpolator might interpolate coordinates not in a Cartesian coordinate system, but in a polar coordinate system. Bostock et al. [BOH11] note that by using their unified timer queue for scheduling transitions, it "easily scales to thousands of concurrent timers".

On top of D3's kernel, modules provide reusable components for commonly used tasks. One such module, *shapes*, is used to provide specialized shapes like the graphical primitives

of ProtoVis. This module supports the creation of parameterized shapes that do not directly correspond to native SVG nodes but are implemented by `path` elements with a customized `d` attribute. Examples for such shapes include arcs, areas, lines, or scatter-plot symbols. The *scale* module extends the interpolators by providing a structured way of defining different scales such as linear, logarithmic, or ordinal ones. Another module is the *layouts* module, which provides a spatial ordering for the applied nodes. For example, the force layout can be used for self-arranging node-link diagrams by calculating physical force relaxation for the nodes. The stacked layout provides the calculations for stacking nodes on top of each other, according to a common baseline. Lastly, the *behaviors* module eases the implementation of common interaction techniques such as zooming or panning.

In comparison with its predecessor ProtoVis, D3's general design is not based on the specification of static scene graphs in a declarative language but instead uses the DOM as a native scene graph for which D3 specifies transformations. This type of specification gives information over the dependence structure of the nodes' properties, leading to a system where only the data items and nodes that changed their values need to be updated. This stands in stark contrast to ProtoVis, where all nodes have to be updated. In D3, this efficient update system is the reason for making the animation system possible. The declarative notation of the scene graph in ProtoVis also has another drawback. The abstract declaration of the scene graph has to be interpreted, and the

(a)
```
new pv.Panel()
  .data([[1, 1.2, 1.7, 1.5, .7]])
  .width(150)
  .height(150)
.add(pv.Wedge)
  .data(pv.normalize)
  .left(75)
  .bottom(75)
  .outerRadius(70)
  .angle(function(d) d * 2 * Math.PI)
.root.render();
```

(b)
```
d3.select("body").append("svg:svg")
    .data([[1, 1.2, 1.7, 1.5, .7]])
    .attr("width", 150)
    .attr("height", 150)
  .selectAll("path")
    .data(d3.layout.pie())
  .enter().append("svg:path")
    .attr("transform", "translate(75,75)")
    .attr("d", d3.svg.arc().outerRadius(70));
```

Figure 2.5: Comparison of drawing a simple pie chart in ProtoVis (a) and D3 (b).
Source: [BOH11]

properties of the nodes evaluated by following the prototype chain of graphical marks. In ProtoVis, this evaluation is delayed to the rendering phase of the system. Instead, D3 evaluates operator calls like `attr` immediately, reducing the internal control flow and computational overhead of the system. [BOH11]

In summary, D3 offers a novel system for displaying visualization integrating directly to the existing web stack ecosystem because it relies on the DOM as a scene graph. Additionally, the choice of declaring data transformations leads to more efficient handling of data changes than previous work.

2.2.3 Vega

Vega [SWH14] is a declarative system for interactive visualizations written in JavaScript, which is built on top of the results of ProtoVis and D3. It provides a *visualization grammar*, which is a JSON based format for specifying interactive visualizations (see Figure 2.6). As in ProtoVis, Vega uses hierarchical graphical marks with associated properties to construct a scene graph that is later rendered. Vega's default render-target is the HTML5 canvas, but it also supports rendering to SVG or server-side rendering. Rendering Vega facilitates D3 in the back. Therefore Vega can be seen as a high-level specification language on top of the low-level D3 framework. In the same vein as ProtoVis, Vega's specification of graphical marks describes how data tuples translate into the mark's properties. Other than in ProtoVis, however, the specification does not rely on anonymous functions, but is entirely described by the static visualization grammar. Vega also follows a bind-build-evaluate pipeline like in ProtoVis. Hence, it traverses the marks specified in the visualization grammar and evaluates their properties along the way. This happens in the *build* and *evaluate* stages. Like in D3, the process of emitting one mark for every data tuple is called a *data-join*. Additionally, Vega sets a status-flag on each mark, depending on whether the associated data tuple was added, removed, or updated.

Reactive Vega

Interactivity in Vega is handled differently to previous declarative models. These previous models often enable interactivity in an imperative approach by handling event callbacks. As a result, users often have to deal with interaction execution details, which quickly becomes an error-prone and complex endeavor. For this reason, Vega treats interaction primitives as first-class-citizens. The following interactivity paradigm is introduced in Reactive Vega [SRHH15], which is now part of the main Vega project. Reactive Vega proposes a system architecture for data streaming that draws from previous works in streaming databases and *Event-Driven Functional Reactive Programming* (E-FRP). This architecture bundles interaction events together with the raw data for the visualization and exposes them under a uniform interface of data streams.

E-FRP [WTH01], popularized by GUI frameworks like Elm [CC13], is a paradigm for modeling value changes using *streams*. These streams are ongoing sequences of events. Further, streams can be composed into signals to build expressions that react to events.

```
{ "data": [{"name": "table", "url": "data/letter_counts.json"}],

  "scales": [{
    "name": "x", "type": "ordinal", "range": "width",
    "domain": {"data": "table", "field": "data.letter"}
  }, {
    "name": "y", "type": "linear", "range": "height",
    "domain": {"data": "table", "field": "data.count"}
  }],

  "axes": [
    {"type": "x", "scale": "x"},
    {"type": "y", "scale": "y"}
  ],

  "marks": [{
    "type": "rect", "from": {"data": "table"},
    "properties": { "update": {
      "x": {"scale": "x", "field": "data.letter"},
      "width": {"scale": "x", "band": true, "offset": -1},
      "y": {"scale": "y", "field": "data.count"},
      "y2": {"scale": "y", "value": 0},
      "fill": {"value": "steelblue"}
    }
  }
}]}
```

Figure 2.6: Vega specification of a bar chart. The data is referenced through an a URL. The scales and marks fields define how data is mapped to the visual representation. **Source:** [SWH14]

Combining signals into expressions forms new dependent signals. E-FRP systems create a dataflow dependency graph encoding these dependent relations. New events enter the streams and travel through the dependency graph to dependent signals, which trigger automatic reevaluations of their values and forward them to further dependent signals of their own. When a new event is fired, two sequential phases occur: In the first phase, the system carries out all computations depending on the previous state. The second phase then updates the state of the computation.

Vega calls its dependency graph *dataflow graph*. In this graph, data tuples travel between nodes, which potentially alter the data along the way. Reactive Vega parses the input data for the visualization and creates a branch in the dataflow graph for every dataset definition. The branches start with an input node, followed by data transformation operators and ending in an output node. Raw data tuples, flagged by their status flag with either *added*, *modified*, or *removed*, start at the input node of this branch, and after application of data transformations leave the branch as data tuples in the output node. Data transformations are able to derive new data tuples. These derived tuples can access the original tuples from which they originated by using prototypal inheritance. Additionally, when data transformations change tuple attributes, the previous values are stored inside the `previous` object. This is used because some transformation operators require access to the previous values. For example, when calculating a running average, the operator might subtract the previous value of a tuple attribute while adding the new one to the aggregate.

For handling interactions, Reactive Vega populates the dataflow graph with event listener nodes, that are listening to primitive JavaScript events such as `mousedown` and `mouseup`, that are declared in the visualization grammar. These event listener nodes are then connected to dependent signals, which are specified by *event selectors* [SWH14]. Event selectors are inspired by CSS3 selectors and enable streams of events to be filtered. Event selectors reference the respective primitive events by their names. If the selector should only apply to events emitted from a specific mark, it can be prepended by a colon. An example of a selector referencing mouse events on a `rect` mark is `rect:mousemove` (see Figure 2.7a). This selector gives a stream of `mousemove` events that are emitted from a `rect` mark. Selectors can also join streams with the `","`-operator (Figure 2.7b). Together with the square brackets (`[]`), which can filter the stream according to properties, and the > operator bounding events for a stream can be defined (see Figure 2.7c). For example, the selector `[mousedown, mouseup] > mousemove` filters a stream of `mousemove` events so that the resulting stream only contains `mousemove` events, which happen between a `mousedown` and a `mouseup` event. Additionally, it is possible to filter streams temporally by providing a minimum and maximum time in curly brackets. For example, the event selector `mousemove3ms,5ms` contains `mousemove` events that are at least 3ms and at most 5ms apart (Figure 2.7d). If such event selectors are used, Vega automatically creates anonymous signals that serve as gatekeepers by ensuring the selector's predicate and only passing the filtered events.

As mentioned before, Vega's scene graph creation is highly inspired by ProtoVis. As Vega's visualization description is static and does not feature functions, it compiles encoding functions for every property when traversing the mark hierarchy in the *bind* stage. The *build* stage performs a data join emitting a mark instance to the scene graph for every data tuple. Like in ProtoVis, the *evaluate* stage is then able to run the encoding functions created in the bind stage. In an additional stage, Vega computes the bounding boxes of generated marks. The resulting scene graph elements are represented as data tuples, which means they themselves can be passed through the dataflow graph where other marks can take them as input. This concept is called *reactive geometry*. It allows, for example, labels to position themselves on the top of the rectangles from a bar chart by taking the top position of the respective bar as input. It also enables higher-level layout algorithms that require prior information like initial layouts, which could otherwise not be implemented.

Vega does not continuously send all of its data through the dataflow graph. Instead, it sends *changesets*. Changesets describe the subset of changes in state information that the system needs to update. This includes new signal values or data tuples and updates to dependencies.

Vega-Lite

With all its mentioned functionality, Vega is a versatile tool that can describe visualizations in great detail. However, as Satyanarayan et al. [SMWH16] put it, Vega, as well as ProtoVis and D3 are useful for *explanatory* data visualization, which is the process of

(a)

rect:mousemove
stream of **mousemove** events that occur on **rect** marks

mousedown, mouseup
single stream merges **mousedown** and **mouseup** streams

(b)

[mousedown, mouseup] > mousemove
stream of mousemove event that occur between mousedown and mouseup

(c)

mousemove{3ms,5ms}
stream of mousemove events that occur at least 3ms and at most 5ms apart from each other

(d)
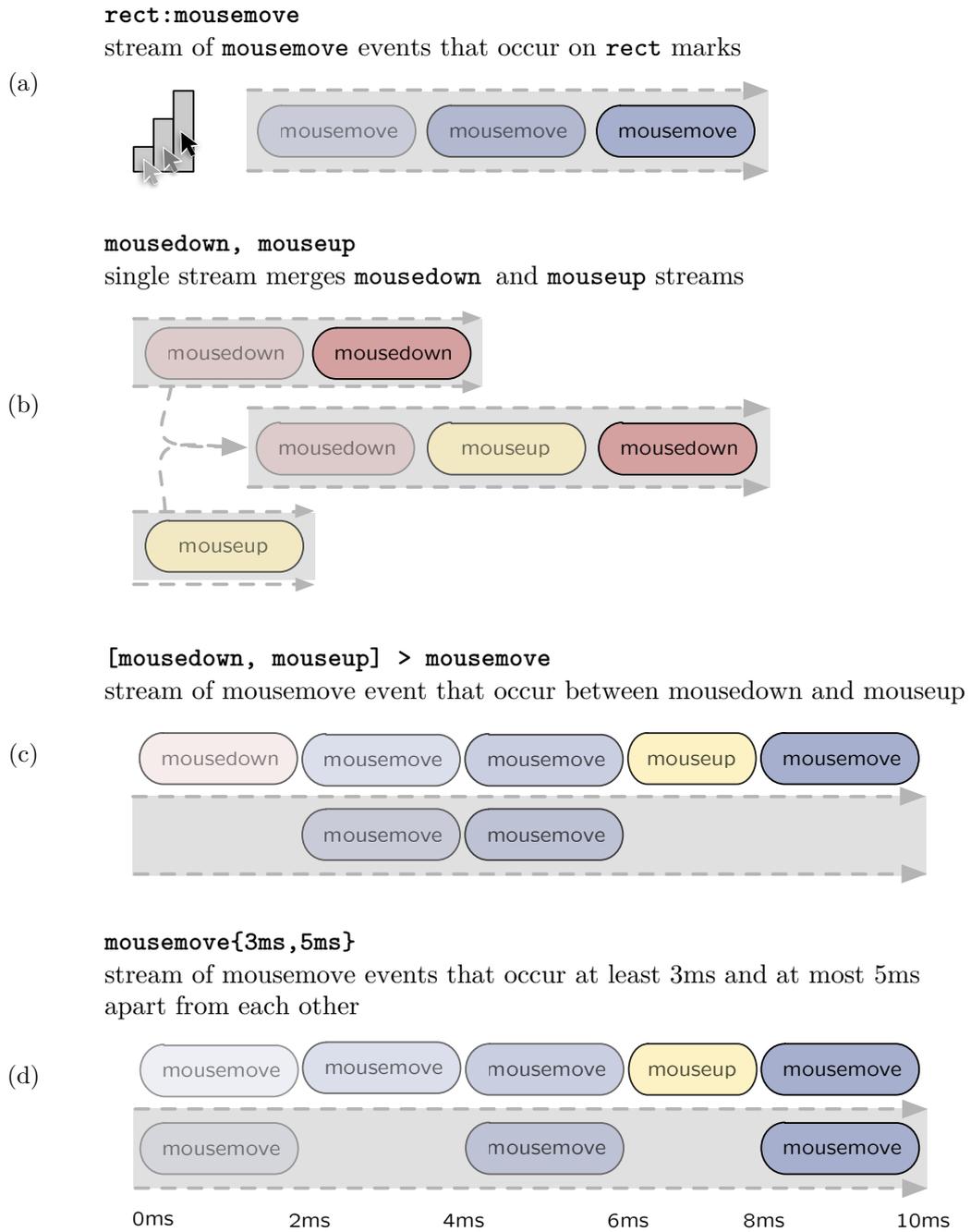
Figure 2.7: Different examples of event-selectors for filtering a stream.
Source: [SWH14]

crafting visualizations to explain data. In contrast, high-level tools like VizQL [STH02] and ggplot2 [Wil12][Wic10] allow a fast exploration of data. These tools provide predefined functionality through concise functions populating various values, that would typically have to be specified, like scales or colors, through default values. This aids a different kind of visualization called *exploratory* visualization. Vega-Lite [SMWH16] is a language built on top of the Vega system to provide similar concise functionality for exploratory visualization.

At the core of the Vega-Lite grammar is the *unit* specification, which describes a single Cartesian plot. Contained in this specification are the *data*, a *mark-type*, and a set of at least one encoding definition for visual *channels*. These channels are equivalent to the properties referred to in previous works. Examples for such channels are color, position, or size. The data in this unit specification needs to be a relational table consisting of records (rows) with named attributes (columns). *Transforms* can operate on these tables, which are capable of filtering values or calculating new columns. As the anonymous property functions in ProtoVis and Vega, the *encodings* describe how data is mapped to the properties of the graphical marks. The first element in the specification of an encoding is the visual channel. Vega-Lite offers various visual channels: *x*- and *y*-position, *color*, *size*, *shape*, and *text*. Also needed in the encoding specification is the *field* value. This value describes which attribute, and therefore column, the specified visual channel encodes. The *data-type* element states to which type of data the value should be encoded. The possible options are: *nominal*, *ordinal*, *quantitative*, and *temporal*. Other than these three mentioned values, encodings can also specify a *scale* that transforms the reference data and a *guide* allowing the definition of an axis or legend. Visual channels that are not specified in the encoding are filled up by sensible default values. Using this approach, various diagrams like bar charts, histograms, or dot and scatter plots can be defined concisely.

An example *unit* specification is displayed in Figure 2.8. Vega-Lite also provides an algebra for composition, which enables the creation of *composite* views featuring multiple such unit specifications. This algebra operates on *views*, which are either unit specifications or composite ones. In that way, views can be combined in arbitrary many levels. One operator of the composition algebra is the *layer* operator. This operator combines multiple views into a single chart that uses the same axes. The concatenation operators *hconcat* and *vconcat* place the specified views either in a single row (horizontal) or column (vertical), respectively. Other possible operators are the *facet* and *repeat* operator.

Interactions are an essential part of Vega-Lite. For this, Vega-Lite introduces a grammar of interaction. This grammar adds *selections* to the unit specifications. Selections form a set of points that the user interacting with the visualization wants to manipulate. When interaction events fire, the selection is filled with selected data points that satisfy the selection *predicate*. By default, these data points are the underlying data tuples of a visual mark instance. Therefore, if the user clicks a visual mark in a visualization, the system adds the respective data tuple to the selection. If a user interacts outside of discrete visual marks, for example, when he tries to draw a selection rectangle, the

```
{
    "data": {
        "url": "data/weather.csv",
        "formatType": "csv" },
        "mark": "line",
        "encoding": {
        "x": {
            "field": "date",
            "type": "temporal",
            "timeUnit": "month" },
        "y": {
            "field": "temp_max",
            "type": "quantitative",
            "aggregate": "mean" },
            "color": {
            "field": "location",
            "type": "nominal" }
    }
}
```
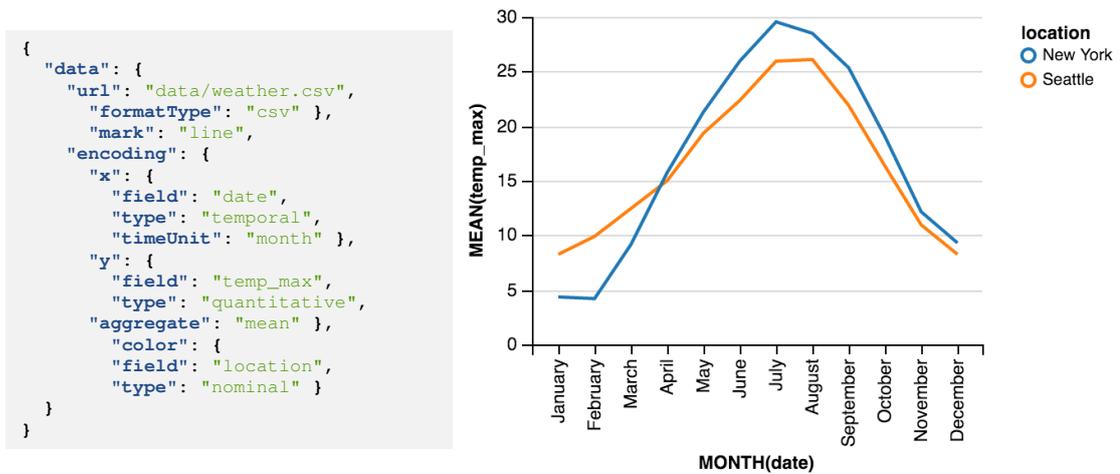
Figure 2.8: Vega-Lite specification of a simple line chart showing the monthly aggregated mean temperature in two cities.
Source: [SMWH16]

mouse positions are transformed by the inverse scaling and checked against the predicates. There are three selection types. Point selections reference only one data point. This single point can also behave like Reactive Vega's signal. Another essential selection type is list selection. The list selection contains a dataset into which data points are inserted, modified, or removed. Lastly, interval selections also reference multiple values, but instead of list selections, they represent continuous selections between a minimum and a maximum value specified in the selection predicate. Selections can not only listen to default events, which are automatically configured depending on the target platform (mouse on desktop, touch on mobile), but also to events specified by Reactive Vega's event selection syntax (see Figure 2.7).

2.3 Functional GUI Frameworks on the Web

2.3.1 Elm

Elm [Cza12][CC13] is a functional language for creating web apps that is based on FRP [EH97] and its related research. It compiles to HTML and JavaScript. Therefore, the language works with the same environment as our solution i.e., front-end web technologies. The Elm architecture consists of three parts: The *model*, which stores the immutable state of the application; The *view* function, which produces and HTML output from data of the model and describes when messages appear; And the *update* function, which reacts to the messages and knows how to update the model accordingly. A simplified event-loop of this system is shown in Figure 2.9.
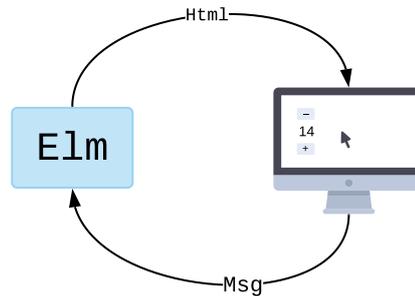
Figure 2.9: The communication flow between Elm and the browser.
Source: [Cza20]

2.3.2 React & Redux

React

Like ELM, React [Inc21] is a functional framework for creating web apps. Therefore it also lives in the web environment. A React application defines functions that hierarchically take the state as input and produce DOM elements as output. The framework then automatically computes which DOM elements to recompute according to how the state changed. React does that by a simple referential comparison. If elements have the same reference, they are considered the same, and no recomputation occurs; otherwise, the system recomputes the DOM elements. Additionally, users may overwrite this behavior with a custom comparison code. React also lets users define stateful components. These components are elements that include state, and hence do not follow the general functional architecture of the framework.

React is written in JavaScript and uses a syntax extension called JSX. This syntax extension allows functions to return HTML specifications of DOM elements directly. Furthermore, JSX enables the mixing of expressions and HTML tags. For example, it is possible to create a JavaScript array and directly map it to HTML elements, producing an element list in the process.

Redux

Redux [Abr21] is commonly used in combination with React and adds advanced state management to the system. At its core, Redux is a single store that handles the complete state. Consequently, all state changes must pass through this one place. They are triggered by so-called *actions* that are dispatched to the store. Redux passes these actions together with the current state to *reducers*. These reducers, in turn, know how to update the state according to the received actions. Finally, code can subscribe to changes in the store and fulfill computations, like updating DOM elements, according to them.

2.4   Visualization Systems for Domain Experts

2.4.1   MeVisLab

MeVisLab [AG21] is a visualization tool primarily developed for medical image processing. It is written in C++, making it a native application [HSP09]. Visualizations are created by constructing a network graph of *modules*. The network graph can also be encapsulated into a *macro module* to reuse visualization techniques or algorithms in new visualizations.

Developers can extend MeVisLab's modules, pipelines or the user interface via both JavaScript and Python. Additionally, an abstract module definition language enables provides an simple method for creating GUIs.

An example of MeVisLab is depicted in Figure 2.10. The network graph in the middle represents the logic of the visualization. The graph consist of various modules the color indicated wether the module is responsible for image processing (blue), visualization (green) or a macro module (orange). Next to the network graph, a 2D (left) and a 3D (right) visualization can be seen. The window below the 3D visualization shows a custom GUI module.

