Responsive Voronoi Treemaps with VoroTree and VoroLib

Christopher Oser

Responsive Voronoi Treemaps with VoroTree and VoroLib

Christopher Oser B.Sc.

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Program: Computer Science

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dr. Keith Andrews
Institute of Interactive Systems and Data Science (ISDS)

Graz, 23 Feb 2022

Responsive Voronoi Treemaps mit VoroTree und VoroLib

Christopher Oser B.Sc.

Masterarbeit

für den akademischen Grad

Diplom-Ingenieur

Masterstudium: Informatik

an der

Technischen Universität Graz

Begutachter

Ao.Univ.-Prof. Dr. Keith Andrews
Institute of Interactive Systems and Data Science (ISDS)

Graz, 23 Feb 2022

Diese Arbeit ist in englischer Sprache verfasst.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The document uploaded to TUGRAZonline is identical to the present thesis.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Dokument ist mit der vorliegenden Arbeit identisch.

Date/Datum

Signature/Unterschrift

Abstract

This thesis presents the VoroTree application and the VoroLib library for interactive Voronoi treemaps. Voronoi treemaps are a space-filling information visualization technique for hierarchical information, based on recursive subdivision of Voronoi regions.

VoroTree is a self-contained web application for exploring hierarchical data with Voronoi treemaps. It is also available as a hosted demo application on the web, and can be built as an installable native package for common desktop platforms. VoroLib is a JavaScript library for integrating Voronoi treemaps into other applications. Both projects are open source, built in TypeScript, and responsive. PixiJS is used for drawing and parts of D3 are used for data handling.

Kurzfassung

Diese Diplomarbeit präsentiert die VoroTree Webapplikation und die VoroLib Library für interaktive Voronoi Treemaps. Voronoi Treemaps sind eine raumfüllende Informationsvisualisierungsmethode für hierarchische Daten und basieren auf der rekursiven Aufteilung von Voronoi Regionen.

VoroTree ist eine alleinstehende Webapplikation die das Durchsuchen von hierarchischen Daten mittels Voronoi Treemaps ermöglicht. Die Applikation wird online bereitgestellt, sowie auch als alleinstehendes Programm für den Offline Gebrauch auf traditionellen Computern. VoroLib ist eine JavaScript Library, die ermöglicht Voronoi Treemaps in andere Webapplikationen einzubinden. Beide Produkte sind Open Source, in TypeScript implementiert und skalieren automatisch plattformübergreifend auf alle Bildschirmgrößen. PixiJS wird für das Zeichnen der Visualisierungen verwendet. Teile der D3 Library werden für das Verarbeiten der Daten verwendet.

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

I want to thank my advisor, Keith Andrews, for his continued support throughout my studies and this thesis. Both my bachelor's thesis and my master's thesis were accompanied by Prof. Andrews and I learned much through the process of working on them with him.

I want to mention Markus Ruplitsch, Lisa Weissl and Romana Gruber for helping me construct the IVT for the 2020 Information Visualization course.

I also want to mention Paul Höfler and Florian Marcher for providing feedback on VoroLib.

I want to thank my mother, a strong woman who always supported me in all my endeavors and without whom I wouldn't be here.

Many thanks also go out to all of my grandparents who have supported me throughout my studies and made it possible for me to focus on school.

Special thanks also go out to Beate Pieschl, for offering me opportunities and supporting me with any struggles I faced during my studies.

Lastly, I want to thank music. For this thesis I specifically want to mention Bashar Jackson, Tavoris Hollins, Karim Joel Martin and Marshall Mathers.

Christopher Oser

Graz, Austria, 23 Feb 2022

Credits

I would like to thank the following individuals and organizations for permission to use their material:

- The thesis was written using Keith Andrews' skeleton thesis [Andrews 2021b].
- Figure 3.1 is used with kind permission of Keith Andrews, Graz University of Technology.
- Figure 3.2 is used with kind permission of Keith Andrews, Graz University of Technology.
- Figure 3.3 is used with kind permission of Keith Andrews, Graz University of Technology.
- Figure 3.4 is used with kind permission of Keith Andrews, Graz University of Technology.
- Figure 3.5 is used with kind permission of Keith Andrews, Graz University of Technology.
- Figure 3.6 is used with kind permission of Keith Andrews, Graz University of Technology.
- Figure 3.7 is used with kind permission of Keith Andrews, Graz University of Technology.
- Figure 3.8 is used with kind permission of John Stasko, Georgia Institute of Technology.
- Figure 4.10 is taken from Balzer and Deussen [2005] with permission of IEEE.
- Figure 4.13 is taken from van Hees and Hage [2015] with permission of IEEE.
- Figure 4.15 is used with kind permission of Keith Andrews, Graz University of Technology.
- Figure 4.16 is used with kind permission of Keith Andrews, Graz University of Technology.
- Figure 4.17 is used with kind permission of Keith Andrews, Graz University of Technology.
- Figure A.1 is used with kind permission of Keith Andrews, Graz University of Technology.
- Figure A.3 is used with kind permission of Keith Andrews, Graz University of Technology.
- Figure A.5 is used with kind permission of Keith Andrews, Graz University of Technology.