2.4.2   Inviwo

Just like our system, Inviwo [JSS+19] offers users to specify the complete visualization pipeline by using modular components. Inviwo calls these units *processors*. In the Inviwo system, users can create their visualization pipeline by linking these processors inside a graphical network diagram. Thus, the processors form nodes inside a node-based flow diagram like it is used in various visual programming systems.

Processors define the hardware (CPU, GPU) and the computing API (OpenGL, OpenCL, etc.) they use. The Inviwo system then automatically marshals the data communication between these different contexts as needed. In general, Inviwo is a native desktop application that is mostly implemented in C++. However, there is the option to program processors in the scripting language Python.

In contrast to our system, where generic information is passed through the pipeline units utilizing the model, the Inviwo system explicitly transports certain data types between processors. The network diagram reflects these data types by coloring the links between the processors in corresponding colors. This coloring has the additional benefit of enforcing a type-safe linkage.

Similar to popular game engines like Unity, Inviwo offers users the option to configure the processors' properties from a panel where live edits to the data can be made. Additionally, visualization developers can also hide some of these properties to produce a reduced view on the visualization environment. End users can then use this view to modify the visualization according to their needs by configuring the enabled properties only.
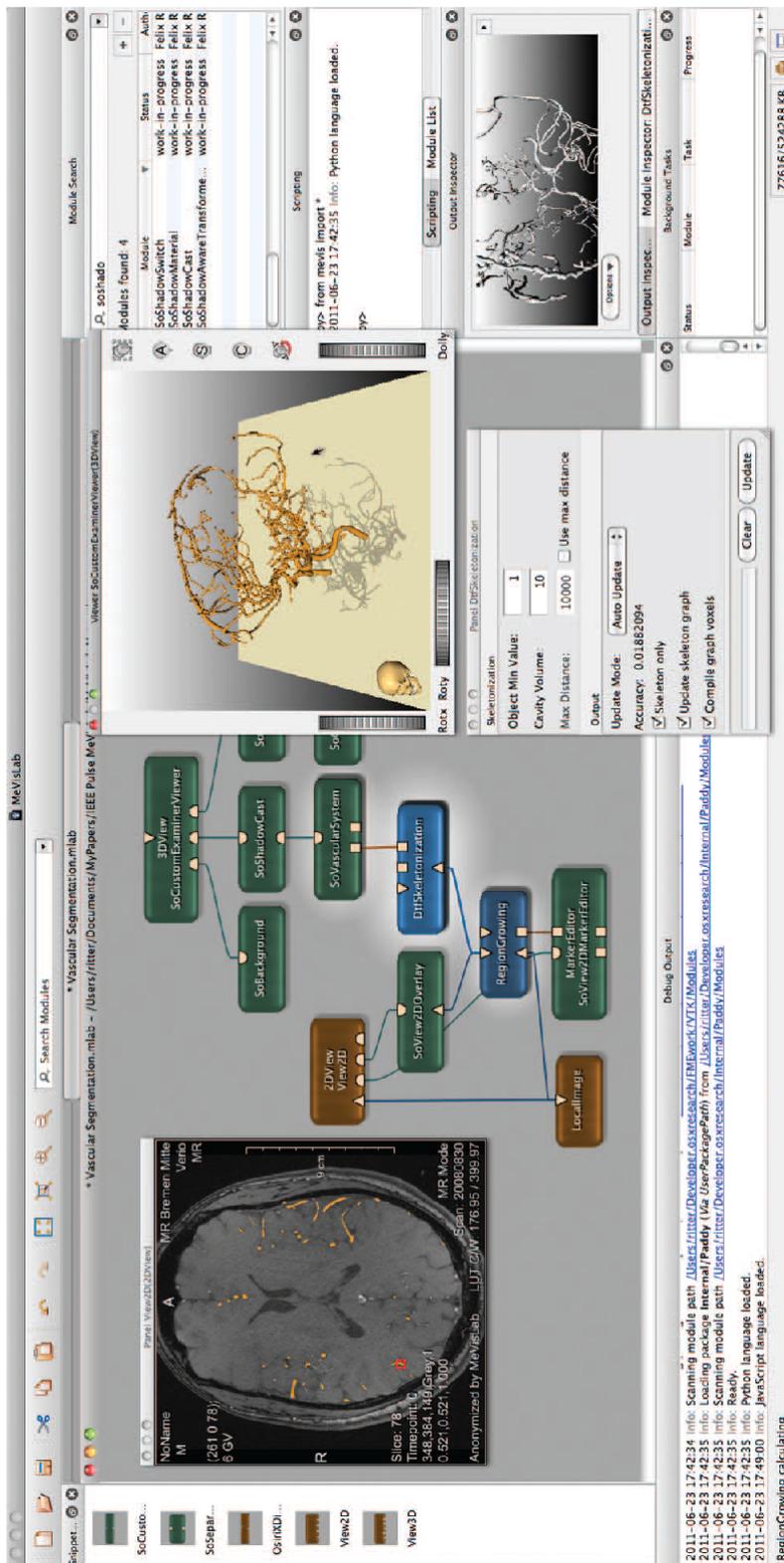
Figure 2.10: A visualization inside the MeVisLab system. With the GUI, the network graph responsible for the visualization can be modified.
**Source:** [RBH+11]
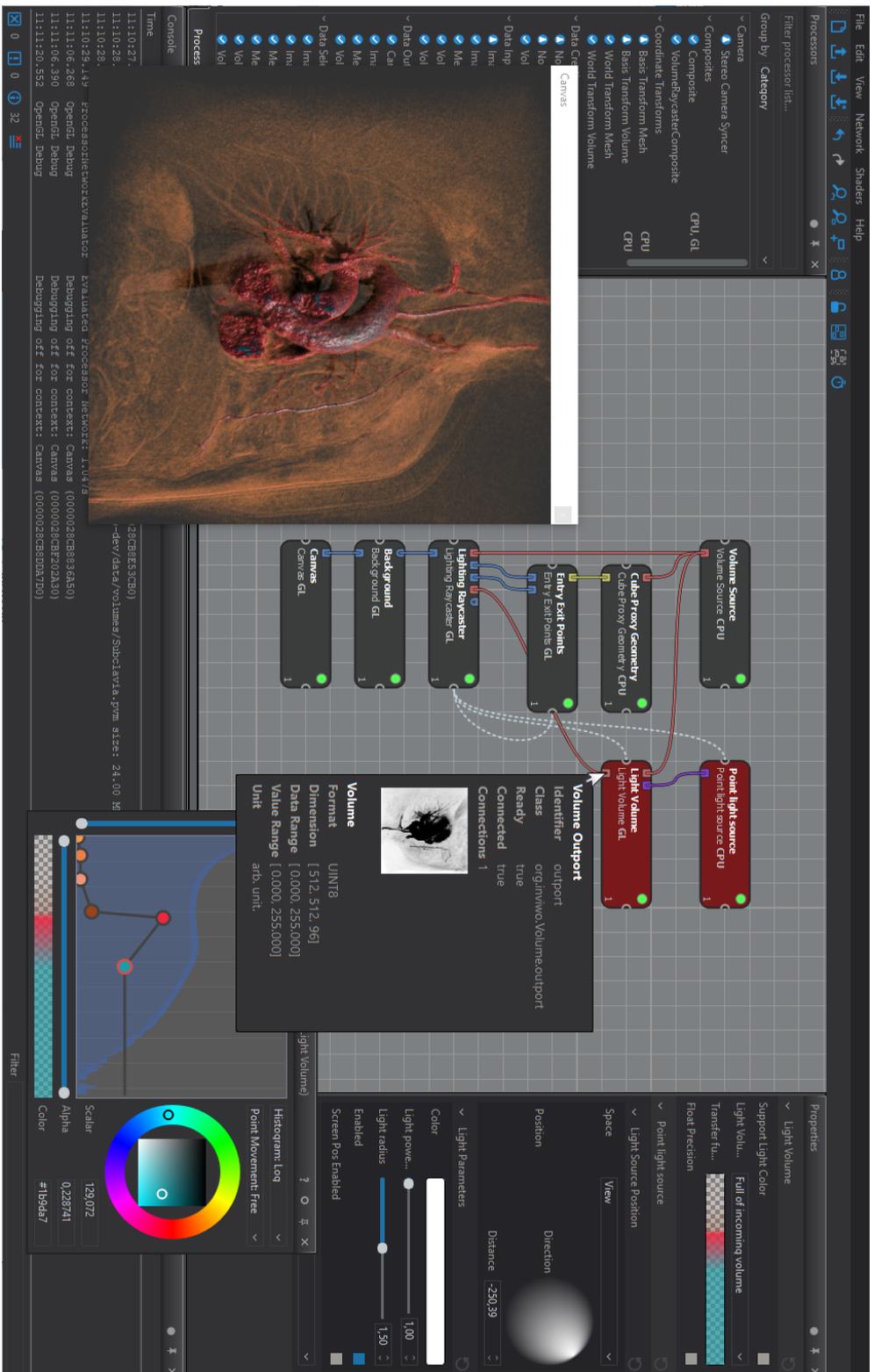
Figure 2.11: Inviwo application showing the configurable node-based flow diagram that creates a volume visualization. **Source:** [JSS+20]

Language Design

This chapter lays the theoretical groundwork for the design of our DSL. At first, we state attributes in the form of design goals that we want the visualization system to have (Section 3.1). These goals then influence our choice of the host language our DSL resides in as an *embedded* DSL. The host language then defines the syntactic language space of our DSL, i.e., the rules defining all possible expressions in the DSL. In Section 3.2, we illustrate how this language space looks like by providing the grammar of the host language. Resulting from the modular design of our system, all expressions in this host language space are also valid expressions of our DSL. This equivalence emerges from the modular components of our system, which can enforce semantic constraints on the DSL, hence defining the semantics of the DSL as a whole. The concept of specifying the DSL loosely by keeping the semantics open for the modules to define is a key aspect of the language-centric view for creating our visualization system. The actual features of the visualization system, provided by this unusual method of defining a language, and the resulting, concrete feature set of the DSL are explained in Chapter 4.

3.1 Design Goals

For starting the design process of our language, we formulate goals that the language and the resulting visualization system should achieve. We formulate goals and argue trade-offs given technological constraints such as compatibility with the web execution environment and usability. The goals of our language are the following:

- **Describe visualizations in a concise, text-based manner:**
  Visualization descriptions should keep the number of code lines small. Definitions should be concise and understandable for non-programmers.

- **Define own interactions inside the language:**
  The visualization description must be powerful enough, to define custom interactions that change the visualization's appearance.

- **Support common visualization types:**
  Our language must be capable of describing common visualization types, like traditional meshes, point clouds, volumes, and flow visualizations. The aim is to provide out of the box support for some popular visualization types, but enable others via an extension mechanism.

- **Enhance the language using an extension mechanism:**
  It must be possible to extend the visualization description language to add custom functionality to the system.

- **Support an interactive workflow:**
  The language must support an interactive workflow, where changes to the source trigger immediate updates and, therefore, enrich the visualization-creation experience for the end user.

- **Performance:**
  The system has to run at similar performance characteristics comparable to hand-crafted visualizations.

- **Efficient recalculation:**
  Changes in visualization state, for example triggered by user interaction, must be handled efficiently by the system. The resulting changes should keep the load of recomputation and allocation of graphics resources small.

- **Saving the visualization state:**
  It must be possible to save the current state of the visualization at any time. The resulting snapshot must reproduce the visualization exactly, enabling a simple way of sharing visualizations.

- **Determinism:**
  The visualization has to update in a deterministic fashion. Therefore, recorded changes must produce the same visualization when replayed, enabling undo/redo and granular debugging capability.

- **Shareability & Platform independence:**
  Visualizations described by the language must be easily shareable to others. The visualization system must run on various operating systems out of the box to fulfill shareability requirements.

3.2 Language Definition

The definition of a language comprises syntax and semantics. The syntax is the set of rules that define how expressions must look to be valid in the given language. Semantics, on the other hand, describe the meaning behind those expressions.

3.2.1 Syntax

One of the design goals is a good sharability of our language. The web ecosystem plays a significant role in how knowledge is shared today. Therefore, we want our language to exist in this environment. Naturally, there already exist languages that are in use for communication. One of these languages is the human-readable JSON format. It is not only the default notation for serializing JavaScript objects, but also a frequently used language for web communication like, for example, in REST APIs. As such, we select the JSON format as the notation of the DSL instead of inventing a new syntax. Therefore, JSON is the host language of our DSL. Automatically, this allows our language to be communicated easily across the web and leverage already existing tools. It also means that the front-end language of the web, JavaScript, can natively operate on it.

Because the DSL uses the JSON format as its basic structure, our language is an *embedded* DSL. Consequently, our language has to abide by the syntactic restrictions that the JSON format enforces. Listing 3.1 shows the grammar of JSON in EBNF form.

In Listing 3.1, we can see that, first of all, JSON has three primitive data types: numbers, booleans, and strings. Additionally, two structure types exist: arrays and objects. Together these five data types formulate every possible instance of what the JSON specification [ECM21] calls *value*. Objects are unordered key-value pairs storing the aforementioned values, indexed by strings as keys. Arrays, on the other hand, are ordered structures that store arbitrary values in a sequence.

These syntactical limitations of the JSON format have some severe consequences for our language. For example, JSON does not feature a way to encode functions and enforces a clear separation between data and code. Another significant limitation is that JSON does not support multi-line strings inside the format. Strings inside the JSON format may contain \n as line-breaks, but they can not span multiple lines inside the file.

3.2.2 Semantics

In any case, our visualization description needs semantics. Other visualization DSLs like Vega [SWH14] and Protovis [BH09] have a grammar that identifies the largest common overlap of the supported visualization-types' functionalities. The grammars then use this common overlap for defining a logical structure for encompassing the similarities as well as the differences of the visualization types. For example, Vega visualizations define data sources and encoding functions as common elements of all visualizations. Vega descriptions specify marks to get a different visualization. Marks are hierarchically structured so that they inherit common properties from their parent marks. Using this

```
json      ::= element
value     ::= object | array | string | number | boolean | null
object    ::= '{' member (',' member)* '}'
member    ::= ws string ws ':' element
array     ::= '[' element (',' element)* ']'
element   ::= ws value ws
string    ::= '"' character* '"'
char      ::= ([#x20-#x10FFFF] - '"' - '\') | ('\' escape)
escape    ::= '"' | '\' | '/' | 'b' | 'f' | 'n' | 'r' | 't'
              | ('u' hex hex hex hex)
hex       ::= [a-fA-F0-9]
number    ::= integer fraction? exponent?
integer   ::= '-'? ([1-9] [0-9]*)
fraction  ::= '.' [0-9]+
exponent  ::= ('e' | 'E') ('+' | '-')? [0-9]*
boolean   ::= "true" | "false"
null      ::= "null"
ws        ::= (#x20 | #x0A | #x0D | #x09)*
```

Listing 3.1: Grammar of JSON in EBNF form based on the JSON specification [ECM21].

structure, other DSLs offer a concise and inherently systematic approach for mapping the space of possible visualization types onto their grammars.

In our design, we take a different approach. While this systematic strategy fits well for languages that define a closed scope of possible visualizations, we think that such a strategy does not suffice with more generic visualization spaces. This inference seems to be particularly valid for the goal of enabling custom visualizations and offering a strong capability of extending the language. Defining an overarching structure where all possible visualizations fit under seems complicated and prone to sacrifices in conciseness for the sake of generality. Therefore, we opted for an unconventional approach: As an alternative to creating this systematic structure, we deliberately do the opposite. We do not define the semantics at all. Instead, we defer this decision to the system's atomic building blocks, *modules* (Section 4.2).

4

System Design

This chapter formulates our visualization systems' architecture that results in the concrete properties and feature set our DSL comprises. The architecture is also key to the following implementation of the system in Chapter 5.

For this chapter, we first give an overview of the general architecture, explaining the various components that make up the visualization system as well as their interplay (Section 4.1). Afterwards, we look at each component individually and give an in-depth explanation of its functionality. In this process, we also sketch alternative approaches and argue for the design choices we take.

4.1 Overview

In general, the whole visualization-creation environment comprises several components, as can be seen in Figure 4.1. The figure is inspired by VisLang's [RBGH14] system overview (see Figure 2.2) to highlight the similarities between the two approaches. On the left, Figure 4.1 shows the DSL. The visualization system in the center takes it as input and produces the final visualization on the right. As illustrated in the figure, the visualization system uses modules that together create the final visualization.

How our language lets users create visualizations is heavily intertwined with how the system operates. Hence, we will describe the system's properties and structure and take those to formulate the functionality and expressiveness of our language.

We start the design process of the system with the central abstraction of every visualization system: the visualization pipeline as described by Card et al. [CMS99]. The visualization pipeline is a highly individual construct and differs between different rendering techniques [Mor12]. However, all pipelines feature two aspects for sure: Every pipeline takes in data initially, and in the end, produces a visualization.
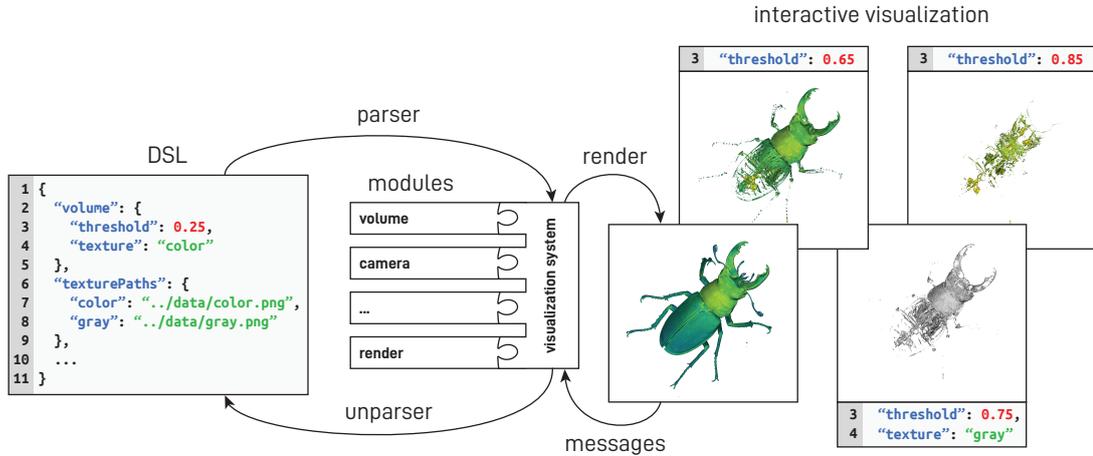
Figure 4.1: The interactive visualization-creation environment consisting of the DSL (left) that describes a visualization which the modular visualization system (middle) reads, producing the interactive visualization (right). Interactions with the visualization trigger changes to the DSL, while changes to the DSL by the user trigger immediate updates to the visualization.

As we want the system to be as generic as possible, users must be able to decide how the pipeline is structured and what its result looks like. Thus, our language enables users to construct the concrete visualization pipeline by combining modular pipeline stages, which we call modules.

A fixed visualization pipeline is good enough if the output is a static image. However, the system needs another functionality: interactions. Hence, the system is not a simple pipeline but a loop that executes the visualization pipeline whenever interactions happen. Figure 4.2 shows the system's event loop. It consists of two distinct parts and the two communication directions between them. A detailed description of these parts and argumentations of why they are crucial to the system, is given in the following sections of this chapter. Here we now give an overview of our final solution to guide the reader through the components and their relationships.
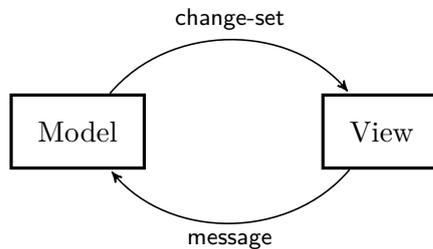

Figure 4.2: The event loop of the visualization system.

We start on the left of Figure 4.2, which shows the model stage. This stage handles the visualization's immutable state, which we call the *model*. During this stage, the system executes the pipeline by taking such a model as input and produces a new model. Then the system computes the differences between the two models constructing the *change-set*, i.e., the collection of all changes. Next, the system passes the change-set to the view stage. This stage controls the *view*, which stores the visualization's mutable state, for instance, references to the GPU, the graphics context, or DOM elements. In this stage, the visualization gets rendered. To complete the circle, the view stage can put user interactions into the message-queue. As a consequence, interactions produce messages that invoke the visualization pipeline. This invocation produces a new model, hence completing the event loop.

4.2 The Module - an Atomic Unit of Functionality

Modules are the building blocks of the configurable visualization system. They also define the semantics of the DSL, which describes the visualizations. By chaining modules together, the whole visualization pipeline emerges, which performs the complex task of transforming input data into an output visualization. Hence, the module concept follows the prominent divide-and-conquer approach to solve difficult tasks by dividing the logic into multiple simple tasks.

From a general point of view, a module is just a function. It takes an input and produces an output. Usually, functions do not take arbitrary inputs. Instead, they constrain their input in some fashion. In statically typed languages, for example, they require the input to be of a specific type. Of course, this might extend to a list of input parameters. Likewise, the functions return a value of a specific type, which other functions can then rely on to have certain properties, as guaranteed by the type.

We want our modules to have the same guarantees. A module wants to state what input it expects and provides a set of constraints its output fulfills. The following example illustrates why this is needed: The pipeline in the example consists of three modules that are arranged in a linear sequence. The first module is the data loading module. It only takes the path to a data file as input. When executed, it reads data from the file with an I/O operation, parses the file in a certain way, and then provides the read data. The second module in the pipeline is a statistics module that expects the first module's data as input and calculates various statistical values and aggregations. Finally, the last module is the actual visualization module producing a visualization as an output. However, as input, the module needs the statistical data from the second module and the raw data points from the first module.

The last module must be able to state that it expects both the result of the first and the second module as inputs. If the third module would indeed be able to state that, a linear pipeline would have no way of passing the output of the data loading module directly to the pipeline's last module.

Reformulating the example into a more formal notation gives us the following: We have a chain of modules m_0, m_1, and m_2. Module m_0 takes input A and produces B. Then module m_1 takes B as input and produces C, while module m_2 takes B and C as input and produces D.

So what we want to do is pass B to m_2 in a sequential pipeline, i.e., a pipeline where modules are executed after another. General purpose programming languages can easily solve this problem. In such languages, one would assign B to a variable and pass the variable to both m_1 and m_2. A simple pseudo-code would look as follows:

Input: Input data A
Output: Output D
1 $B \longleftarrow m_0(A)$;
2 $C \longleftarrow m_1(B)$;
3 $D \longleftarrow m_2(B, C)$;
4 return D

In the sequential pipeline example, this behaviour is impossible, as we only pass the result of one module to the next module's input. Therefore, sequential pipelines cannot express complex dataflows directly. One option for providing such functionality would be to extend the language with constructs for dataflow similarly to dataflow in arrowized functional-reactive-programming [NCP02].

Another option is a more flexible type system that permits data-type generic modules. Since modules should be reuseable without additional dataflow specification, we opt for data-type generic modules. Our method is inspired by *row-polymorphism* known from programming-language theory [Wan91].

Row polymorphism accepts a record as input that contains required fields. Besides these required fields, the record may contain additional fields that the receiver ignores. However, operating on the record retains all fields, the required ones, and the additional ones. As a result, the output automatically retains all the constraints of the input. Usually, it will also provide new guarantees, coming from the computed result.

The rationale behind our composition scheme is that each module adds to the currently available inputs for the next module. However, each component imposes minimal input fields, resulting in a flexible reuse of components in different concrete pipelines.

For the previous example, this would mean the following: The first module m_0 takes A as input, but instead of producing only a B, it produces a record containing the information of A and B. We call this record B'. It is the combined set of A and B so that we can formulate it in set notation as $B' = A \cup B$. The second module, m_1, requires B as input, which is contained within B'. Therefore, B' can be used as input for the second module. Like with the first module, we get a combined output $C' = A \cup B \cup C$. The third module, m_2 requires B and C as inputs, which are both subsets of C' - thus they can be used

as input for the last module which solves the initial problem. Finally, it also creates an output that suffices the combined constraints $D' = A \cup B \cup C \cup D$.
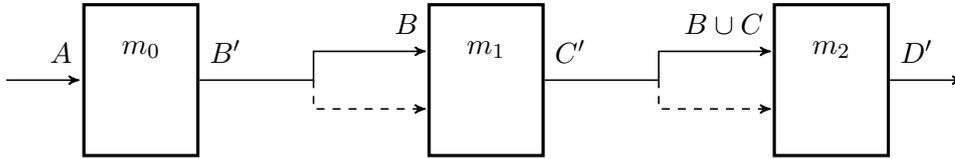

Figure 4.3: The final output contains all previous results because of row polymorphism.

Figure 4.3 shows the row polymorphism solution for the example problem. Additionally to specifying pipelines concisely, this approach allows subsequent modules to only use a subset of the output from the previous module, which again improves reusability. For example, a module could take any input that has a bounding box field for physics calculations. Such a module could take input from various different modules' output as long as they have such a field. More formally, this means that if a module m_n requires an input of form I_n, it can use the output O_{n-1} of a module as long as it satisfies $I_n \subseteq O_{n-1}$. In turn, the module produces an output O'_n that is different from the output its type enforces to produce, O_n. Namely that it is satisfying $O'_n = O_{n-1} \cup O_n$.

4.3 The Pipeline - a Sequence of Computation

Analogous to functions, modules can create complex computations by composition. Composing modules, both hierarchically and sequentially, creates the full computation description of our visualization, which we call the *pipeline*.

The pipeline states the order and the instructions of the execution. The pipeline's main requirement is the ability to specify it inside the DSL, that is, inside the JSON format, while still enabling composition so users can construct intricate visualization programs.

A generic abstraction that comes to mind for modeling the pipeline is a directed graph. While graph execution order can be specified in textual form as demonstrated by, for example, the DOT [aut21] language it has drawbacks: Such graph description languages suffer from the integral problems of translating a spacial object, the graph, to a text-based format. For instance, in cases where various nodes share the same parent, users must split them into multiple text lines. Even worse, the order, especially the starting and endpoint, is not immediately evident, as the line order is often irrelevant in such languages. In general, graphs specified in such a manner can be challenging to visualize as they require users to form a mental model of the graph for understanding its structure.

Another significant drawback of using a pipeline description as a generic directed graph is that the execution order of neighbor nodes (siblings) is not defined. At first, this lack of definitions might seem like a good thing, as the system could decide on an order between such nodes based on performance considerations. However, it might not be apparent

to the user how the system handles the nodes, which increases the user's confusion. Therefore, we refrain from a graph representation of our pipeline.
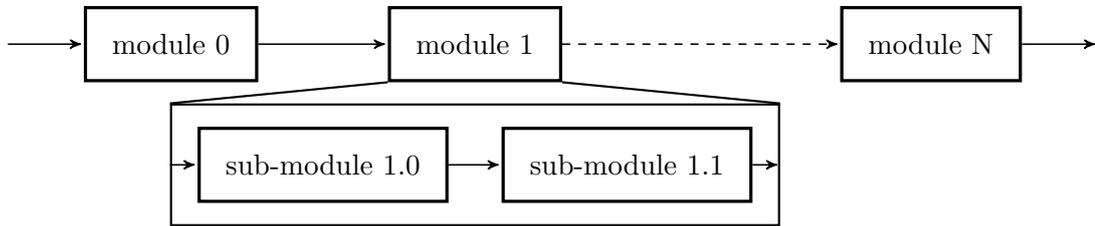

Figure 4.4: A pipeline of linked modules. These modules can recursively contain sub-modules creating a hierarchical pipeline.

Instead, we choose a representation that is easily expressable in the JSON format: An array of modules. The order in the array naturally gives a notion about the linear sequence of execution. For facilitating a more versatile composition, the pipeline array may also recursively contain pipeline arrays themselves (see Figure 4.4). If this capability is still not enough, users can extend the pipeline easily. For extending the pipeline, they have two options. On the one hand, they can create a simple module that executes other hard-coded modules in the desired fashion. On the other hand, they can also define their own custom pipeline-handling scheme inside a module. Then they can reference that module as the only module of the original pipeline, while taking their custom pipeline-description format as input. With this method, users can even implement the pipeline representation with the DOT language, as mentioned above. This notion of offering a sensible default, but allowing to implement custom systems on top is a central design philosophy or our language.

4.4 The Model - an immutable Bag of Data

Semantically, modules, and their use within a concrete pipeline specify a static visualization. The standard way of adding dynamism is to employ a dataflow language or functional-reactive programming. While this is a viable approach, it is hard to align with the design goal of saving the visualization state. For accomplishing this goal, we regard the visualization's state as a principal part of our system's architecture, which we call the *model*. Furthermore, the DSL contains the full information to reconstruct that model and hence enables snapshot capability.

The model stores the complete state of the visualization, so every change to the visualization is reflected in it. This method enables an essential property: Only when the model changes, the respective visualization may change. Because of this property, the model contains all the information needed to create or recreate the full visualization. This information is organized in an immutable tree structure, which means that every model instance is a fixed snapshot of the visualization. Therefore, the pipeline creates new model instances when executed.

The model is not only the output of the pipeline invocation, but it also serves as a communication medium during the computation. Similar to some distributed-computing architectures, the model is the central data hub. Modules take the information they need from it and put back what they produce. However, as models are immutable, putting back information creates a new copy of the complete model. This process might sound inefficient at first, but unchanged branches can simply reference their old instances because of the model's structure as a tree. Thus, the computational load is proportional to the actual changes. One might ask why the model is immutable in the first place. An important reason for this choice is the previously mentioned snapshot capability. The system is constructed from the ground up to deal with new, potentially unrelated models (snapshots) at any invocation. Some beneficial characteristics and use-cases emerge from this decision. For example, different states of the visualization can easily be stored by saving their corresponding models. Later, users can load the saved models and compare the visualizations. Going back and forth in visual exploration thus boils down to restoring a previous state.

The model and the DSL have a close relationship. Other frameworks that feature data stores typically hide their stores from the user. Thus, they require runtime inspections to read values. Our model differs from that by directly linking the input DSL and the model through an intermediary, the parser.

In fact, the model is the parsed representation of the DSL. This linkage allows mapping from the model to the DSL and vice versa, an essential feature for the system (details see Section 5.8).

In general, this relationship means that the DSL itself is an ever-changing representation of the system's live state in human-readable form. Therefore there is not much difference between the model and the DSL. The model is just the system's internal representation, while the DSL shows the system's state to the user. They directly modify their counterpart upon changes and hence are synchronized. Thus, the DSL as textual representation contains all information needed to recreate the visualization.

4.5 The View - an Interface to the Outer World

While the model is the core of the system and directly accessed by the user via the DSL, the view contains all rendering specific data structures such as GPU buffers or graphical elements. While the primary data store, the model, keeps track of all the conceptual states, the view is tracking and storing data that is only relevant for the current execution. Such data includes references to DOM elements or GPU buffers required for running the visualization, but are only valid inside the current context. Those foreign data objects are also prone to internal modification without the system knowing about it. Therefore, the view is mutable by nature. In contrast to the model, the information inside the view may change independently from the system.
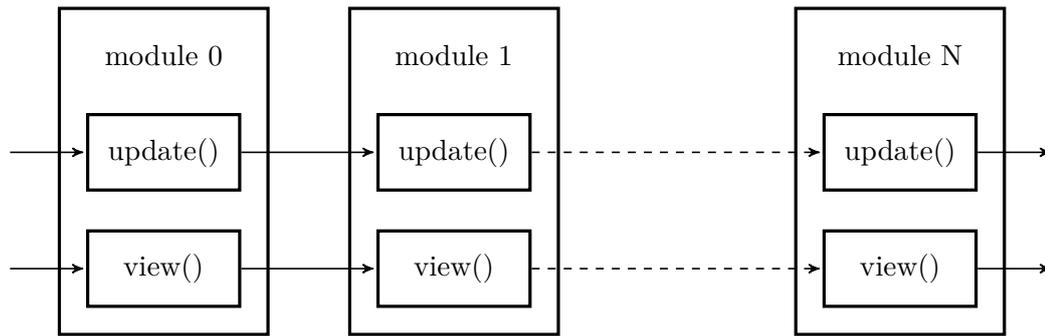
Figure 4.5: The two pipeline tracks: model and view.

The clear separation of those two storages results in a duality of the pipeline's execution logic (see Figure 4.5). We call these two execution paths the model- and the view-*track*. As the name implies, the origins of the model and view separation go back to the Model-View-Controller (MVC) pattern [KP88]. Traditionally the model component in this pattern manages the state of a program. The view manages the GUI and updates it according to the state. Lastly, the control component works as an intermediary by taking user input from the GUI and modifying the model. While this programming pattern was first used for its separation of concern principle [Rea89] and the resulting reusability, our system draws another vital benefit from such a separation: Separating purely functional and imperative, state-based code.

Drawn from functional programming, purity—that is side-effect free code—allows reasoning about the model and its changes easily. It is possible to analyze pure computations and, for example, reorder them in an efficient fashion.

Interfacing with low-level graphics APIs requires mapping of the immutable model state to graphics commands. This problem is not new. Elm (see Section 2.3.1), for example, uses their runtime system to update the GUI state, given a new application state. React (see Section 2.3.2) takes a similar approach and uses so-called reconciliation to update user interface elements automatically. In this work, we take a different approach. While our system provides basic adapters for *Three.js*, the module system enables the creation of custom adapters for other imperative backend APIs.

Traditional systems often use a dependency graph to track changes that are then executed by immutable updates. Instead, in our approach, a module's view function can handle changes of the underlying model through provided *change handlers*. These change handlers know how to modify the view according to changes in the model. For example, when adding entries to the model, the system calls a specific function where the module can react to that change. If other changes like removals or updates happen, other respective functions get called. Further explanation of this process and the implementation is given in Section 5.7.

4.6   The Change-set - a Difference of Models

Invoking the view track requires information on how the model store changed over time. We supply this information to the view track with the *change-set*. The change-set expresses the change between models by describing the difference between them.

A change-set can be one of the four primitive changes (*None*, *Add*, *Update*, or *Remove*) or a complex object with properties that can be complex changes or primitives. This representation expresses the difference between two models in a hierarchical fashion, making early exits possible. For example, if a large subtree of the model stays constant, that subtree's root node is a primitive None change instead of a change-set object.

Figure 4.6 shows an example of such a change-set, between the old model M_{old} and the new model M_{new}. The resulting change-set of this example contains all different cases that can arise: Branch c of M_{old} does not exist in M_{new}, wherefore its node points to a *Delete* value in the change-set. The reverse happens for branch ac. It exists in M_{new} but not in M_{old}, so therefore it gets an *Add* value. If a value changes but the structure stays the same, like in the ab branch, the change-set will contain an *Update* value. If the values stay the same, two cases can appear. Either the change is a direct *None* as in the aa branch. Alternatively, as with branch b, the whole subtree stays the same. In the latter case, the *None* value is the subtree's direct child instead of containing the complete node structure as in the models.
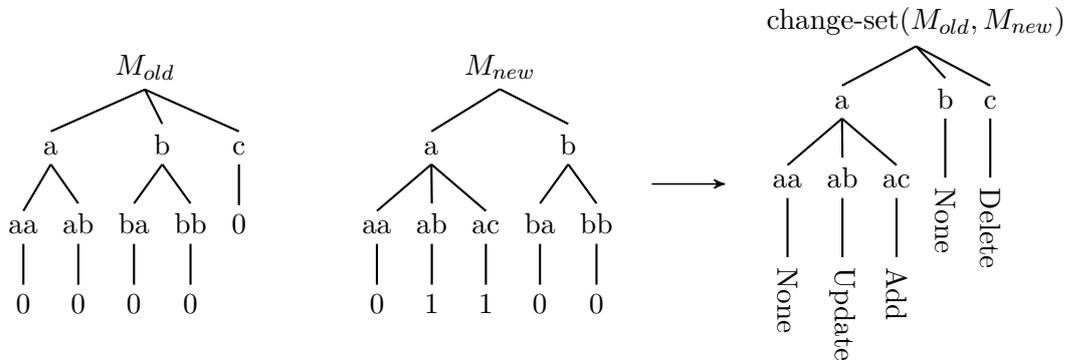

Figure 4.6: A change-set of two example models, M_{old} and M_{new}.

In brief, the explained change-set covers the information flow from the model track towards the view track. Therefore, we covered the upper arrow from the event-loop diagram (Figure 4.2). Additionally, as the change-set provides the exact information about how a new model changed compared to the old one, it is essential for models' reuse and snapshot capabilities.

4.7   The Message-queue - an Order of Communication

The message-queue is the communication medium for getting information from the view back to the model. Hence it completes the communication circle from Figure 4.2. The queue is a simple sequential storage where asynchronous processes can enqueue messages at any time. Modules can emit messages either during the view-track invocation directly or through callbacks. The view function can pass these callbacks to other objects outside of the pipeline's control, such as DOM elements. Therefore, the message-queue works as a bridge, translating between external events and internal modules.

For each frame, the system reads out the queue until it is empty and processes the messages in FIFO order, invoking the model track once for each of them. This process works similar to the typical message pumps encountered in GUI applications.

Messages consist of two parts, the type, and the payload (see Figure 4.7). Modules can discriminate the messages by the type and, in that way, decide if an incoming message is relevant for them or not. The payload contains the actual information transported via the message. Of course, this can also function as a secondary mechanism to decide whether the message is relevant.

Every track invocation produces a new model that serves as input for the next one. Eventually, after all messages are processed, the calculation of the change-set (Section 4.6) happens, followed by a single invocation of the view track, completing the event loop.
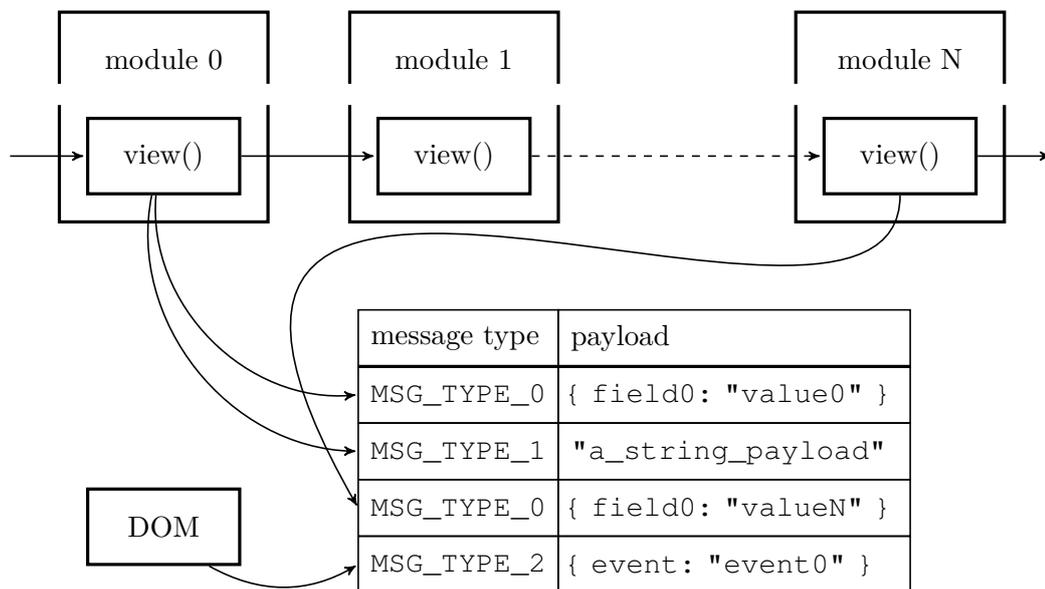

Figure 4.7: The message queue, in which modules as well as other processes like DOM elements enqueue messages.

Implementation

This chapter outlines the implementation of our system as described in Chapter 3. As our system lives in the web-ecosystem, our choice of the implementation language is constrained by a vital requirement: the ease of interfacing with this ecosystem. Naturally, JavaScript, as the language of the web, fits this requirement. However, JavaScript is *loosely typed* and a *dynamic* language. We want to enforce some constraints on user-written code and provide some guarantees through a type system to avoid costly runtime checks. Hence, we choose TypeScript as our language, which is a superset of JavaScript and beside a *static* type system also provides various features from the latest ECMAScript definitions.

The result of the system's computation, the view, is a generic JavaScript object that contains an arbitrary output. Hence, the system can also produce other output than visualizations. Nevertheless, we created the system with the visualization context in mind. Therefore, the view store references a canvas element that contains the resulting visualization. We also do not require a specific technology to create the canvas element's content. As a result, the system is rendering-backend agnostic. However, the example modules use the *Three.js* JavaScript library, a WebGL wrapper, for its rendering. With the example modules, we show how our system can interface with such a rendering library and create visualizations in the process.

Our system is largely influenced by the *module* concept described in Section 4.2. Similar to classes, modules handle their own, encapsulated tasks. The modules only interface with each other in a predefined manner. Therefore, the library managing the system and language is mainly a hierarchical construct of these modules. The system's whole functionality executes from a central place that is itself a module, the *core* module. The whole system, except a thin interfacing layer that exports the external calls and manages the event loop, is part of this module hierarchy (see Figure 5.1).
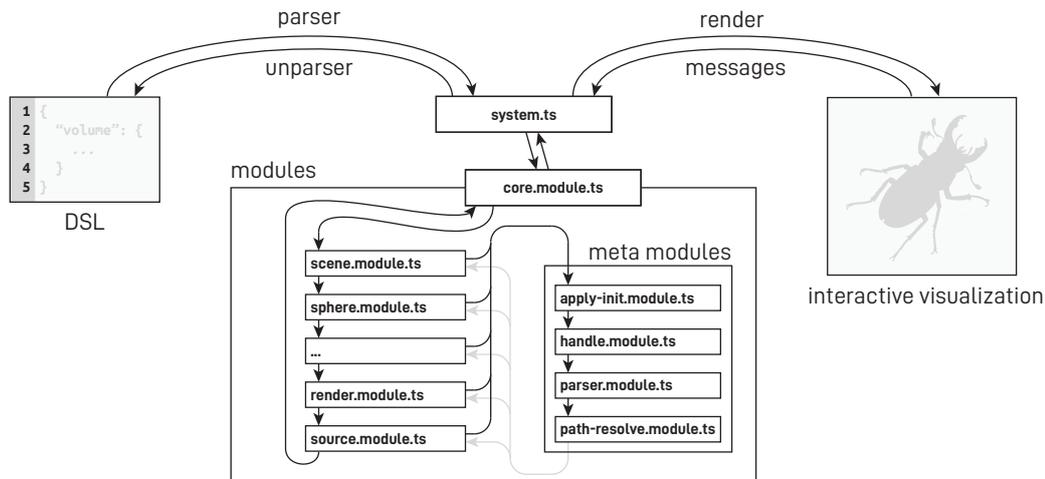
Figure 5.1: Implementation-centric overview of our system from Figure 4.1. The system handles the event loop and communicates with the core module, which delegates tasks like rendering and message processing down the pipeline where modules and meta modules operate in a user-defined sequence.

The event-loop handling consists of the four parts from Figure 4.2. While invoking the model and view tracks are simple calls to the core modules *update* and *view* functions (as seen in Figure 4.5), the computation of the change-set and management of the message-queue reside at the top level of the system, outside of the module hierarchy.

5.1 Computing the Change-set

As described theoretically in Section 4.6, the change-set represents the difference between two models. With this information, the system knows how its state changed and how it must change its visualization to reflect the new state. Thus, it is of vital importance for the execution of the view track. In practice, the system computes the change-set for two cases: Either once per frame by taking the newly computed model (M_{new}) and the model from the previous frame as input (M_{old}), or when a model snapshot is loaded.

Before describing how we compute the change-set, we first define the structure of the computation's output. Listing 5.1 shows the type definition of the change-set in TypeScript notation. As can be seen, the four primitive cases of the change-set are described by the `Change` enum. A change-set of generic type `T` is either an instance of such a `Change` enum or a complex object. This complex object has the same structure as the given `T`, but for every field (key) in `T`, it has itself a change-set. Whenever the key points to a primitive, the change-set must resolve to a primitive enum instance. The `T` in this case is the union type of M_{old} and M_{new}, hence `T` $= M_{old} \cup M_{new}$.

```
enum Change {
  None,
  Add,
  Update,
  Remove,
}

type ChangeSet<T> =
| { [K in keyof T]: T extends object ? ChangeSet<T[K]> : Change }
| Change
```

Listing 5.1: Type definition of the change-set.

With the definition of the output, we can now formulate Algorithm 5.1, the diffing algorithm that manages the change-set computation of its inputs.

At first, the algorithm compares both models (Line 2). If the two models are both primitives (number, string, or boolean), this comparison compares their values. Instead, if they are both objects, their references are compared, which means that the test only passes if M_{old} and M_{new} are the same objects. In both cases, `Change.None` is returned if the comparison evaluates to true. Afterwards, the algorithm only continues if both models are objects (Line 5). Otherwise, the change-set is a flat `Change.Update`. However, if the check passes, we probably have to construct a complex change-set object.