Chapter 1

Introduction

This thesis presents the VoroTree application and the VoroLib library for interactive Voronoi treemaps. Voronoi treemaps are a space-filling information visualization technique for hierarchical information, based on recursive subdivision of Voronoi regions. VoroTree is a self-contained web application for exploring hierarchical data with Voronoi treemaps [Oser 2022c]. It is also available as a hosted demo application on the web [Oser 2022d], and can be built as an installable native package for common desktop platforms. VoroLib is a JavaScript library for integrating Voronoi treemaps into other applications [Oser 2022b]. Both projects are open source.

The first part of the thesis comprises three chapters and describes work in related areas. Chapter 2 gives an overview of modern web technologies and packages involved in the practical implementations. Chapter 3 describes the field of information visualization, with a focus on hierarchies and hierarchy visualization. Chapter 4 covers the concept of Voronoi diagrams, their construction, and their history, before explaining the development of Voronoi treemaps and their characteristics.

The second part of the thesis documents the process of developing software to implement Voronoi treemaps. The VoroTree web application is described in Chapter 5. The VoroLib library is described in Chapter 6. Chapter 7 gives inspiration for possible future work and lists some potential improvements for the existing implementations. Chapter 8 concludes the thesis with some final thoughts on the topic of interactive Voronoi treemaps.

The four appendices provide guides for various pieces of software. Appendix A is a user guide for the VoroTree application, explaining to end users how to install and use the application. Appendix B is a developer guide for the VoroTree application, explaining to developers how to install, modify, and build the VoroTree project. Appendix C is a developer guide for the VoroLib library, explaining to developers how to install, modify, and build the library. The final Appendix D describes a utility project, the Sweep Line Demonstrator, which implements Fortune's sweep line algorithm for constructing a Voronoi diagram.

Chapter 2

Web Technologies

Many technologies are required to build web-based applications, starting with the fundamental web technologies of HTML, CSS, and JavaScript, progressing through project tooling like npm, gulp, and webpack, to helpful code libraries such as D3 and PixiJS. The VoroLib library and the VoroTree application described in this thesis both make extensive use of these technologies.

2.1 Fundamental Web Technologies

Web-based software is based on three main technologies, HTML, CSS, and JavaScript. In essence, HTML defines the content of a web page or application, CSS defines its presentation, and JavaScript defines its behavior. Whilst the balance between these technologies differs between projects, all are necessary for modern web pages and applications.

2.1.1 HTML

The HyperText Markup Language (HTML) [WHATWG 2022b; Keith and Andrew 2016], is used to construct the skeleton of a web-based project. It defines the structure of a web page at a semantic level and provides the browser with the initial content it should display. The working standard of the language is maintained by the WHATWG (Web Hypertext Application Technology Working Group), a group of individuals from many large companies and organizations tasked with evolving HTML.

An HTML page is made up of *elements*, structural nodes that inherently contain information for the web browser. HTML elements are defined by *tags*, keywords delimited by angled brackets, such as `<p>` for paragraphs and `<section>` for sections. Different elements have different functions, ranging from simple structural information to the introduction of media to the page. Constructing an HTML page provides the web browser with the information necessary to display an initial unstyled version of the page. All further visual customization and functionality is then done via CSS and JavaScript. A simple example of an HTML page is shown in Listing 2.1.

HTML has gone through multiple iterations since its inception in 1992 [Berners-Lee 1992]. After four major evolutions, HTML5 was introduced in 2008 with support for multimedia content, additional structural elements, scalable vector graphics (SVG) and mathematical markup (MathML). SVG is a markup language for vector graphics [W3C 2022], which is described in more detail in Section 2.1.5. MathML is a markup language for mathematical content and formulae [Anthes 2012; W3C 2021b].

```
<!DOCTYPE html>
<html>
<body>

<h1>A Heading for a Webpage</h1>
<p>Lorem ipsum dolor sit amet...</p>

</body>
</html>
```

Listing 2.1: A simple HTML page with a heading and a paragraph of text.

```
body {
  background-color: grey;
}

h1 {
  color: white;
  text-align: center;
}

p {
  font-family: verdana;
  font-size: 20px;
}
```

Listing 2.2: A simple CSS style sheet, defining styles for `body`, `h1` and `p` HTML Elements.