Constructing this complex change-set object requires two loops each over the properties of one of the models. If a property exists in both models, the `diff` function computes the change-set recursively (Line 10), resulting in a potentially nested change-set. Therefore, the algorithm, in general, walks the object tree in a depth-first fashion. This choice models the expected form of changes in the object tree. Most changes to the model will only affect a local branch of the object tree.

If the property only exists in M_{old}, the change is a flat `Change.Remove`. In the symmetric case, if the property only exists in M_{new}, the change is a flat `Change.Add`. During the two loops (Line 8, Line 17), the algorithm keeps track whether one of the resulting entries in the change-set is different than `Change.None`. If all entries are `Change.None`, the algorithm does not return a complex change-set but a flat `Change.None` instead. Therefore the `Change.None` "bubbles up". Even if the two models' objects have different references (they do not pass a simple equality comparison), if all of their containing properties have equal values, they are themselves considered equal.

5.2 Managing Messages

As described in Section 4.7, we collect messages inside the message-queue and thereby handle the communication from the view to the model track. For this approach, we have

---

Algorithm 5.1: Diffing algorithm for computing the change-set between two input models.

Input: An old model M_{old}, and a new model M_{new}
Output: Change-set C

1 **Function** `diff`(M_{old}, M_{new}) **is**
2   **if** $M_{old} = M_{new}$ **then**
3   **return** Change.None;
4   **end**
5   **if** `isObject`(M_{old}) **and** `isObject`(M_{new}) **then**
6       $C \longleftarrow \emptyset$;
7       $changed \longleftarrow$ false;
8   **foreach** $p \in M_{old}$ **do**
9   **if** $p \in M_{new}$ **then**
10             $C[p] \longleftarrow$ `diff`($M_{old}[p]$, $M_{new}[p]$);
11   **if** $C[p] \neq$ Change.None **then** $changed \longleftarrow$ true;
12   **else**
13             $C[p] \longleftarrow$ Change.Remove;
14             $changed \longleftarrow$ true;
15   **end**
16   **end**
17   **foreach** $p \in M_{new}$ **do**
18   **if** $p \notin M_{old}$ **then**
19             $C[p] \longleftarrow$ Change.Add;
20             $changed \longleftarrow$ true;
21   **end**
22   **end**
23   **if** $changed$ **then return** C;
24   **else return** Change.None;
25   **end**
26   **return** Change.Update
27 **end**

---

to provide a mechanism to access the system's message queue inside the view track and thus from the modules' view functions.

To achieve this, modules have access to the `emit` function through the view store (see Listing 5.2). This `emit` function expects another function as input, that we call `handler`. The `handler` function can produce a message given a view `V` and a potential event `E`. The output of the `emit` function is another function, this time taking an event `E` and producing some output. Whenever the function is called, the system executes the provided `handler` function with the latest view and the event of the outer function. Subsequently,

the system enqueues the resulting message of the `handler` function automatically into the message-queue.

```
type Emitter<V> = V & {
  emit<E>(handler: (view: V, event: E) => Message<any, any>):
  ↪  (event: E) => any;
};
```

Listing 5.2: Type definition of the emitter with the emit function.

In practice, this mechanism looks like follows: We want the system to send a message through the pipeline, for example, a `CLICK` message when the user clicks anywhere on the screen. A basic implementation could look like in Listing 5.3. At first the `emit` function is called with a function that creates a click message with the `x` and `y` position of the `MouseEvent`. The `emit` invocation produces another function, the `clickListener`. This `clickListener` has the right signature, thanks to the type definition in Listing 5.2, to be passed to the generic DOM `addEventListener` method. Now when the `click` DOM event fires, the `clickListener` automatically enqueues a `CLICK` message with the `MouseEvents` x and y position into the message-queue.

```
let clickListener = v.emit((_, e: MouseEvent) => Click(e.x, e.y));
window.addEventListener('click', clickListener);
```

Listing 5.3: Emitting a message when a click happens.

Implementation-wise, the message-queue is an array where messages arrive in FIFO order. When the system processes the messages, it reads them sequentially and executes the model track once for each of them. After this process, the system clears the message-queue, and new messages can arrive.

5.3   The Core Module

The core module is the interface between the system's event loop and the module system. In general, the system handles the event loop (see Figure 4.2) by executing the four parts:

- model track invocation

- change-set computation

- view track invocation

- message management

Of these parts, the system covers the change-set computation (Section 5.1) and the message management (Section 5.2) implicitly. For the other two parts, the model- and the view track invocation, the system uses the core module. This module handles the heart of the module composition system, the pipeline as described in Section 4.3. Like all modules, it offers an update and a view function to external callers. However, instead of ordinary modules that must be defined inside the DSL, the core module implicitly always exists and takes the information from the `pipeline` path into consideration. The system can access the update and view functions and consequently start the complete pipeline execution with a single call of the core module.

The the system can, by the use of the core module, orchestrate the full event loop. The core module then parses the current pipeline information from the model and recursively executes the matching modules. As the pipeline definition is part of the model itself, an exciting property arises: Modules can modify the execution order by changing the pipeline description. Hence, modules can remove themselves from the pipeline or enqueue other modules.

Processing the update and the view function of the core module works identically. The module evaluates the pipeline recursively and passes either the model store or the view store through the module chain. As described in Section 4.3, a pipeline node can not only be a module but also an array containing other pipeline nodes. Therefore the pipeline is hierarchical and has an implicit execution order by the nodes' position inside the array(s).

5.4 Meta Modules

In Section 5.1 and 5.2, we described the system's aspects that live outside the modules. The rest of the system's functionality works solely through the module system, with essential parts implemented inside the core module (Section 5.3). For providing the full system functionality, the core module also relies on various aspects that the individual modules must fulfill. A novel mechanism provides modules with these aspects, i.e., meta modules.

Meta modules are a special class of modules. They can add functionality to other modules. As such, meta modules work in a similar fashion as aspect-oriented programming [KLM$^+$97], where so-called *aspects* can add functionality at specific points, before or after a function. With meta modules, this addition of functionality happens via the `moduleTransform` function. This function takes a module as input and outputs a new module. In general, meta modules provide `moduleTransform` functions that wrap (or "box") the incoming module inside a pseudo module with different functionality.

To illustrate this behavior, we formulate a hypothetical *logging* meta-module. The logging meta-module would supply a `moduleTransform` that takes the incoming module and outputs a new module where, before and after the `update` and the `view` function, a logging line is called (for example: `console.log("start module: " + module.name);`). Now the system applies the `moduleTransform` function to every module in the pipeline.

Therefore we achieved a simple logging mechanism where all modules log their invocation to the console.

In the DSL, users may specify meta modules in the same manner—an array that potentially contains recursive sub-arrays—as in the normal pipeline definition (see Section 4.3). The core module expects the resulting meta-module pipeline under the `metaPipeline` identifier. On incoming `INIT` messages, the core module invokes the meta pipeline before parsing the regular pipeline. While parsing the regular pipeline, the system applies the `moduleTransform` function from the model to all modules. In theory, the meta modules can also add other arbitrary information into the model. The meta modules we implemented however work solely by chaining their internal `moduleTransform` to the `moduleTransform` function of the model. By default, this is an identity function initially. As this function chaining is entirely linear, special consideration must be placed on the order if an interdependence between certain meta modules exists. Meta modules can chain themselves in two different positions of the `moduleTransform` function call chain: as first or as last modules.

The meta-module system enables multiple essential aspects of our system, on which standard modules may depend. Consequently, the correct order of meta modules is crucial. Therefore this functionality is not exposed to novices, and by default, the system includes a sensible choice. Nonetheless, everything in our system is configurable so that experienced users may modify it.

5.5 Guaranteeing a Valid Model

In Section 4.2, we established that models could expect a particular input I_n on which they operate for producing their output O_n. Even though the passed input I'_n can be a superset of the required input ($I'_n \supseteq I_n$), the required I_n must still exist inside the input. We want to guarantee this property so that modules can rely on the system, ensuring that the input satisfies the required constraints. We achieve this with the `applyInit` meta-module.

The `applyInit` meta-module is a rather basic, but important, module. It guarantees that on an invocation of the model track, all requested model properties already have values. The guarantee stems from the enforcement of the type system. Namely, every module must include an `initModel` property that has to abide by the same type. As such, the `applyInit` meta-module just wraps the `update` function. This wrapped function passes a combined object containing the properties of the `initModel` as fallback values overwritten by the properties of the passed model down the actual `update` function. In this way, an important assumption for a module implementor holds: every property described in the model type exists at runtime.

5.6 Path Resolution

For communication between modules we use a resolution scheme to map variable names to values. This is handled by the `resolve` meta-module. It resolves the path specifications of the data stores that a module requests. This functionality is integral to more complex visualizations as it allows scoping of model and view. Usually, the complete visualization operates in the root scope, meaning that all modules receive and provide their data from the top level of the model, or view respectively. Communicating over this root scope is essential, as it offers a default space for transferring broadly shared data.

For example, *Three.js* has a scene object where all rendered elements must register. At the beginning of the pipeline, a module provides such a scene object and puts it in the root scope to provide easy access in all subsequent modules. However, if a user wants to render two scenes simultaneously, a conflict occurs if both scenes are placed and expected under the same location.

However, also in simpler setups scoping is critical. This can be seen, for example, if two simple model definitions as in Listing 5.4 exist. If these two models placed their data in the root scope, one `position` property would overwrite the other one. The modules defining these models can have a default scope that is different from the root scope to solve this problem. For example, the module defining the box model would save its model inside the `box` scope. Likewise, the module defining the sphere model would save its model inside the `sphere` scope by default. Now that the two models are independent, they form a structure inside the root model as seen in Listing 5.4b.

```
type BoxModel = Model<{
  dimensions: Vector3;
  position: Vector3;
}>;


type SphereModel = Model<{
  radius: number;
  position: Vector3;
}>;
```

```
box: {
  dimensions: Vector3;
  position: Vector3;
},
sphere: {
  radius: number;
  position: Vector3;
},
// ...
```

(a) Model type definition   (b) Model types inside the root model

Listing 5.4: A simple box and sphere model definition.

Equally important is, how modules access each other. As mentioned in Chapter 3, they communicate via the two stores: the model and the view. So a module has to state somehow that it requires specific information from those stores. The system accomplishes this via the `modelPaths` and `viewPaths` properties of the module. The type system guarantees these two properties to have at least one element each: the module's own model and view. Furthermore, if *requirements* are defined in the type definition of the module,

an equal number of additional fields in the `modelPaths` or `viewPaths` array have to exist. Each field of the two path properties is an array of strings defining a path into the model, starting from the root node. Paths of the two sub-models in Listing 5.4b would be `["sphere"]` and `["box"]`. Deeper paths are just arrays with more entries. For example, the radius of the sphere sub-model is located under `["sphere", "radius"]`.

The path-resolve meta-module resolves the requirements of the modules in the respective `update` and `view` function. From the module creator's perspective, the meta module acts like the injector of the popular dependency injection pattern [Fow04]. Conversely, an implemented module acts as the client by requesting certain other stores to be accessed. The module does this by stating the types of the stores it expects in a specific order.

It is the user's responsibility to link to the correct paths whenever the defaults need to be overwritten. This responsibility is a conscious design choice. An alternative would be that a module supplies a list of names (or actual references) referencing the required modules in the same fashion as the DSL references the pipeline. However, such a system could not contain multiple instances of the same module, which is a useful property. This can be illustrated by a simple *copy-data* module that takes the data from a specific path and writes it to another model path. It would be quite surprising for the user, if that module could only do that copy operation once with a fixed path inside the pipeline. So we certainly need a user for wiring (connecting) the modules.

Another approach to that might be to bind inputs and outputs of modules to variables. One could specify that the copy-data module reads from x and writes to y. The next module would then read from y and write to z. Therefore, the user would specify named storage locations that the system, in general, would translate back to paths inside the stores. For avoiding conflicts with actual data, such variables could use specific prefixes like `__var_x:` inside the model. In practice, the result of this approach would again be a path-based solution. That is the reason why we choose to rely on paths directly for wiring up the modules. The described variable system is then just a subset of our more general approach. By taking this more general choice, a meta module can easily implement the mentioned variable system as an abstraction layer of the path system. Additionally, the path system offers an explicit view of what the system does, which helps users to understand the control flow faster.

5.7 Module Handling

With the system we covered until now, users can already create their own modules. However, we want to ease the module creation as much as possible and also give users the guarantees we discussed through the type system. This feature requires an optional layer that translates between how the system expects to call the modules and how users write them. We construct this layer through the *handle* meta-module.

The handle meta-module builds the link between the system's internal framework and the specification of a module by a user. The system expects simple `update` and `view` functions with certain parameters as seen in Listing 5.5. For the `update` function, the parameter list consists of the message, which includes the message type and the message payload, as the first parameter and a list of required models as following parameters (at least one element for the own model). The `view` function takes the own model as the first parameter, then a change-set, and lastly a list of required views as the following parameters (also at least one for the own view). Noticeable is the choice of only incorporating one, namely the own model, as input for the view function.

```
type UpdateFunction<T extends ModelModuleType> = (
  msg: Message<any, any>,
  ...m: ModelRequirements<T>
) => ModelOf<T>;

type ViewFunction<T extends ViewModuleType> = (
  m: ModelOf<T>,
  c: ChangeSet<ModelOf<T>>,
  ...v: ViewRequirements<T>
) => ViewOf<T>;
```

Listing 5.5: Update and view function type definition.

This choice has two reasons: First of all, only one parameter can syntactically be the rest parameter (indicated by the `"..."` before the parameter name), which has to be the last parameter of the function signature. The rest parameter translates to an arbitrary number of actual parameters passed to the function. In the `view` function, the required views already take the spot of the rest parameter, so the required models would have to be specified in a different, more convoluted fashion. The second reason for taking this approach is to simplify the system's execution model and the user's mental model. The user expects only one model, and the system only passes one model down the view function. If a module requires data from a different model to access inside the view function, a simple approach exists: The module registers the other model as a required model so that the module has access to it inside the `update` function. Then this function puts a reference to the foreign model inside the own model. Therefore the view function, having only access to the own model, can still access the foreign model.

5.7.1 Update

While the definitions in Listing 5.5 provide a simple calling interface for the system, from the perspective of a module implementor, they have several drawbacks. The `update` function accepts all messages type-wise, and also the system passes all messages to every module. Therefore the first task most implemented update functions would do is deciding what messages are relevant for the module and only handling those. This task

is what the handling meta-module provides for describing the update step. Instead of the `update` function, it expects modules to have an update object, the update handler (see Listing 5.6).

```
type HandledUpdateFunction<T extends ModelModuleType, P> =
  (payload: P, ...m: ModelRequirements<T>) => ModelOf<T>;

type UpdateHandler<T extends ModelModuleType> = {
  [K in MessageType<T>]: HandledUpdateFunction<T, T['message'][K]>;
};
```

Listing 5.6: Type definition of the update handler.

```
1  update: {
2    INIT: () => {
3        // do some computation
4        return newModel;
5    },
6    RESIZE: canvasSize => ({ canvasSize }),
7    MOUSE_MOVE: mouseMoveListener,
8    CLICK: (point, model) => ({ ...model, point })
9  },
```

Listing 5.7: Example of an implemented update handler.

This update handler has to have properties for all message types it expects (i.e., the module wants to handle). The values of these properties are "HandledUpdateFunctions". They look similar to the normal update functions. However, instead of taking the whole message object as a parameter, they only specify the payload respective to the message type defined as the property name. Defining updates in such a way is very compact and clear, as seen in Listing 5.7. Thanks to JavaScript's arrow function expressions it is possible to write short functions in one line (see Line 6, 8). The syntax also supports more complex functions that span multiple lines (as in Line 2), but long functions clutter the update definition. Instead, we recommend referencing a function defined elsewhere as in Line 7, which gives a concise overview of what happens on the incoming messages.

Given such an update handler, the handle module checks whether an incoming message type exists in the update handler. If yes, the handle module executes the update handler and unpacks the payload of the message. If not, the handle module just skips the message and returns to the caller early.

5.7.2 View

Similarly, the handler module augments the view function with a view handler (see Listing 5.8). Here the module can specify functions for the four change states from Section 4.6 (Add, Update, Remove, None) as well as an additional state, *Always*. The respective functions are called if the model changes in the specified way (see Algorithm 5.1). The signatures of these view functions differ from the normal view function in Listing 5.5, in that they skip the second parameter, the change-set, as it is not needed. Additionally, the optional function for the *Always* state is, as the name implies, called in any case after the normal case (one of the four change states). This *Always* case is useful to reduce code duplication if a module wants to execute a function in all branches. For example, the render module uses this functionality to execute the render call in all cases.

```
1   type ChangedViewFunction<T extends ViewModuleType> = (
2     m: ModelOf<T>,
3     ...a: ViewRequirements<T>
4   ) => ViewOf<T> | void;
5
6   type ChangeHandler<M, T extends ViewModuleType> =
7     (T extends ModelModuleType ?
8       (M extends object ?
9         { [K in keyof M]: ChangeHandler<M[K], T>; }
10        : never) // kill else branches, move to ChangedViewFunction
11      : never)
12    | ChangedViewFunction<T>;
13
14  type ViewHandler<T extends ViewModuleType> = {
15    Add: ChangedViewFunction<T>;
16    Remove: ChangedViewFunction<T>;
17    Update: ChangeHandler<ModelOf<T>, T>;
18    None?: ChangedViewFunction<T>;
19    Always?: ChangedViewFunction<T>;
20  };
```

Listing 5.8: Type definition of view handler.

In the same vein as the previously mentioned update handler, the view handler can supply functions in-line (see Listing 5.9, Line 11) or reference them (see Line 2).

Handling of the update stage happens differently to the other changes. In addition to defining a view function directly, it is also possible to supply an object, the change handler. This change handler is guaranteed to define functions for the complete structure of the changed model by the type system (see Listing 5.8, Line 6). The change handler requires a view function or an object containing all the model properties at the same depth. For every property of the model, the same requirement applies recursively. In the end, there always exists an executable view function. For example, in Listing 5.9 a view handler for a

```
1  view: {
2    Add: initView,
3    Update: {
4      radius: (m, v) => {
5        v.mesh.scale.set(m.radius, m.radius, m.radius);
6      },
7      position: (m, v) => {
8        v.mesh.position.set(m.position.x, m.position.y,
         ↪  m.position.z);
9      }
10   },
11   Remove: (_, v) => {
12     v.mesh.parent?.remove(v.mesh);
13     v.geometry.dispose();
14     v.material.dispose();
15   },
16 },
```

Listing 5.9: Example of an implemented view handler.

simple sphere module is defined. The *Update* change state of this view handler references a change handler instead of a view function. Therefore all properties of the respective model of the sphere module (Listing 5.4) either have to have view functions directly or sub-objects. The former is the case in the example of Listing 5.9. Alternatively, the `position` property could reference an object with different view functions for the three resulting properties x, y, and z. For the `radius` property, this would not be possible, as it is a primitive type (`number`) without the possibility to subdivide it further.

With this selective change handler, the view can react in a granular fashion to changes in the model. If two or more properties of a model update simultaneously, for example, the `radius` and the `position`, all relevant view functions get executed, **but their order is not guaranteed**. This case should not be a problem as the view functions should not have order dependencies, but it is still something one must consider in the implementation. Usually, iterating over the properties happens in the same order as they are defined, which would mean that the execution order should be consistent. However, this behavior is no hard guarantee by the ECMAScript standard, so dependencies have to avoid this ambiguity. If a predefined order is needed, the update order must be handled one level higher.

Deciding which view function to call is not as simple as it might seem at first. The algorithm in which the handle module decides this is given in Algorithm 5.2. First, the algorithm discriminates between a simple and a complex change. The change-set is a simple change if it is one of the elementary change enums (Add, Update, Remove, None). If, on the other hand, the change is an object with properties, then it is classified as a complex change. In the former case, the simple change, the corresponding view

function is called, if it exists. However, if this is not possible because the view handler's corresponding entry is an object, all of its containing view functions are called. Whenever a complex change happens, two different scenarios may arise. First, we could also have a corresponding change-handler object for this complex change. In this case, the properties in the complex change-set trigger executions of the corresponding functions in the change-handler object. The second possibility is that there only exists a single view function instead of a change handler. In this case, the view function is simply called directly.

Algorithm 5.2: Resolving the view functions for a given change-set C.

Input: A view handler H, a model M, a change-set C, and view-requirements V^*
Output: View V_{new}

1 **Function** `resolveViewHandler(`H, M, C, V^*`)` **is**
2   **if** $C \in \{\mathsf{Change.Add}, \mathsf{Change.Update}, \mathsf{Change.Remove}, \mathsf{Change.None}\}$ **then**
3        $handler \longleftarrow H[\texttt{Change(}C\texttt{)}]$;
4   **if** $handler \neq$ null **then**
5           $V_{new} \longleftarrow$ `callAllHandlerCases(`*handler, M, V**`)`;
6   **end**
7   **else**
8        $V_{new} \longleftarrow$ `resolveComplex(`$H[\mathsf{Update}], \mathsf{Change.Update}, M, C, V^*$`)`;
9   **end**
10   **if** $V_{new} =$ null **then**
11        $V_{new} \longleftarrow V^*[0]$;
12   **end**
13   **if** Always $\in H$ **then**
14        $V^*[0] \longleftarrow V_{new}$;
15        $V_{new} \longleftarrow H[\mathsf{Always}](M, V^*)$;
16   **if** $V_{new} =$ null **then**
17           $V_{new} \longleftarrow V^*[0]$;
18   **end**
19   **end**
20   **return** V_{new}
21 **end**

Input: A change handler *handler*, a model M, and view-requirements V^*
Output: View V_{new}

1 Function `callAllHandlerCases`$(handler, M, V^*)$ **is**
2   **if** `isFunction`$(handler)$ **then**
3   **return** $handler(M, V^*)$
4   **else**
5   **foreach** $p \in handler$ **do**
6   `callAllHandlerCases`$(handler[p], M, V^*)$
7   **end**
8   **return** $V^*[0]$
9   **end**
10 end

Input: A change handler *handler*, a target change *target*, a model M, a change-set C, and view-requirements V^*
Output: View V_{new}

1 Function `resolveComplex`$(handler, target, M, C, V^*)$ **is**
    `// just flat change (object not complex)`
2   **if** $C = target$ **then return** `callAllHandlerCases`$(handler, M, V^*)$;
    `// different flat change we don't care about`
3   **if** $C \in \{$Change.Add, Change.Update, Change.Remove, Change.None$\}$ **then return**;
4   **if** `isFunction`$(handler)$ **then**
5   **return** $handler(M, V^*)$
6   **else**
7   **foreach** $k \in C$ **do**
8 $cur \longleftarrow C[k]$;
9   **if** $k \in handler$ **then**
10 $result \longleftarrow$ `resolveComplex`$(handler[k], target, M, cur, V^*)$;
11   **if** $result \neq$ null **then**
12 $V^*[0] \longleftarrow$ `merge`$(V^*[0], result)$;
13   **end**
14   **end**
15   **end**
16   **end**
17 end

5.8 Parsing

Translating the DSL to the model and vice-versa is what we define as parsing. If modules support parsing, they store their state visible to the user. Therefore users can modify the state through the DSL and thus configure the module's functionality. For modules to support parsing, the system requires them to provide a *parser* and *unparser* object. Modules that do not provide such objects are expected to handle their parsing themselves or refrain from participating in the parsing, effectively isolating their data from the user.

5.8.1 Parser - from DSL to Model

At the beginning of the visualization execution, the system passes the (potentially scoped) input DSL to all modules through the `INIT` message. The parser module intercepts this message and parses the input DSL by utilizing the parser object, which takes the DSL as input and outputs a corresponding model. The type definition of the parser object can be seen in Listing 5.10. At the center of this type definition is the `ParseFunction`, which simply takes any input and outputs an enforced *T*. A module providing a parser must supply a `Parser` of the generic type matching its model type. Therefore at the root level, the generic type *T* of the `Parser` is the model type. A `Parser` is either a `ParseFunction` that directly takes the input and outputs a matching *T*, or it is an object. If it is an object, it has to have an entry for every property of the given *T*. For every property a `ParseFunction` or alternatively a `Parser` object itself must exist.

```
type ParseFunction<T> = (v: any) => T;

type Parser<T> =
  | {
      [P in keyof T]: T[P] extends object ? Parser<T[P]> :
      ↪  ParseFunction<T[P]>;
    }
  | ParseFunction<T>;
```

Listing 5.10: Type definition of the parser.

This recursive definition allows module creators to implement parsing hierarchically. For example, suppose a parser implementation of the simple sphere model in Listing 5.4b. As input DSL we choose an object containing a `radius` of `string` and a `position` given as an array as seen in Listing 5.11a. Implementing this parser can happen in different ways. The most straightforward, but broad, approach is a simple parsing function outputting the complete model (`radius` and `position` properties). Thanks to the arrow function syntax, this can still look surprisingly concise as Listing 5.12a shows. A more granular approach, would be an object containing two functions, one for `radius` outputting a `number` and one for `position` outputting a `Vector3` (see Listing 5.12b). If instead of the input DSL of Listing 5.11a, we would have the input DSL Listing 5.11b

(`position` is an object with x, y, and z properties), we could go one step deeper. So instead of a function for the `position` property as a whole, we have functions for all properties of the `position`, namely x, y, and z as seen in Listing 5.12c. In that way, we completely delegate the parsing functions into the leave nodes.

```
{
  radius: "0.5",
  position: [0, 5, 2]
}
```

(a)

```
{
  radius: 0.5,
  position: { x: 0, y: 5, z: 2 }
}
```

(b)

Listing 5.11: Possible input DSLs for the simple sphere model of Listing 5.4b.

(a)
```
const modelParser: Parser<SphereModel> = raw => ({
  radius: Number(raw.radius),
  position: {
    x: Number(raw.position[0]),
    y: Number(raw.position[1]),
    z: Number(raw.position[2])
  },
});
```

(b)
```
const modelParser: Parser<SphereModel> = {
  radius: Number,
  position: raw => ({ x: raw[0], y: raw[1], z: raw[2] })
};
```

(c)
```
const modelParser: Parser<SphereModel> = {
  radius: Number,
  position: { x: Number, y: Number, z: Number }
};
```

(d)
```
const modelParser: Parser<SphereModel> = {
  radius: Number,
  position: parser.toVector3
};
```

Listing 5.12: Different parser implementations for a simple sphere model.

Now parsing a `Vector3` is a common operation and supporting both the array notation of Listing 5.11a and the property notation of Listing 5.11b offers the most flexibility for the user. This is incorporated into Listing 5.12d, the final and most compact, approach. Here the `toVector3` function of the `parser` module can accept an array or an object

and returns the resulting `Vector3`. By supplying such utility parsing functions, complex parsers can rely on a set of standard types that have the additional benefit of offering a consistent input DSL for the user.

5.8.2 Unparser - from Model to DSL

The unparser is the object specifying the inverse operation of the parser. Hence its definition is closely related to that of the parser (Listing 5.10) as seen in Listing 5.13. The system needs the parser's inverse operation for an essential mechanism. This mechanism is the process of getting changes out of the visualization and feeding them back into the DSL, thus enabling interactivity. Without this unparser, the user would only note changes in the system's state visually. However, to facilitate interactivity and taking snapshots (by copying the DSL), the user must also get a textual representation. In our demo examples (see Section 6.1), this is done by a split view, one side showing the DSL and the other one showing the visualization. Alternatively, the DSL could also solely run in the background and supply the user with the needed textual representation on demand (e.g., the user wants to take a snapshot, so he clicks a button that places the current DSL into the clipboard). Regardless of where the current DSL exists, it must update according to the visualization state.

```
type UnparseFunction<T> = (v: T) => any;


type Unparser<T> =
  | {
      [P in keyof T]: T[P] extends object ? Unparser<T[P]> :
      ↪  UnparseFunction<T[P]>;
    }
  | UnparseFunction<T>;
```

Listing 5.13: Type definition of the unparser.

Whereas the parser object is used by the `parser` meta-module, the unparser object is used by a regular module, the `source` module. In this way, users can easily decouple the back-translation functionality, when it is not be needed. This decoupling might be useful for performance-critical applications as the `source` module produces additional overhead per frame. The `source` module ignores the update stage and only acts in the view stage, as objects listening to changes of the DSL may be external ones. As such, the view function works as follows: Initially and every time the pipeline changes, the module creates a function object called `unparseFunction` that is responsible for taking the model as input and applying the hierarchically structured unparser of all modules that provide an unparse object. Modules that do not provide this object effectively hide their model information in this stage. This strategy of opting-in for the unparsing process is a conscious choice.

An alternative would be that parts of the root model that are not referenced in unparser objects are simply parsed into the DSL "as is." Consequently, modules that have model information that should not exist in the DSL (e.g. binary data) would have to take the route of opting out of the unparsing process somehow. Moreover, the unparsing in such a scenario could facilitate the `toJSON` function. If `JSON.stringify` encounters a `toJSON` function while operating on an object, it uses the function for converting the object to a JSON string instead of the usual generic approach. So the type system could require parsers to supply this function for all objects resulting from the parsing process. The described hypothetical method seems like a simple and intuitive solution for the unparsing mechanism. Nonetheless, it has several drawbacks that led to the decision of implementing our system differently. First, using the `toJSON` approach, while simplifying the unparsing implementation could complicate the normal update-loop invocation as it requires the creation of new models. Therefore all modules would have to make sure to persist potential `toJSON` properties. Second, using this simplification mechanism, the system would be restricted by the `JSON.stringify` implementation, which only offers rudimentary styling possibilities.

Instead, our approach with the unparser offers a few significant benefits: Because we do not want to restrict ourselves by outputting a formatted string, we write the unparsed model back into the root view-store as a JavaScript object. Any other code that has access to the view, including the system itself, can parse it in a way that suffices its purpose in the best way. For example, for GUIs, a beautified JSON can be emitted while a minified version can be sent for network transmission.

A relevant aspect of the unparsing system is its performance. Whenever the pipeline changes, the `unparseFunction` needs to be recomputed. Since pipelines rarely change, if at all, runtime overhead is low. For efficiently performing the actual unparsing, the process calculates a model *patch*. This patch is the subset of the model that is not directly or indirectly a children node of a `Change.None` in the respective change-set. Therefore the patch is usually smaller than the whole model after this step, which speeds up the subsequent computations. If the patch is an empty object or `null`, the unparsing process stops immediately. Next, the previously created `unparseFunction` takes the patch as input and computes the unparsed patch. This step identifies the modules' unparse functions that have overlapping models and executes the respective functions for each of them. Noteworthy in this step is the reduction of initial values. As initial values for the modules fill the model automatically (Section 5.5), they are not needed to reconstruct the visualization (e.g., if `radius` has the value 1 in the current model, but 1 is also the initial value). Therefore, the `unparseFunction` skips over model entries that are identical to the initial models provided by the modules. Again, if the resulting output of the `unparseFunction` is `null` or empty, the process aborts. Finally, the source gets updated, and the system notifies all listeners of the change.

CHAPTER 6

Evaluation

In this chapter, we evaluate our system in terms of the envisioned design goals (Section 3.1). First, we want to analyze whether the system accomplishes what we created it for: creating interactive 3D visualizations with a descriptive DSL. To analyze this, we provide case studies in Section 6.1, that answer the following questions:

1. Is the system applicable to representative use cases?

2. Is the system sufficiently expressive for creating interactive 3D visualizations?

As our approach to visualization is high-level and declarative, the system's execution clearly has some cost. The description needs to be interpreted and mapped to concrete visualization implementations. Therefore, the second part of the evaluation concerns the following question:

> Does the system have system-intrinsic overheads limiting the applicability in terms of performance?

We answer this question in Section 6.2 by measuring the performance and analyzing the asymptotic overhead that our system introduces.

Lastly, in Section 6.3, we compare our system to other approaches and describe how our approach fits into the larger landscape of visualization tools. This external evaluation focuses on the aspects of performance, expressivity, and usability. In general, this section answers the following question:

> In which situations does our language offer added value in comparison to other approaches.

6.1 Use-case Study

To evaluate our system's practicality for visualizing data, we show its different functionalities using case studies. The case studies focus on various aspects of the system. Therefore, this part of the evaluation examines whether the system is a useful tool for creating interactive 3D visualizations.

Each of the example applications focuses on a particular aspect of the problem domain:

- The first example (Section 6.1.1) shows the system's general working, how to write a simple interactive visualization by creating a custom module and how to extend it with custom inputs, domain logics, and interactions.

- In the second example (Section 6.1.2), we focus on real-world data and present the construction of a volume-visualization with our system. Since volumetric data-visualization is data-intensive, this showcases how the system handles vast amounts of data.

- In the last example (Section 6.1.3), we show how to create mapping functions with our system that map data to the visual attributes of a point cloud. Furthermore, we integrate an external visualization library for 2D visualizations and implement bidirectional communication with our system resulting in a linked view.

6.1.1 Case study #1 - Sphere

The sphere case-study uses the most basic setup: a single sphere (see Figure 6.1). Visualizing this sphere shows how a module operates inside the system using a simple example. The module managing the sphere defines a model, the immutable data store, for basic properties such as *position*, *color*, and *radius*. It parses these properties from the DSL and creates a geometrical *Three.js* object from them. It reacts to raycasting messages emitted from another module and potentially changes its model. When such a model change occurs, either from the sphere module itself or from outside, the module reacts to these changes and updates the *Three.js* object accordingly. Therefore, this example shows the whole system event-loop from the point-of-view of a module.

This case study also shows another aspect of the system: The standard approach to custom interactions in our system is to write a module that encapsulates the interaction logic. In the example, the sphere module reacts to messages from the raycasting module. Additionally to this standard approach, we also show a second method for creating interactivity by directly describing it inside the DSL. Using this method, a user can easily react to incoming messages and change the model without in-depth knowledge of the system.

```
 1  {
 2      "camera": {"rotation": [-20, 0, 0], "position": [0, 10, 20]},
 3      "sphere": {
 4          "radius": 2.5,
 5          "color": [0, 1, 0],
 6          "position": [0, 2.5, 0],
 7          "onHover": {"color": [1, 0, 0]}
 8      },
 9      "events": {
10          "CLICK[m.sphere.radius < 10]": "{ sphere: { radius: m.sphere.radius + 1 }}",
11          "CLICK[p.shiftKey]": {
12              "sphere": {
13                  "radius": 2.5,
14                  "color": {"r": 0, "g": 1, "b": 0}
15              }
16          },
17          "KEY_DOWN[p.key == '1']": "{ sphere: { color: { r: 1, g: 1, b: 0 }}}",
18          "KEY_DOWN[p.key == '2']": "{ sphere: { color: { r: 0, g: 1, b: 1 }}}",
19          "KEY_DOWN[p.key == '3']": "{ sphere: { color: { r: 1, g: 0, b: 1 }}}",
20          "KEY_DOWN[p.key == 'q']": "{ sphere: { color: { r: 1 - m.sphere.color.r }}}",
21          "KEY_DOWN[p.key == 'w']": "{ sphere: { color: { g: 1 - m.sphere.color.g }}}",
22          "KEY_DOWN[p.key == 'e']": "{ sphere: { color: { b: 1 - m.sphere.color.b }}}"
23      },
24      "pipeline": [
25          "scene",
26          "perspective-camera",
27          "sphere",
28          "render",
29          "raycaster",
30          "orbit-controls",
31          "key-listener",
32          "argumented-events",
33          "source"
34      ]
35  }
```

Figure 6.1: Case Study #1, Sphere. The left side shows the visualization description (enlarged in Listing 6.6), while the right side shows the resulting rendering. Changes to the description are immediately reflected in an updated visualization and vice versa.

Sphere Module

Next, we show the implementation of the module, which is exactly the code, visualization authors need to write to create the DSL described in Figure 6.1. We start by defining the type for the two data stores: the model and the view. Those type definitions are shown in Listing 6.1. For the model type-definition, we must identify the data types representing the state of the module. In this example, the model consists of the basic properties of radius, color, and position. Additionally, two optional properties `onHover` and `afterHover` can recursively contain models of the same type. As hinted by the name, the two properties will later define how the sphere should change when the raycast of the user's mouse enters or exits the sphere.

The view references the mutable state, which is required for rendering. It includes references to the required properties such as GPU buffers or DOM elements. As we use *Three.js* for rendering, we need references to some *Three.js* objects to change the sphere's appearance and deallocate the GPU buffers after use.

Now we have all the information required for constructing the module type-definition (see Listing 6.2). For the example, this definition consists of five fields. First is the message field. The model will only handle raycast messages, thus we do not define additional ones. Therefore, we reference the message object of the raycast module. Next is the model where we supply the `SphereModel`. We do not have any model requirements (no additional data from previous pipeline stages is needed), so we supply an empty array. For the view, we reference the `SphereView`, and finally, for the view requirements, we supply an array containing one element for the `SceneView`. The `SceneView` is

```
7   type SphereModel = {
8     radius: number;
9     color: Color;
10    position: Vector3;
11    onHover?: Partial<SphereModel>;
12    afterHover?: Partial<SphereModel>;
13  };
14
15  type SphereView = {
16    material: THREE.MeshStandardMaterial;
17    mesh: THREE.Mesh;
18  };
```

Listing 6.1: Type definition of the model and view of the sphere module. The first type defines all properties needed to define the scene, while the second one represents all data structures needed for rendering the view.

```
20  type SphereModuleType = {
21    message: raycast.RaycasterMessage;
22    model: SphereModel;
23    requirements: [];
24    view: SphereView;
25    viewRequirements: [scene.SceneView];
26  };
```

Listing 6.2: Type definition of the module type for the sphere module. All the needed type information for the module is supplied: The types of the messages it handles, a reference to its model and view type, and references to external models and views that are required.

essential because it provides a *Three.js* scene object where all rendered objects must be registered. By declaring the SceneView to be a view requirement, the view function can access the *Three.js* scene object stored in the SceneView. This scene object is the root of *Three.js*'s scene graph, where all rendered objects must be registered.

To guarantee valid data at all times, even if some values are not defined in the visualization specification, we provide an initial model (see Listing 6.3, Line 28). It contains the fallback values for the three required properties. Also, the initial view must be specified (Line 34). Other than for the model, the initial view is not an object, but a function returning the initial view. The function expects the model and view of the sphere as well as the SceneView as input. Taking these parameters, it creates the *Three.js* object for rendering the sphere and appends it to the scene object from the scene view.

```
28  const initModel: SphereModel = {
29    radius: 1,
30    color: Color.white,
31    position: Vector3.zero,
32  };
33
34  const initView = (
35    m: SphereModel,
36    _: SphereView,
37    sceneView: scene.SceneView
38  ) => {
39    let geometry = new THREE.SphereGeometry(1.0, 48, 24);
40    let material = new THREE.MeshStandardMaterial({
41      color: new THREE.Color(m.color.r, m.color.g, m.color.b)
42    });
43
44    let mesh = new THREE.Mesh(geometry, material);
45    mesh.position.set(m.position.x, m.position.y, m.position.z);
46    mesh.scale.set(m.radius, m.radius, m.radius);
47    sceneView.scene.add(mesh);
48
49    mesh.userData.module = module;
50
51    return { material, mesh };
52  };
```

Listing 6.3: Initial model and view for the sphere module. The initial model is a standard object, while the initial view is a function taking the model and other dependencies as input.

To translate between the DSL and the model, we need a parser and an unparser (see Listing 6.4). The former translates from the visual description to the model, and the latter translates the other way around. Enforced by TypeScript's type system, we have to provide the respective parsing functions for all required properties.

Finally, we define the core of the sphere module, the module object itself (see Listing 6.5). This object is the central point from which the type safety originates. By specifying the module type to be StandardModule<SphereModuleType>, the complete substructure and all enforcements onto its properties arise. First of all, we specify the name of the module. The core-module uses this name field for the look-up when reading the pipeline description. Following this property, the module object references the initial model, the parser, and the unparser. Subsequently, we define the update property. This property specifies the update function: Which messages result in what kind of model change. Here we need to handle all messages specified in the message field of the module type. In our case, these are the messages emitted by the raycasting-module:

```
54  const modelParser: parser.Parser<SphereModel> = {
55    radius: Number,
56    color: parser.toColor,
57    position: parser.toVector3,
58    onHover: parser.recursivePartialParse,
59    afterHover: parser.recursivePartialParse,
60  };
61
62  const modelUnparser: parser.Unparser<SphereModel> = {
63    radius: r => r,
64    color: c => [c.r, c.g, c.b],
65    position: p => [p.x, p.y, p.z],
66    onHover: parser.recursivePartialUnparse,
67    afterHover: parser.recursivePartialUnparse,
68  };
```

Listing 6.4: Parser and unparser of the sphere module. Recursively, the parser contains all information to provide the complete model, in this case by having a parsing function for each field.

RAYCAST_INTERSECTION_ENTER and RAYCAST_INTERSECTION_LEAVE. Handling them requires us to provide update handlers, i.e., functions that take the message payload, and the model as input and return a (potentially new) model. For both messages, we check whether or not to update the model depending on the incoming raycast message. If it contains a reference to our sphere, the update returns a modified model. In the other case, the update ignores the message and returns the input model. The view field of the module object describes the view handler, reacting to changes in the model. The Add property references what happens when the visualization description initially adds the sphere data into the model. Therefore, we supply the previously described initial view function. Inside the Update field, we further distinguish between the different fields that can update. In all cases, we get the model with the new values and update the *Three.js* objects accordingly. This setup completes the control flow from user change to visualization update. Therefore, if a user changes the model, for example, the sphere's position, the visualization updates. The last field of the view property is Remove. The function referenced under this field handles the clean-up of the *Three.js* resources so that the GPU can release the allocated buffers. After the view property, the last information the module needs are the paths for the data stores so that the module can find them. For the model paths, we only supply the first element, the sphere model's path. In contrast, the view-paths property has two elements: the path to the sphere view and the scene view-path.

At this point, the module is complete. Everything is set up so that the pipeline property in the visualization description can reference it by its name (see Listing 6.6 Line 27). Furthermore, users can parameterize the sphere module by the sphere property in the

```
70  export const module: StandardModule<SphereModuleType> = {
71    name: 'sphere',
72    initModel: initModel,
73    parser: modelParser,
74    unparser: modelUnparser,
75    update: {
76      RAYCAST_INTERSECTION_ENTER: (p, m) => {
77        if (m.onHover && p.userData.module === module) {
78          return { ...m, afterHover: m, ...m.onHover};
79        }
80        return m;
81      },
82      RAYCAST_INTERSECTION_LEAVE: (p, m) => {
83        if (m.afterHover && p.userData.module === module) {
84          return { ...m, ...m.afterHover };
85        }
86        return m;
87      },
88    },
89    view: {
90      Add: initView,
91      Update: {
92        radius: (m, v) => {
93          v.mesh.scale.set(m.radius, m.radius, m.radius);
94        },
95        color: (m, v) => {
96          v.material.color.setRGB(m.color.r, m.color.g, m.color.b);
97        },
98        position: (m, v) => {
99          v.mesh.position.copy(m.position as THREE.Vector3);
100       }
101     },
102     Remove: (_, v) => {
103       v.mesh.userData = {};
104       v.mesh.parent?.remove(v.mesh);
105       v.mesh.geometry.dispose();
106       v.material.dispose();
107     },
108   },
109   modelPaths: [['sphere']],
110   viewPaths: [['sphere'], viewPath(scene.module)],
111 };
```

Listing 6.5: Module definition of the sphere module. Here, all the implementation logic of the module, enfoced by its type from Listing 6.2, must be supplied.

```
1   {
2       "camera": {"rotation": [-20, 0, 0], "position": [0, 10, 20]},
3       "sphere": {
4           "radius": 2.5,
5           "color": { "r": 0, "g": 1, "b": 0 },
6           "position": [0, 2.5, 0],
7           "onHover": {"color": [1, 0, 0]}
8       },
9       "events": {
10          "CLICK[m.sphere.radius < 10]": "{ sphere: { radius: m.sphere.radius + 1 }}",
11          "CLICK[p.shiftKey]": {
12              "sphere": {
13                  "radius": 2.5,
14                  "color": { "r": 0, "g": 1, "b": 0 }
15              }
16          },
17          "KEY_DOWN[p.key == '1']": "{ sphere: { color: { r: 1, g: 1, b: 0 }}}",
18          "KEY_DOWN[p.key == '2']": "{ sphere: { color: { r: 0, g: 1, b: 1 }}}",
19          "KEY_DOWN[p.key == '3']": "{ sphere: { color: { r: 1, g: 0, b: 1 }}}",
20          "KEY_DOWN[p.key == 'q']": "{ sphere: { color: { r: 1 - m.sphere.color.r }}}",
21          "KEY_DOWN[p.key == 'w']": "{ sphere: { color: { g: 1 - m.sphere.color.g }}}",
22          "KEY_DOWN[p.key == 'e']": "{ sphere: { color: { b: 1 - m.sphere.color.b }}}"
23      },
24      "pipeline": [
25          "scene",
26          "perspective-camera",
27          "sphere",
28          "render",
29          "raycaster",
30          "orbit-controls",
31          "key-listener",
32          "argumented-events",
33          "source"
34      ]
35  }
```

Listing 6.6: DSL for describing the sphere visualization. It specifies information about the camera's orientation, the sphere module as defined in this section, custom events, and the pipeline.

visualization description (see Listing 6.6 Lines 3-8). Thus, the user can update the rendered visualization of a sphere in real-time. Last but not least, the module also reacts interactively to messages from the raycast module.

In summary, this sphere module example showed how users can create modular components for visualizing data. Therefore our goal of offering a system powerful enough to create new modular components that extend the DSL is accomplished.

Reacting to Messages inside the DSL

Our system allows enriching the visualization description by custom sub-DSLs for interactions. In this example we extend the system with the argumented-events module to support the CLICK and KEY_DOWN callbacks as shown in Listing 6.6. With this module,

users can react to arbitrary messages from within the DSL itself, providing a concise method to add interactivity quickly.

The module's task is simple: When a message passes through the module, it checks whether it has a property with the message type as the key inside its sub-model. If it has such a property, it interprets the value as a partial state of the model. The argumented-events module then applies this partial model to the real model, hence updating it. An example of this is shown in the case study's DSL (Listing 6.6) from Line 11 to 16. Here the CLICK message maps to an object with the same hierarchy as the root model. The module recursively walks down the root model and overwrites the properties provided in the partial model.

The "argumented" part of the module's name refers to the square brackets that all the event keys of the DSL example (see Listing 6.6) contain after the message type. As the name implies, these brackets contain arguments. Such arguments are valid JavaScript expressions that evaluate to a boolean value. Arguments have access to two variables: m referencing the model and p the payload of the message. Whenever the argument evaluates to true when the respective message arrives, the model gets updated. For example, with the CLICK message mentioned above, the argument checks if the shift key—stored in the payload because the payload is a MouseEvent—is pressed. Whereas the events module can only specify one change per message, the argumented-events module must handle multiple change specifications for the same message, as there could be different arguments. Looking at Listing 6.6, this need becomes evident. The description only specifies two messages: CLICK and KEY_DOWN, but these multiple times with different values. Without the arguments, the module could only react to the KEY_DOWN event in general. Therefore one would need to create different messages for each key. Instead, the allows reacting differently to the same message by inspecting the payload and, in this specific example, looking at the attached key type.

Additionally, the event entries in Lines 10 and 17-22 of Listing 6.6 show another feature of the argumented-events module. The values of the event keys are not objects, but strings. When an event occurs, the strings are evaluated just like the arguments. In contrast to the boolean value from the arguments, they define an object, which is the partial model that is later applied to the root model. In the same way as the arguments, the strings contain JavaScript expressions that can access the model m and the payload p. This setup results in a powerful way of creating custom interactions not possible with the object notation used by the event entry in Lines 11-16 of Listing 6.6. For example, the first event entry in Line 10 listens to the CLICK message emitted when the user clicks with the mouse. Its argument checks whether the sphere's radius is below 10. Only then it applies the value, which results in a dynamic evaluation and increments the sphere's radius by 1.

All in all, this approach for user-defined reactions to messages provides a broad range of interaction possibilities. They can describe changes to the model state in an expressive fashion. Nonetheless, they are concise and hence enable users to add functionality to the visualization quickly.

6.1.2 Case Study #2 - Volume

Whereas the first study showed a purposely simple example, this case study will show how our system operates on more complex visualizations. Therefore, this study will examine how feasible the system is for real visualization use cases. The subject of this case study is the visualization of volumetric data as used in disciplines such as medicine or non-destructive testing. More concretely, we will visualize the dataset from Gröller et al. [GGK05]. This volumetric dataset stores one-dimensional density data for each voxel totaling 652 MB. To visualize it, we will use a raycasting shader from *Three.js*. Besides the large dataset and the resource-intensive shader, we will be able to change shader uniforms from the visualization description. Lastly, with the example of texture loading, we also show how the DSL's JSON format enables a natural way of variable definition.

Figure 6.2: Case study #2, Volume. The DSL on the left side (enlarged in Listing 6.7) defines the volume's data source as well as texture used for the transfer function. The right side shows the final visualization of the stag beetle dataset from Gröller et al. [GGK05].

Handling large Data

Loading large datasets is no easy feat for a garbage-collected language like JavaScript, especially if it involves a transformation that allocates new objects. In this specific case, the raw volume data consists of unsigned integers with 12 bits padded to 16 bits per value. However, we use *Three.js*, whose WebGL volume shader expects 32-bit floating-point numbers in the $[0, 1]$ range. Consequently, we need to convert all values while loading the data and allocate an additional buffer twice the original input size. With the 652 MB input, we have at least an additional 1.304 GB buffer, resulting in 1.956 GB in total,

only focusing on the input data's storage. The runtime allocation of the whole system will easily surpass the 2 GB mark.

As this loading is such a complex task in this example, we have a whole module managing just that: the volume-loading module. It reads the input data from the specified file and converts it to a `FloatArray` asynchronously. Afterwards, it emits a `VOLUME_LOADED` message storing the float data in the message's payload. The module handles this message itself by storing the data in its model. This model change triggers the upload to a 3D texture that finishes with a `VOLUME_ARRIVED` message, signaling the volume module, that it can start rendering.

Rendering a Volume parameterized by the DSL

Now that we have the volumetric data on the GPU, we want to render it through a shader that is linked with the visualization description through the volume module. The shader used by this module is *Three.js*'s volume shader. Besides the data and the dimensions of the volume, the shader takes several parameters:

- The texture representing the transfer function
- A range, restricting the volume data values that map to the transfer function
- The renderstyle - a binary value switching between the two render modes: maximum intensity projection (MIP) and isosurface calculation (ISO)
- The threshold for the computation of the isosurface (if activated)

We want to modify these shader parameters from the visualization description, so that users can interactively modify them. The volume module, managing this linkage, has corresponding entries for its model (see Listing 6.7 Lines 2-8). When they change, the module updates the shader uniforms accordingly. Due to this linkage, users can change, for example, the texture to `gray`, which switches the transfer function to a grayscale color scheme. Likewise, users can change the `renderStyle` to `iso` instead of `mip` and immediately see the results when changing the threshold to another value.

Variable Fields in the DSL

One particular property in the `volume` path of the visualization description, the texture field, receives special treatment. It references the name of the texture that is itself defined in the DSL. The definition is found at the `texturePaths` path that the texture module manages. While the previously mentioned modules always apply a fixed grammar to the DSL, this module works differently. Instead of forcing the model to have rigid fields, it defines its model as an object containing fields with an arbitrary name that have paths to texture files as values. The texture module then takes this model and loads all the entries, while preserving the mapping of the file paths to the given names. In this way, other modules can request textures from the texture module by only providing the texture's user-defined name.

```
 1  {
 2      "volume": {
 3        "loadPath": "../data/stagbeetle832x832x494.dat",
 4        "colorRange": [0, 1],
 5        "renderStyle": "iso",
 6        "threshold": 0.2,
 7        "texture": "color"
 8      },
 9      "texturePaths": {
10        "color": "../data/cm_viridis.png",
11        "gray": "../data/cm_gray.png"
12      },
13      "camera": {
14        "position": [14, 54, -20],
15        "rotation": [-110, 13, 147],
16        "zoom": 4
17      },
18      "scene": {
19        "grid": true,
20        "backgroundColor": [0.1, 0.1, 0.1]
21      },
22      "pipeline": [
23        "scene",
24        "volume-loader",
25        "texture",
26        "orthographic-camera",
27        "volume",
28        "render",
29        "orbit-controls",
30        "source"
31      ]
32  }
```

Listing 6.7: DSL for describing the volume visualization. It specifies the source and visualization method for the volume data, paths to textures for the transfer functions, and finally camera and pipeline information.

Altogether, this case study shows that our system can handle not only simple demo visualizations but also practical visualization cases with large amounts of data. The example shows how to asynchronously load data and map DSL attributes to shader parameters that are completely transparent to the user. The underlying modules are based on the concepts introduced earlier and comprise approximately 250 lines of code in total, which is roughly what a hand-crafted alternative would need. Lastly, this case study shows how versatile the visual description inside the JSON format can be with the texture module's aid.

6.1.3   Case Study #3 - Point Cloud

The last of the three case studies concerns the rendering of a point cloud. For rendering this point-cloud, the case study contains a system where users can create complex mapping functions between the dataset and the points' appearance. Apart from showing a different visualization type, this case study also features the most prominent example of

the system's strengths. We integrated an external framework for creating 2D charts, *Vega* (see Section 2.2.3). This integration highlights the importance of the modularity that we have built into the system from the start. Integrating *Vega* enables users to specify *Vega* and *Vega-Lite* definitions inside our DSL. Moreover, our system communicates bidirectionally with the *Vega* visualization resulting in a linked view.
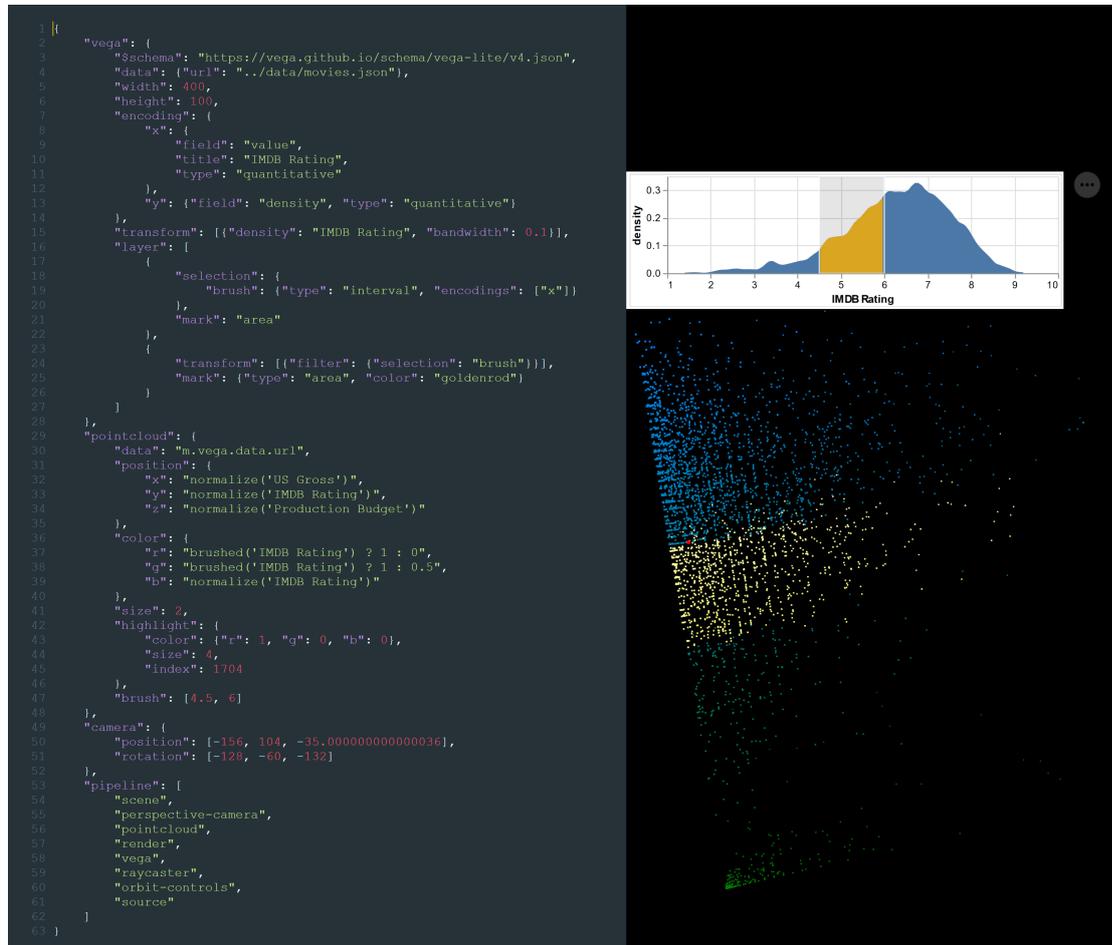
Figure 6.3: Case study #3, Point Cloud. The DSL on the left (enlarged in Listing 6.8) includes a full *Vega-Lite* description as well as encoding functions that transform the referenced point cloud data into the visualization on the right. The visualization shows the 3D point cloud as well as the *Vega-Lite* 2D chart resulting in a linked view, which links the brushing (visualized in yellow).

Loading the Point Cloud Data

As we already showed how loading large datasets work in Section 6.1.2, we will use a comparatively small dataset in this case study. Our choice is the movies.json dataset

71

from the *Vega* examples page [SMWH15]. It contains a JSON array with around 3 200 entries. Each entry represents a movie and is an object containing various fields like the IMDb rating, the production budget, or the gross profit.

As the *Vega* sub-model already references the file path to the dataset, we do not want to specify it again, violating the DRY (do not repeat yourself) principle. Thus, we reference the *Vega* file path within the `pointcloud` model. This approach can be seen in Listing 6.8, Line 38. The reference has the same syntax as in the previously presented argumented-events. Namely that `m`, referring to the model, points to the root of the model, and hierarchical paths are chains of the point-operator and the fields' names inside the model. Hence, with `m.vega.data.url` we start from the model, go inside the `vega` field then further into the `data` field and finally read the value with the `url` key. As a consequence, we arrive at the correct URL reference to the dataset, the same as *Vega* uses. Then the `pointcloud` module—the module containing all the logic concerning the point cloud—loads this dataset and stores it in its internal model.

Dynamic Encoding Functions

Rendering point clouds from a multivariate dataset involves mapping of the data to the appearance of the point. *Encoding functions*, as they are also called in various other visualization frameworks, create this mapping. While they can be as simple as referencing a single field, they can become quite complex, for example, by normalizing over all values or by applying mathematical transformations like logarithms.

For starting with the encoding functions, it is essential to know what properties we want to encode in the first place. We choose the most common properties, position, color, and size for this task. Accordingly, the `pointcloud` sub-model in the visualization description contains fields for these three properties (see Listing 6.8, Lines 39-49). Only the size property in the model references an encoding function directly. The other two properties are objects containing sub-encoding functions. While the `pointcloud` module also supports direct encoding functions for the position and color, they would need to return complex object types because they are not scalars like the size, but objects with three fields each. Having separate encoding functions for these fields allows users to easily map different data attributes to different axes or color values, respectively.

In principle, the encoding functions work similarly to the argumented-events module (see Section 6.1.1) in that they execute short snippets of JavaScript code that are defined as strings inside the DSL. Likewise, the module supplies the encoding functions with parameters. However, instead of only executing the encoding function once for every message, like the argumented-events module does, the `pointcloud` module must execute the encoding functions for every data element inside its dataset. Distinct is also the number of parameters provided to the function. The accessible parameters inside the encoding functions are:

- `model` (also accessible as `m`): the root model of the visualization, enabling the encoding functions to access everything that is in the visualization's current state

```
1   {
2     "vega": {
3       "$schema": "https://vega.github.io/schema/vega-lite/v4.json",
4       "data": {"url": "../data/movies.json"},
5       "width": 400,
6       "height": 100,
7       "encoding": {
8         "x": {
9           "field": "value",
10          "title": "IMDB Rating",
11          "type": "quantitative"
12        },
13        "y": {
14          "field": "density",
15          "type": "quantitative"
16        }
17      },
18      "transform":[{
19        "density": "IMDB Rating",
20        "bandwidth": 0.1
21      }],
22      "layer": [{
23        "selection": {
24          "brush": {
25            "type": "interval",
26            "encodings": ["x"]
27          }
28        },
29        "mark": "area"
30      }, {
31        "transform": [{
32          "filter": {"selection": "brush"}
33        }],
34        "mark": {"type": "area", "color": "goldenrod"}
35      }]
36    },
37    "pointcloud": {
38      "data": "m.vega.data.url",
39      "position": {
40        "x": "normalize('US Gross')",
41        "y": "normalize('IMDB Rating')",
42        "z": "normalize('Production Budget')"
43      },
44      "color": {
45        "r": "brushed('IMDB Rating') ? 1 : 0",
46        "g": "brushed('IMDB Rating') ? 1 : 0.5",
47        "b": "normalize('IMDB Rating')"
48      },
49      "size": 2,
50      "highlight": {
51        "color": { "r": 1, "g": 0, "b": 0 },
52        "size": 4
53      }
54    },
55    "camera": {
56      "position": [-126, 154, -44],
57      "rotation": [-111, -43, -119]
58    },
59    "pipeline": [ "scene", "perspective-camera", "pointcloud", "render", "vega",
60                  "raycaster", "orbit-controls", "source" ]
61  }
```

Listing 6.8: DSL for describing the point cloud visualization. It contains a full *Vega-Lite* specification. It further describes encoding functions for the point cloud data as well as camera attributes and a pipeline description.

- `data` : a reference to the whole dataset

- `index` (also accessible as `i`): the current index of the data entry inside the dataset

- `datum` (also accessible as `d`): the current data point, shortcut for `data[index]` .

- `aggregate` (also accessible as `agg`): a store holding aggregates like the minimum and maximum value of the fields over all entries.

- `brush` : an object storing a potential brush range ([`brush.min` , `brush.max`])

Additionally to these parameters, the encoding functions also have access to the following helper functions for routine tasks:

- `normalize(k)` : shortcut for normalizing a data field by dividing it by the maximum aggregate while ignoring the minimum aggregate
  defined as: `datum[k] / aggregate[k].max`

- `brushed(k)` : shortcut for checking whether a data field of the current data point lies inside the brushed range or not
  defined as: `datum[k] >= brush.min && datum[k] <= brush.max`

When encoding functions do not compile, they use fallback values from the module. This case can happen, for instance, when the encoding functions do not contain valid JavaScript or access non-existing variables. The same approach that provides the fallback values in such cases is also used for another feature: additional encoding functions for highlighted points. In this example, points are highlighted when the mouse hovers over them. To draw them visually distinct from the others, they use their own set of encoding functions. The DSL snippet (Listing 6.8) provides them in Lines 50-53. Although the highlight-specification in the example uses constants, they are still valid encoding functions. Of course, instead of these constants, JavaScript encoding functions can be used. Additionally to those, however, the highlight encoding functions have an extra parameter they can access, i.e., `original` . This parameter provides the highlight function with access to the value the respective default encoding function for that point would calculate. Due to this feature, users can, for example, take the original color that the point would have if it were not highlighted and increase the brightness proportionally.

Concluding, the computation of the encoding functions for the points looks as follows: For the position, we map three different fields to the axes, all of them normalized (see Listing 6.8, Lines 40-42). The x-axis shows the gross sales inside the US, the y-axis shows the IMDb rating, and the z-axis shows the production budget. Consequently, looking down the y-axis on the point cloud shows how the budget correlates with the movies' gross sales. The side views (i.e., the views on the x,y-plane and the z,y-plane) show how money influences the IMDb rating—an indicator of the movies' quality. All

three color channels operate on the IMDb rating field. Whereas the blue color channel represents the same information as the y-axis, the other channels check if the point's rating field lies inside the brushed range. Depending on this state, one of two fixed values is selected. Therefore, when the brushing range changes, the colors of the points update accordingly. This change can happen either through the visualization description or through *Vega*. Admittedly, also, the highlighting could have been implemented by this approach. Another parameter would have been passed to the shader. In this case, though, all points would need to be recalculated if the highlight changes.

In summary, we can see that the encoding functions offer users a broad capability for visually deciding how data is represented. As they are part of the visualization description themselves, users can change them interactively and immediately see the results, making them a valuable data exploration tool.

Vega Integration

The *Vega* integration happens through the `vega` module, whose model lives inside the `vega` scope. Everything inside this scope is a valid *Vega* description. Therefore, our DSL is factually a superset of *Vega*. In this case study, the *Vega* description visualizes the movies.json dataset by plotting a density function of the IMDb rating distribution. Additionally, the visualization features an intuitive brushing mechanism over the x-axis in the *Vega* chart.

The `vega` module also reacts to the changes of its model—the *Vega* description—by recomputing the *Vega* chart. Because of this recomputation, the users can change the model interactively. For instance, users can modify the `bandwidth` field, which specifies the smoothness of the underlying kernel density estimation, and the chart will automatically update. This feature already provides an interactive mechanism for modifying *Vega* visualizations. However, we want to go a step further by communicating from the *Vega* visualization to the point cloud and vice versa.

For the case study, we link the brushing state of the data. Hence, brushing the *Vega* chart changes the point cloud's brushed points, and changing the brushed range inside the `pointcloud` model also changes the displayed brush range in the *Vega* chart. Accomplishing this requires bidirectional messages. We decided to emit the same message format from both the `vega` and the `pointcloud` module. In this format, the messages have the message type BRUSH with a number tuple as payload. Emitting and receiving these messages works analogously to the message handling in Section 6.1.1. However, hooking up these messages with *Vega* demands a new method. For getting notified when a user changes the brush state, we use *Vega*'s signal listener, with which we react to the `brush` signal. When the listener is called, we simply emit a message with the `brush` signal's payload. Communicating in the other direction also uses signals. Calling the `signal` function on *Vega*'s view, with a subsequent call to `run`, which evaluates *Vega*, allows our system to change the brushing of the *Vega* chart when the brushing information updates from the outside. As a result, users can interact with both the 2D

visualization of *Vega* and the 3D visualization, and in both cases, the respective other visualization changes too.

In conclusion, we created a linked view using all the capabilities of the powerful *Vega* framework inside our system. With this linked view, we showed the system's modularity and extensibility.

6.2   Performance Evaluation

The performance evaluation analyzes the intrinsic overhead our system introduces compared to traditional, hand-crafted visualizations. This overhead arises from the computations that these traditional visualizations do not need to execute because they have precompiled logic. In contrast, a dynamic pipeline governs our systems' execution. Furthermore, the model's changes and the system's reaction to these changes are dynamic as well. Consequently, they demand performance-intensive runtime computations.

In general, our system introduces overhead in three locations:

- The update step, where messages travel through the pipeline, which produces a new model.

- The change-set computation by calculating the difference of two models so that the system knows how the state changed

- The re-play of the changes from the change-set to update the visualization.

In the following subsections, we study these different overhead-introducing computations by analyzing their impact on the total overhead. Finally, we will give performance characteristics and metrics of the system so that we can create an informed estimate on its practical limits.

6.2.1   Update Overhead

Instead of directly modifying the application state, our system stores an immutable representation of the state, the model. Modules react to user interactions during the update stage by creating a new immutable state, overwriting the old model. The resulting cost of these immutable updates depends on the data structure.

The prevalent data structures in the model are objects. In the update stage (see Section 5.7.1), the module can react to the user interaction and either return the old model or create a new one. How the module creates the new model is up to the module creator to decide. Usually, the `...`-operator will be used like:

```
{ ...m, modifiedKey: newValue }
```

Equivalents of this operator exist in various functional languages like the `with`-operator in F#. It allocates a new object and copies all properties from the old object into

the newly created one. Finally, it overwrites the specified keys with the new values. Copying the properties creates shallow copies. Therefore, the overhead only depends on the number of properties, not their values. It is $O(n)$ with respect to n, the number of properties.

Since updating also involves nested updates, the depth of the model tree needs to be considered. The reason for this is the recursive nature of the update mechanism. If, for example, a module updates a primitive that lies three levels deep in the module, an update would look like follows:

```
{ ...m, key1: { ...m.key1, key2: { ...m.key1.key2, key2: newValue }}}
```

Such a call results in a computational complexity of $O(n * d)$ with n being the maximum number of properties for all levels of objects and d being the updates node's depth. This overhead might seem large, but usually, n and d does not grow arbitrarily large and is bound by a small number. The n is bound because objects are not expected to expand their properties continually. In our experience, the depth of the model is typically limited to a few levels as working with deeper nested models becomes inconvenient for the user. However, a module could produce an excessive model depth d that is not reflected in the DSL. Such an excessive depth could arise, for example, if the model stores a linked-list where each node is one level deeper down in the model tree.

One particular use case that lends itself to a linked-list implementation is the realization of an undo/redo system storing, for example, the history of a sub-model. However, in such an implementation, a module would reuse the existing list and only insert a new head. Hence, the update overhead is constant ($O(1)$).

Besides objects, the model may also contain arrays. Large arrays occur prominently when raw data for the prospective visualization gets loaded into the model. When entries of the array change or the number of elements updates, modules must create new arrays and copy all elements so that the system can react to the change. If the array has an excessive size, the module must clone the complete array ($O(n)$) even if only one element changes. In such cases with frequent updates of large arrays, efficient immutable array data structures can be used (e.g., Straka [Str09]).

All in all, we can see that even though there exists an overhead in the update stage, it scales linearly to the model size at worst. Due to the reuse of the unchanged branches in the model and through persistent data structures when needed, the overhead becomes linearly with respect to the model's actual changes.

6.2.2 Diffing Overhead

To evaluate the cost of the difference calculation, we take performance measures on a synthetic test setup. The setup consists of an arguably inefficient implementation of a scene, where each visualized entity is stored as an extra object in the model. As a result of this analysis, we can empirically show how many changes can be naively processed

by the system, i.e., without further optimizing by utilizing better data structures. The overhead depends on the size of the model.

Therefore, we identified a parameter space of possible scenarios for the model. Generally, there are two extreme cases for the model that we base the benchmark on: a deeply structured model type and a flat structure with large fanout. In both cases, the worst-case scenarios are changes of the leaves, as these changes are computationally the most intensive ones to compute. Hence, we only change the leaf nodes. There are two parameters involved with the model's structure: the tree's depth and the number of nodes on each level. To keep the number of possible benchmarks low, we reduce the set of possible trees by choosing symmetric trees with the same number of children per node. The remaining parameter that could be relevant for the difference calculation is the number of changes between two possible model trees t_1 and t_2. For testing the worst case, we have to assume that the two models do not share any nodes, preventing speed-ups by early exits. Hence, the diffing algorithm must visit all nodes regardless and compare them recursively. Thus, we postulate that this parameter, the number of changes, does not impact the general performance, reducing the parameter space to the two former parameters.

To verify this hypothesis, we perform a first benchmark with a small number of different trees on five different change values each, as seen in Table 6.1. As shown in this table, the choice of children per node in relation to the tree depth is roughly fixed to sum up to 1 million nodes. This relationship exists to ensure that the total computation needed for each case has a common problem size. Later we normalize the benchmark results by the actual number of leaf nodes. We choose this number instead of the number of total nodes, as the inner nodes are only a structural overhead that does not transport the model's actual values. Of course, we, therefore, expect the deeper models to impose an additional overhead.

Depth	Children per node	Leaf / Value nodes	Inner nodes	Total nodes
1	1 000 000	1 000 000	1	1 000 001
2	1 000	1 000 000	1 001	1 001 001
3	100	1 000 000	10 101	1 010 101
6	10	1 000 000	111 111	1 111 111
10	4	1 048 576	349 525	1 398 101
20	2	1 048 576	1 048 575	2 097 151

Table 6.1: Tree variations and their node counts for a benchmark analyzing the impact of the change between the trees.

Even with preliminary test benchmarks, we noticed a performance inconsistency between the browsers on which we tested the diffing algorithm. With Chrome, the model creation takes longer while the diffing is faster than with Firefox. Therefore, we measured the benchmarks on both browsers, doubling the number of benchmarks to run. We selected Google Chrome (86.0.4240.183) because it is the most used browser at the time of writing. Additionally, the Chromium framework, which Google Chrome is based on, also serves

as the engine for multiple other browsers like Microsoft Edge and Opera. What we benchmark with Chrome is the V8 JavaScript engine that is part of Chromium. On the other hand, we use the Firefox Developer Edition (83.0b7) as an alternative browser. Its JavaScript engine is called SpiderMonkey.

All following benchmarks were performed on an HP Spectre x360 15 Convertible notebook, using Windows 10 Home 64-bit, Version: 19041.572 as the operating system. It has an Intel i7-8705G quad-core processor with a clock speed of 3.10 GHz. The RAM has a storage size of 16 GB. Finally, for graphical processing, it has an onboard and a dedicated GPU. These are not used in the benchmarks, however, as we are concerned about the additional CPU overhead, assuming optimal implementation of the graphic card calls.

For running the initial benchmarks we ran the six trees from Table 6.1 with the five different change values: 0, 0.25, 0.5, 0.75, and 1. In total, 30 different combinations of trees and change values exist. Doubling this, because of the two browsers, results in 60 cases to be checked. For this benchmark, we let each of these 60 cases run for 10 minutes. The results can be seen in Figure 6.4 for Chrome and in Figure 6.5 for Firefox.

We plot five box-plots for each tree case, one for every change-value, showing the diffing algorithm's speed for 1 million nodes. As can be seen in these benchmarks, there are significant outliers. We can see that sometimes, like in Figure 6.4a, the outliers lie inside the whiskers, and sometimes they are just outside, but most often above the upper whisker as in Figure 6.4b. In most cases, the interquartile range is tightly around the median except in Figure 6.4a and Figure 6.4d. Figure 6.4d is especially interesting because the three middle cases, change values of 0.25, 0.5, and 0.75, have a considerably higher median than the other two. The outliers also have a distinctive gap in the two deepest trees, Figure 6.4e and Figure 6.4f, which is probably due to different garbage collection stages. However, all in all, no stark deviation between the different change values can be noticed for the Chrome benchmarks.

The Firefox benchmarks look quite different from the Chrome ones, which we will analyze further with the second benchmark. In this benchmark, we can already see that, compared to the Chrome ones, not all tree cases are skewed. Instead, Figure 6.5d and 6.5e and to a lesser degree Figure 6.5c and 6.5f have more symmetric distributions. In the benchmarks with a lower depth, namely Figure 6.5a and 6.5b, the cases with 0% changes between the two trees stand out. While in Figure 6.5b, the distribution of the 0% case seems to be identical with the other cases except for a higher tail, the distribution is quite different in the Figure 6.5a case. Here, the median takes considerably longer, and also, the variance is larger.

However, the depth 1 case is unique as it is just a flat object, and because the stark deviation can only be observed in this one case for the Firefox browser. We conclude that the percentages of changes between the two trees have no relevant impact on the benchmarking. Therefore, our initial hypothesis is affirmed. We proceed with a fixed number of changes and select 50%, half the leaf nodes, as a neutral value.
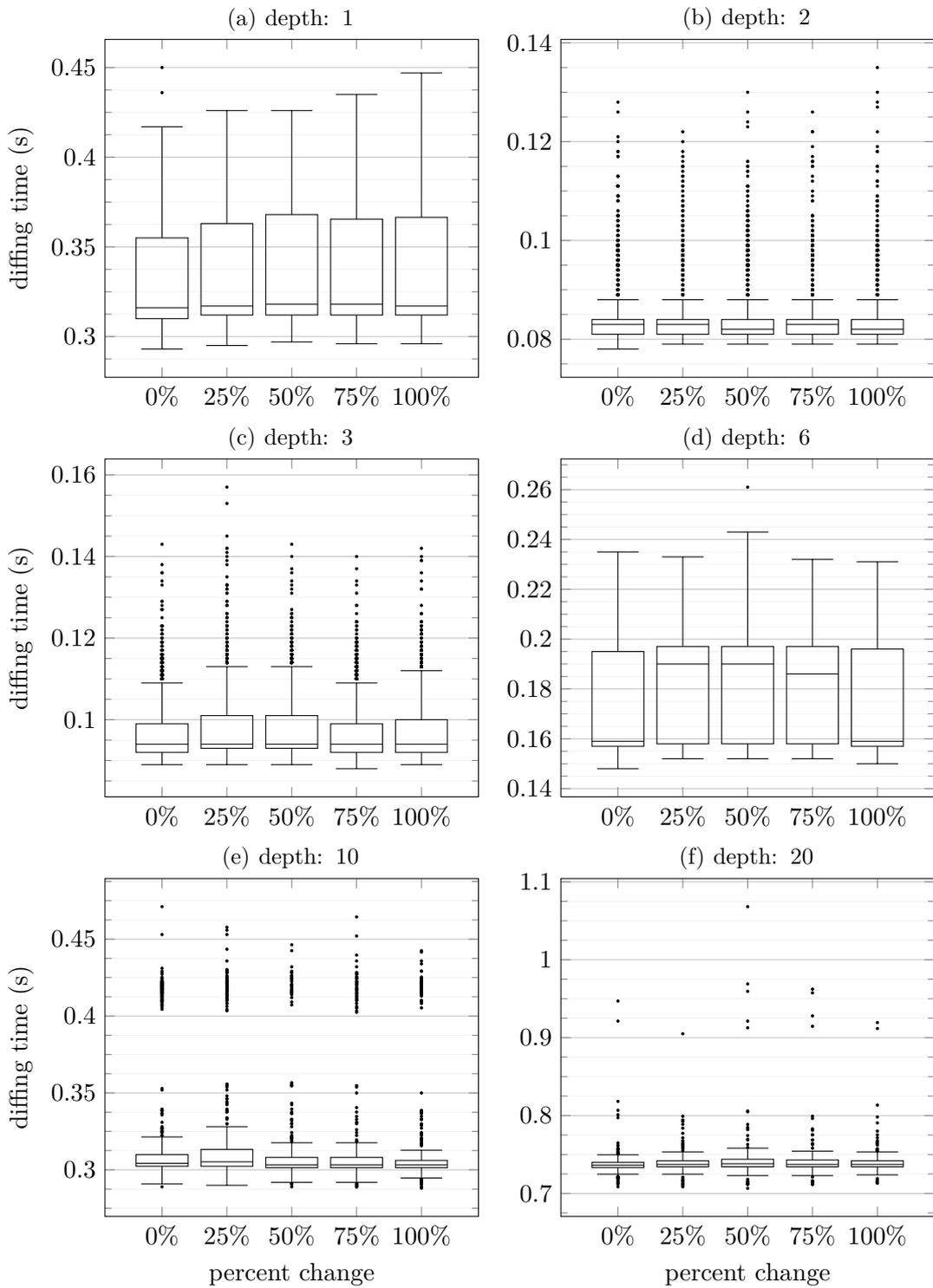
Figure 6.4: Speed of the diffing algorithm on the Chrome browser for tree variations from Table 6.1, including varying percentages of change.
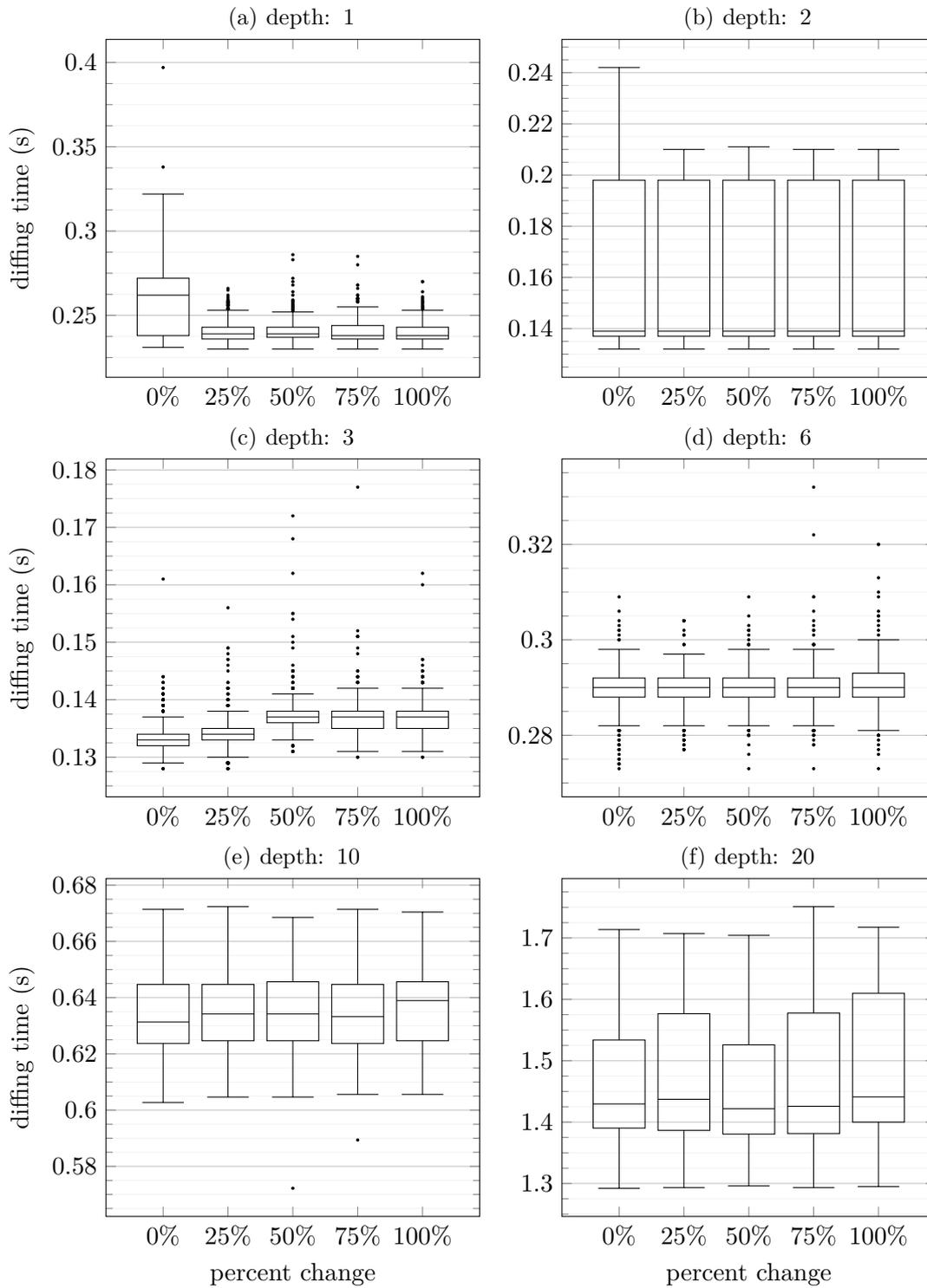
Figure 6.5: Speed of the diffing algorithm on the Firefox browser for tree variations from Table 6.1, including varying percentages of change.

For analyzing the relationship between the tree depth and the diffing speed more closely, we run a more excessive benchmark with a larger number of tree cases. Thus, we add more intermediate cases to the ones from Table 6.1. Thereby, we arrive at the 12 cases, shown in Table 6.2, with which we hope to get a better picture of the diffing algorithm's performance characteristics. As shown in the table, some combinations of tree depth with number of children lead to numbers of leaf nodes that are somewhat higher (depth 8, 9, and 13) or lower (depth 7), then the 1 000 000 or 1 048 576 respectively. In any case, we normalize the results by the actual number of leaves, as with the previous benchmarks. Nonetheless, we want to point out that these cases might differ slightly from trees with precisely one million leaves.

Depth	Children per node	Leaf / Value nodes	Inner nodes	Total nodes
1	1 000 000	1 000 000	1	1 000 001
2	1 000	1 000 000	1 001	1 001 001
3	100	1 000 000	10 101	1 010 101
4	32	1 048 576	33 825	1 082 401
5	16	1 048 576	69 905	1 118 481
6	10	1 000 000	111 111	1 111 111
7	7	823 543	137 257	960 800
8	6	1 679 616	335 923	2 015 539
9	5	1 953 125	488 281	2 441 406
10	4	1 048 576	349 525	1 398 101
13	3	1 594 323	797 161	2 391 484
20	2	1 048 576	1 048 575	2 097 151

Table 6.2: Tree variations and their node counts for a benchmark thoroughly analyzing the impact of the tree depth on the diffing time.

We ran each of the 12 tree cases 1 000 times on Chrome as well as on Firefox. The results of the benchmarks are shown in Figure 6.6. We plot the median number of processed leaf nodes per milliseconds against the tree depth. Hence, a higher number on the y-axis means that a higher number of actual values in the model can be processed with the respective depth. As with the first benchmarks, the result may not surprise at first glance, as a higher tree depth naturally results in more internal nodes that do not contain values, which in turn cause more recursive invocations for the difference algorithm. However, the plot shows some interesting peculiarities. First of all, tree depth 1, a single object where all leaves are properties, is an outlier with both browsers. Thus, putting one million properties in one object is slower than a more structured tree with fewer children per node. Specifically, with Firefox, trees with a depth between 2 and 6 perform better than the depth 1 case, and for Chrome, depth 2 up to the depth 10 case perform better. In general, Chrome performs faster on all tree cases except for the special case of depth 1. The difference between the browsers is also roughly constant.
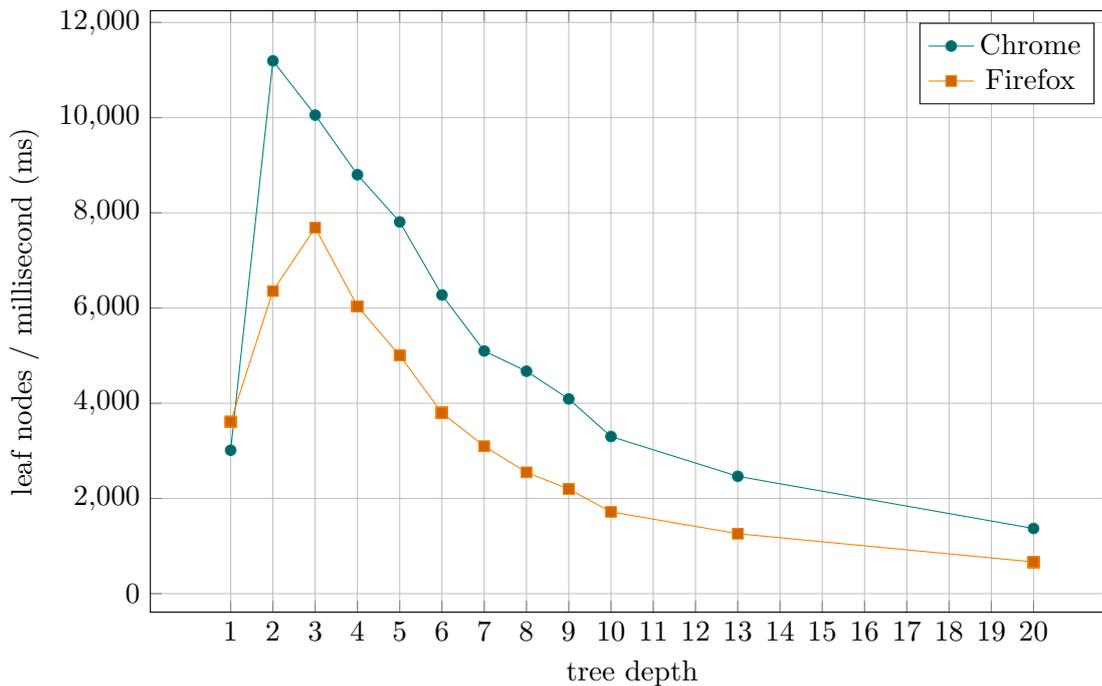
Figure 6.6: Number of leaf nodes the diffing algorithm (see Section 5.1) can compute per millisecond compared to the tree depth of the model.

Interestingly, Firefox also performs differently on the case with depth 2. It is faster than with a depth of one, but still below the peak performance at tree depth 3. Only after that, it follows the expected slowdown as the depth increases. Chrome, on the other hand, has its highest performance for trees with depth of two.

The takeaways from these comparisons are manifold. First, a flat hierarchy in the DSL should be strictly avoided, or else the depth 1 special case happens. Our implemented modules set positive examples by scoping their DSL inside a property named after the module instead of cluttering the root scope. Thus, we encourage module creators to follow suit, which gives a tree depth of at least two, avoiding the slow special case. On the other end of the spectrum, we can see that larger tree depths, for example, six and higher, are also increasingly slower than their shallower counterparts. This lower performance should not discourage model creators from using slightly higher depths when the scoping and clear separation calls for it. The reasoning behind this recommendation is that the benchmarks are an artificial test setup where all values lie in the highest depth of the tree, i.e., the leaves. Such a situation is generally not given in a practical case. The other significant aspect where the benchmark deviates from an expected assumption is the number of changes. Although the number of changes between the two trees are fixed at 50%, both trees still have internal nodes with unequal references. Therefore the diffing algorithm has to visit every leaf node in both trees and compare them. This

setup illustrates the worst-case scenario. Usually, real-life scenarios will enable the diffing algorithm to skip large parts of the tree as soon as a node has referential equality compared to the other tree.

With this difference between real-life and modeled worst-case scenario in mind, we present the actual benchmark numbers. The benchmark measures how fast the diffing algorithm can process leaf nodes. These leaf nodes correspond to the actual values in the system's model, while the inner nodes are just structural nodes. In Figure 6.6, we see the number of such leaf nodes that the diffing algorithm can process per millisecond if it has to investigate them and can not use early exits. To get a notion about what these numbers mean for the system, we will now calculate how many changes the model may contain maximally to produce 60 frames per second reliably. We start with the duration of one frame at 60 frames/s, which is $16.\dot{6}$ milliseconds. Given this time interval, we have to multiply the values from Figure 6.6 with it to arrive at the number of nodes for a ballpark figure. For example, we take the measurements for the tree with a depth of five.

The Chrome measurement is $7\,809$ nodes per millisecond, so we get $7\,809 * 16.\dot{6} \approx 130\,150$ nodes. In comparison, Firefox has a value of $5\,003$ nodes per millisecond, resulting in $5\,003 * 16.\dot{6} \approx 83\,383$ nodes. As an example use case, we take a scenario where positions of objects are updated every frame through the model system. A position comprises three values: x, y, and z. So every position uses three value nodes. Consequently, we can arrive at the number of objects we could continuously update every frame on our testing machine by dividing the former numbers by three. With Chrome, we could handle $43\,383$ objects, and with Firefox, $27\,794$. Even with this conservative estimate with a relatively high depth, the numbers of simultaneous changes our system can support are very assuring. In reality, most visualizations will probably have a lower depth than five, and their changes per frame will be far lower than the tens of thousands of changes we could support. For example, the highest number of changes per frame in the use cases of Section 6.1 is six, three values for the position, and three values for the rotation, which is four orders of magnitude smaller than the numbers in the benchmarks. Note that benchmark numbers do not mean that there can only be so many values in the final model. The measured bounds only represent the changes. Nor does that mean that these bounds limit the data displayed in our system in any way. After all, a non-naive implementation of the moving objects example would not store the position values as discrete elements in the model but instead as an array. Then this whole array would only comprise one value, and thereupon correspond to only one change. Hence, thousands of such arrays could be handled by the system per frame, at least by the diffing algorithm.

One aspect remains for evaluating the diffing algorithm, namely the generation of the change-set itself. As described in Section 5.1, the change-set can vary from being a single enum value to a complex object with the same structure as both models. Although the generation time is already included in the benchmark numbers of Figure 6.6, analyzing this process separately gives further insight into the JavaScript implementations' specifics. Therefore, we measure only the generation time of a tree with the same structure and size as the input trees. As the tree generation is also the performance bottleneck of

Section 6.2.1, the measurement serves a dual purpose by adding numerical evidence to the previous subsection's arguments.
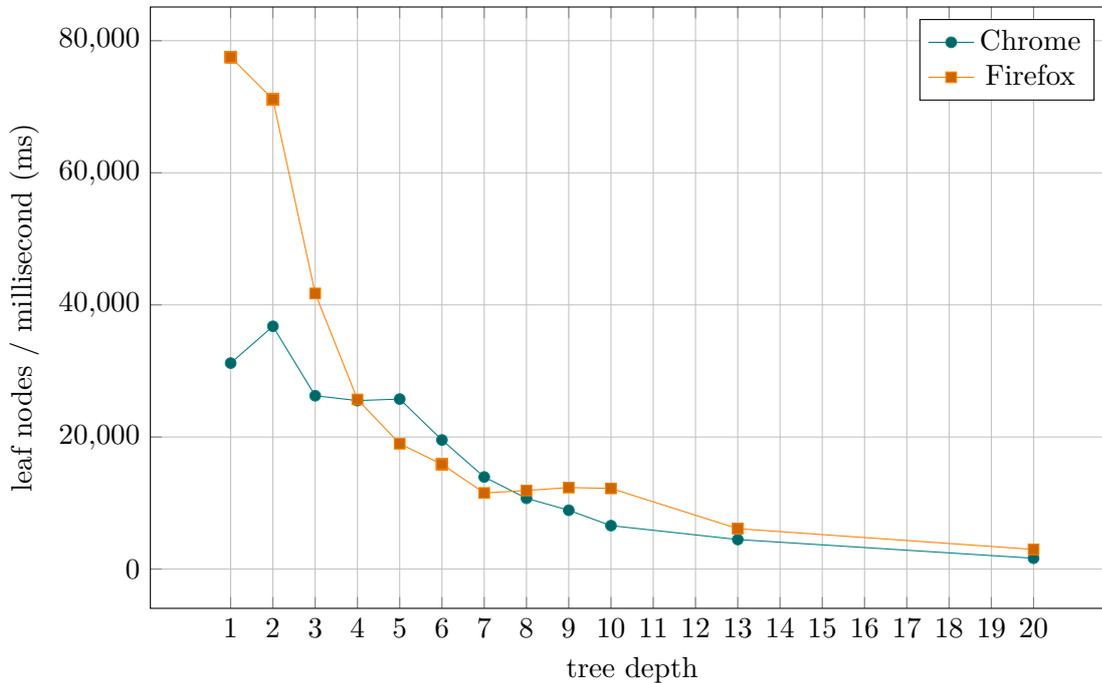
Figure 6.7: Number of leaf nodes that can be generated per millisecond compared to the tree depth.

Analogous to Figure 6.6, Figure 6.7 shows the median number of leaf nodes per millisecond in relation to the tree depth for generating a respective tree. An exponential decay curve is expected because of the relationship between the inner nodes and our symmetric trees' leaf nodes. Nonetheless, Figure 6.7 shows something quite peculiar. Although the difference calculations in Figure 6.6 were faster for Chrome in all instances, except for the depth 1 special case, tree generation time is running faster with Firefox except for the depth cases 5 to 7. Especially the low depths 1 to 3 are way faster with Firefox than Chrome. This phenomenon raises the question why Chrome is still faster than Firefox concerning the complete difference calculations even though the tree generation is an integral part of it. The other operations that the tree generation does are calling a recursive algorithm, traversing a tree, chasing pointers, and making equality comparisons. Hence, we conclude that these operations must be even faster so that, in total, they compensate the speed penalty that Chrome has in generating the change-set.

Furthermore, the numbers in Figure 6.7 give an estimate of the model's maximum size without reused objects inside the update step. The resulting subset of the model is the subset that carries all structural and value updates. Comparing Figure 6.7 with Figure 6.6 shows, that the model's size can almost be an order of magnitude larger than

the change-set computation allows. Consequently, the plot proves that the difference calculation is indeed the stricter bound of our system's performance.

Summarizing, given the benchmarks, we can now accurately judge our system's capabilities. Even with conservative problem sizes, these capabilities suffice for expected real-world uses.

6.2.3 Overhead for applying Changes

If the diffing algorithm detects changes, the resulting change-set (benchmarked in Section 6.2.2) travels through the pipeline and causes reactions from the modules, which in turn changes the mutable data store, i.e., the view. Compared to handcrafted systems, the overhead introduced here is due to the dynamic mapping of change handlers to the change-set. During runtime, our system has to evaluate which change handlers to execute, given an arbitrarily complex change-set. When a property at depth $= d$ exists, but no corresponding change handler, the system has to look into the change handlers on level $d - 1$ and so on until a matching change handler is found.

This behavior is illustrated, for example, by the color change handler from Listing 6.5, Line 95 to Line 97. If somebody edits only the color's red value, the system does not find a direct change handler for that. Instead, it has to go one level higher where it finds the change handler for the whole color, which it then calls.

In general, the number of changes to be applied is bounded by the change-set itself. Furthermore, the overhead per change is dependent on the module that handles the change. The module should be optimized in such a way that the change handlers fit to the occurrence and size of the changes inside it. For example, if a module expects changes inside its leaves, the change handlers should form a similar structure, with view functions in the leaves (see Section 5.7.2). Conversely, if changes always deliver entirely new models, it might not make sense to create fine-grained change handlers and instead use a single view function as change handler.

Concluding, the performance of applying changes to the underlying view data structures depends on the granularity of the change handlers. Whenever changes are managed by a corresponding change handler directly, no significant runtime overhead is introduced. If a handler cannot manage the change directly, the system needs to fall back to a coarser change handler. Optimizations and the trade-off between expensive coarse-grained changes and numerous micro-changes need to be taken care of by the module author.

Therefore, our system's architecture does not introduce a direct overhead, but rather the modules themselves if they are not optimally implemented in relation to the changes they encounter during runtime. In conclusion, the overhead for applying changes falls into the module author's responsibility and not into the system's internal overhead.

6.3 External Evaluation

At the beginning of this external evaluation, we want to note that our system is not designed as a direct replacement for any existing system. Instead, it aims to fill a niche that other systems do not cover. As seen in Section 6.1.3 with the example of *Vega-Lite*, our system is also capable of creating an overarching structure for combining existing visualization tools under a common framework.

The prominent use case for our system is the usage as a communication medium for visualizations. Another benefit is the creation of visualizations without in-depth knowledge about computer graphics, which enables quick prototyping capabilities. Additionally, our system gives visualization designers the ability to create modules for custom domain-specific visualization use cases. Domain experts can then use these custom modules for their specific visualization needs.

For sharing visualizations and quick prototyping, a popular tool in the scientific community are Jupyter notebooks, as mentioned in Chapter 1. Although Jupyter notebooks are stored in JSON files, most notebooks use Python as a scripting language instead of JavaScript. As a result, libraries like Altair [Dev19] offer methods for creating visualizations using the Vega-Lite syntax in Python. However, our system focuses on interactive 3D visualizations. There also exist Python libraries that offer interactive 3D capabilities like for Jupyter notebooks like Matplotlib [HDF+21] and IPyvolume [Bre17]. Both of these libraries, however, use an imperative design with rigid, predefined functions. Instead, our approach is modular, declarative, and allows custom interactions.

Another difference between these solutions is, that our system is self-contained inside the JSON file. In contrast, Matplotlib and IPyvolume exist in Python snippets that themselves need to be included inside a Jupyter notebook. Nevertheless, because the native format of Jupyter notebooks is JSON, our language can be easily integrated. Therefore, users can even combine the libraries' strengths by using Matplotlib or IPyvolume when quick, imperative plotting is preferred, and our system when a declarative approach, modularity, and custom interactions are needed.

Performance-wise, IPyvolume uses the same framework as our system in the backend, i.e., *Three.js*, to create a WebGL visualization. Therefore, the performance differs only on the CPU site covering the visualization logic. Hence, the analysis from Section 6.2 applies.

The second category we evaluate our system against are tools that visualization designers can use to craft environments where domain experts can create and modify visualizations related to their domain. As described in Section 2.4, examples of such tools are MeVisLab and Inviwo. Both systems support the creation of highly configurable visualizations through network interfaces. Since these systems are both native applications written in C++, the compiled code has a lower abstraction level than the interpreted JavaScript code.

A drawback of MeVisLab and Inviwo is the use of proprietary file formats for the visualizations. Instead, our DSL is defined inside JSON files, which numerous existing

tools and languages can directly use and manipulate. In combination with the fact that our system can be integrated as a JavaScript library, our visualization system has a lower threshold for users to act as a visualization exchange format.

In general, native visualization systems need to be installed locally first, while users can directly access our system in the web browser. Not only does this complicate the initial time of opening such a visualization, but it also brings the problems of platform dependence with it.

In summary, while our system might not be the most performant compared to native applications, it offers high accessibility and shareability through its embedding in the web ecosystem.
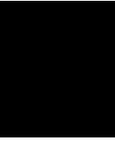
Conclusion

7.1 Contribution

In this thesis, we proposed a novel domain-specific language. It enables the definition of 3D visualizations and the creation of custom interactions without knowledge about computer-graphics concepts such as buffers or shaders. The expressiveness of the DSL enables users to configure all stages of the visualization pipeline. Furthermore, our language forms a dynamic visualization description inside the JSON format, aiding the shareability of 3D visualizations.

Furthermore, we developed a modular system that can be manipulated using the DSL. Our system allows users to extend the language themselves by creating new modules. These generically reuseable modules were made possible through the type system. As such, our language has a modular semantic instead of a fixed set of instructions.

We integrated the DSL and the visualization system into a visualization creation environment. It shows the visualization itself plus the system's current state as DSL and updates it accordingly when changes appear. With this environment, users can easily explore the language, modify values, create new interactions, and observe the updates taking place inside the system. Additionally, users can take snapshots of the visualization's current appearance by copying the DSL. We achieved this functionality through the system's immutable state, an aspect influenced by functional paradigms.

We further explored how a visualization system on the web can look like that handles a declarative DSL while interfacing with an imperative environment. With this interface, we bridged the gap between the immutable visualization snapshot represented in the DSL and the imperative implementation of the used graphics library.

We showed the versatile capabilities of our system by exploring various aspects of it with case studies. These case studies ranged from simple examples of custom modules to

data visualizations of real use cases. They also contained an example of integration with another visualization framework, i.e., Vega, which highlighted our system's modularity. Finally, we performed quantitative benchmarks and theoretical analyses of the additional overhead our language introduces.

7.2 Lessons Learned

With TypeScript as our language of choice for the system, we can use modern ECMAScript functionalities like the `...`-operator. However, compiling such modern syntax features to older JavaScript versions often results in bloated and potentially inefficient code. For example, using the `...`-rest-operator when providing function arguments compiles to a loop executing successive `apply` calls. This backward-compatible compilation also confuses the JavaScript debuggers and thus complicates the debugging process.

Additionally, even though we get static types with TypeScript, our system's dynamic nature requires us to inspect types during runtime continually. This problem gets even further aggravated because checking whether a variable contains an object or a value type requires string comparisons. As we do not know how models change between pipeline invocations, we must execute such string comparisons frequently inside the diffing algorithm and at other places in the code. Consequently, our system could be even faster without these drawbacks.

7.3 Future Work

One approach to alleviate the problems of Section 7.2 could be to implement the system's performance-critical bottlenecks like the diffing algorithm or the pipeline execution, in web assembly. Such an implementation could also ease integrating modules programmed in other languages having web assembly as compile target.

Another area for further research is the module system we envisioned. We set out with a module as a simple function taking input and providing an output. However, more and more of our guarantees required constructs where data and instructions interact and thus needed ever tighter relations. We forced a separation between the two, and in the final implementation, the update and view functions have no direct notion to which module they belong. Sometimes this separation seems unintuitive, and indeed it might make sense to provide a reference of the module itself to those functions. The alternative to this separation would be to handle the modules as classes. Such a change would probably also add additional compile-time enforcements and type checks from TypeScript as they are a prominent feature of the language.

List of Figures

List of Tables

List of Algorithms

List of Listings

98

Acronyms

API Application programming interface. 3, 13, 22, 27, 36

CPU Central processing unit. 22, 79

CSS Cascading Style Sheets. 13, 17

DOM Document Object Model. 12, 13, 14, 15, 21, 31, 35, 38, 43, 61, 92

DSL Domain-specific language. 3, 5, 7, 8, 9, 12, 25, 27, 28, 29, 30, 31, 33, 34, 35, 44, 45, 47, 54, 55, 56, 57, 59, 60, 61, 63, 66, 67, 68, 69, 70, 71, 72, 74, 75, 77, 83, 87, 89, 91, 92, 97

EBNF Extended Backus–Naur form. 27, 28, 97

FIFO first in, first out. 38, 43

GPU Graphics processing unit. 2, 9, 22, 31, 35, 61, 64, 69, 79

GUI Graphical user interface. 15, 22, 23, 36, 38, 57, 91

HTML Hypertext Markup Language. 13, 15, 20, 21

I/O Input/Output. 31

JSON JavaScript Object Notation. 2, 15, 27, 28, 33, 34, 57, 68, 70, 72, 87, 89, 97

REST Representational state transfer. 27

SVG Scalable Vector Graphics. 13, 14, 15

URL Uniform Resource Locator. 16, 72, 91

W3C World Wide Web Consortium. 13

XML Extensible Markup Language. 13

Bibliography

[Abr21]   Dan Abramov. Redux. `https://redux.js.org/`, 2021. Accessed: 2021-03-30.

[AG21]   MeVis Medical Solutions AG. mevislab. `https://www.mevislab.de/`, 2021. Accessed: 2021-04-02.

[aut21]   Graphviz authors. The dot language. `https://www.graphviz.org/doc/info/lang.html`, 2021. Accessed: 2021-01-26.

[BH09]   Michael Bostock and Jeffrey Heer. Protovis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1121–1128, 2009.

[BOH11]   Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

[BRC$^+$19] Rene Brun, Fons Rademakers, Philippe Canal, Axel Naumann, Olivier Couet, Lorenzo Moneta, Vassil Vassilev, Sergey Linev, Danilo Piparo, Gerardo GANIS, Bertrand Bellenot, Enrico Guiraud, Guilherme Amadio, wverkerke, Pere Mato, TimurP, Matevž Tadel, wlav, Enric Tejedor, Jakob Blomer, Andrei Gheata, Stephan Hageboeck, Stefan Roiser, marsupial, Stefan Wunsch, Oksana Shadura, Anirudha Bose, CristinaCristescu, Xavier Valls, and Raphael Isemann. root-project/root: v6.18/02, August 2019.

[Bre17]   Maarten A. Breddels. `https://ipyvolume.readthedocs.io`, 2017. Accessed: 2021-04-03.

[CC13]   Evan Czaplicki and Stephen N Chong. Asynchronous functional reactive programming for guis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation-PLDI'13*. ACM Press, 2013.

[CER20]   CERN. `https://swan.web.cern.ch`, 2020. Accessed: 2020-12-15.

[CMS99]   Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1999.

[Con20]   Web3D Consortium. `https://www.web3d.org`, 2020. Accessed: 2021-03-24.

[Cza12]   Evan Czaplicki. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University*, 30, 2012.

[Cza20]   Evan Czaplicki. `https://elm-lang.org/`, 2020. Accessed: 2020-12-15.

[Dev19]   Altair Developers. `https://altair-viz.github.io/`, 2019. Accessed: 2021-04-03.

[Dis20]   Distill. `https://distill.pub`, 2020. Accessed: 2020-12-15.

[ECM21]   ECMA. Ecma-404, the json data interchange syntax, 2nd edition, december 2017. `https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf`, 2021. Accessed: 2021-02-11.

[EH97]   Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN International Conference on Functional programming*, pages 263–273, 1997.

[Fjc21]   OpenJS Foundation and jQuery contributors. jquery. `https://jquery.com/`, 2021. Accessed: 2021-03-26.

[Fow04]   Martin Fowler. Inversion of control containers and the dependency injection pattern. `https://www.martinfowler.com/articles/injection.html`, 2004. Accessed: 2021-02-04.

[GGK05]   Eduard Gröller, Georg Glaeser, and Johannes Kastner. Stag beetle. `https://www.cg.tuwien.ac.at/research/publications/2005/dataset-stagbeetle/`, 2005.

[HB10]   Jeffrey Heer and Michael Bostock. Declarative language design for interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1149–1156, 2010.

[HDF⁺21]   John Hunter, Darren Dale, Eric Firing, Michael Droettboom, and the Matplotlib development team. `http://matplotlib.org`, 2021. Accessed: 2021-04-03.

[HSP09]   Frank Heckel, Michael Schwier, and Heinz-Otto Peitgen. Object-oriented application development with mevislab and python. *Informatik 2009–Im Focus das Leben*, 2009.

[Inc21]   Facebook Inc. React. `https://reactjs.org/`, 2021. Accessed: 2021-03-30.

[JSS+19]   Daniel Jönsson, Peter Steneteg, Erik Sundén, Rickard Englund, Sathish Kottravel, Martin Falk, Anders Ynnerman, Ingrid Hotz, and Timo Ropinski. Inviwo - a visualization system with usage abstraction levels. *IEEE Transactions on Visualization and Computer Graphics*, 26(11):3241–3254, 2019.

[JSS+20]   Daniel Jönsson, Peter Steneteg, Erik Sundén, Rickard Englund, Sathish Kottravel, Martin Falk, Anders Ynnerman, Ingrid Hotz, and Timo Ropinski. `https://inviwo.org/`, 2020. Accessed: 2020-12-15.

[KLM+97]   Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242. Springer, 1997.

[Knu84]   Donald Ervin Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[KP88]   Glenn Krasner and Stephen Pope. A coolbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, 1988.

[KRKP+16]   Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.

[Mor12]   Kenneth Moreland. A survey of visualization pipelines. *IEEE Transactions on Visualization and Computer Graphics*, 19(3):367–378, 2012.

[MRNL20]   Alexander Mordvintsev, Ettore Randazzo, Eyvind Niklasson, and Michael Levin. Growing neural cellular automata. *Distill*, 2020. https://distill.pub/2020/growing-ca.

[NCP02]   Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64, 2002.

[PG15]   Fernando Perez and Brian E Granger. Project jupyter: Computational narratives as the engine of collaborative data science. `http://archive.ipython.org/JupyterGrantNarrative-2015.pdf`, 2015. Accessed: 2021-03-24.

[RBGH14]   Peter Rautek, Stefan Bruckner, M Eduard Gröller, and Markus Hadwiger. Vislang: A system for interpreted domain-specific languages for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2388–2396, 2014.

[RBH⁺11]   Felix Ritter, Tobias Boskamp, André Homeyer, Hendrik Laue, Michael Schwier, Florian Link, and H-O Peitgen. Medical image analysis. *IEEE pulse*, 2(6):60–70, 2011.

[Rea89]   Chris Reade. *Elements of functional programming.* Addison-Wesley Longman Publishing Co., Inc., 1989.

[SMWH15]   Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. movies.json dataset. `https://vega.github.io/vega-lite/examples/data/movies.json`, 2015. Accessed: 2021-03-31.

[SMWH16]   Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, 2016.

[SRHH15]   Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):659–668, 2015.

[STH02]   Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, 2002.

[Str09]   Milan Straka. Optimal worst-case fully persistent arrays. *Trends in Functional Programming*, 2009.

[SWH14]   Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. Declarative interaction design for data visualization. In *Proceedings of the 27th annual ACM Symposium on User Interface Software and Technology*, pages 669–678. ACM, 2014.

[W3C17]   W3C. Xsl transformations (xslt) version 3.0. `https://www.w3.org/TR/xslt-30/`, 2017. Accessed: 2021-03-26.

[Wan91]   Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, 1991.

[Wic10]   Hadley Wickham. A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28, 2010.

[Wil12]   Leland Wilkinson. The grammar of graphics. In *Handbook of Computational Statistics*, pages 375–414. Springer, 2012.

[WTH01]   Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven frp. *Lecture Notes in Computer Science*, 2257, 12 2001.