2.1.2 CSS

Cascading Style Sheets (CSS) [W3C 2021a; Cederholm 2015] is a style sheet language which defines the presentation of markup language documents. It determines the visual appearance of a web page or application, from colors and sizes to fonts and positions. The current CSS standard is maintained by the World Wide Web Consortium (W3C).

Style sheets written in CSS contain style rules which are applied to HTML elements. Properties can be set individually for each HTML element, and in particular contexts (such as when that element is in focus), enabling numerous customization options. Any properties not set assume the default value defined by the web browser. Properties defined later in a style sheet override proceeding ones. A simple example of a CSS style sheet is shown in Listing 2.2.

CSS was first introduced by Håkon Lie in 1994 [Lie 1994]. At the time, many style sheet languages were proposed to be used with HTML. CSS managed to prevail and with the support of W3C, turned into the standard on the web [Bloom 2016]. Until CSS 2.1, the CSS standard was a single document, but for CSS 3 it was split across different modules [BitDegree 2020].

```
document.getElementById('button').addEventListener('click', () => {
  window.alert('Hello World!');
})

document.getElementById('heading').addEventListener('hover', () => {
  document.getElementById('heading').style.background-color = "#ccffee";
})
```

Listing 2.3: JavaScript code defining behavior for a button and a heading via the DOM API.

2.1.3 JavaScript

The third pillar of web development is JavaScript, a programming language used to define the functionality and behavior of web pages and applications [ECMA 2021; Marquis 2016]. JavaScript is a weakly typed language for which many third party libraries are available. All the usual programming functionality is provided, such as variables, loops, conditions, and functions. The current standard is defined by Ecma International (formerly ECMA), a non-profit organization which develops global standards for many languages and formats.

JavaScript interacts with elements on web pages via the Document Object Model (DOM) [W3C 2004; WHATWG 2022a; Lindley 2013]. The DOM represents the content and structure of a web page as a hierarchy of nodes, one for each HTML element. The DOM's API can be used from JavaScript to modify or add interactivity to HTML elements. Listing 2.3 shows two such DOM operations. In both cases, an element is first accessed from the DOM by specifying its identifier, and then an operation is performed upon it.

Development of JavaScript was initiated at Netscape during the development of the Netscape Navigator browser under the name Mocha and later LiveScript [Peyrott 2017]. The first version of JavaScript was released in December 1995 [Netscape 1995]. Over the years, other companies and organizations also adopted the language, and it became a standard for web software.

2.1.4 TypeScript

As mentioned previously, JavaScript is a weakly typed language, making it prone to errors and attacks. To counter these shortcomings, Microsoft developed TypeScript, a syntactical superset of JavaScript [Microsoft 2022; Baumgartner 2021]. Code in TypeScript can be compiled (transpiled) to JavaScript, enabling the result of any TypeScript project to run in a browser as JavaScript.

TypeScript was first published in 2012 by Microsoft. It offers many additional features on top of JavaScript, like classes, interfaces, and a rich type system. TypeScript makes working with JavaScript cleaner and better structured. Potential attacks and security risks enabled by weak typing in JavaScript are countered by using TypeScript [Bierman et al. 2014].

2.1.5 SVG

Scalable Vector Graphics (SVG) [W3C 2022; Coyier 2016], is a standardized XML-based format for vector graphics. Similar to HTML, it is human-readable, with an outermost `<svg>` element containing graphical elements such as `<rect>`, `<circle>`, and `<text>`. Being a vector graphics format, SVG has the added advantage of being fully scalable, without loss of quality. Apart from writing SVG files by hand, it is also possible to draw them using vector-based drawing editors such as Adobe Illustrator or Inkscape,

or to generate them by scripting [Adobe 2022; Inkscape's Contributors 2022]. Since HTML5 in 2008, web browsers support the mixing of SVG elements alongside HTML elements inside a web page.

2.2 Dynamically Drawing Web Graphics

There are three ways to programmatically draw arbitrary graphics inside the web browser: Canvas2D, WebGL, and SVG injection. Canvas2D and SVG injection support two-dimensional drawing, WebGL supports three-dimensional drawing and takes advantage of graphics hardware (GPU) in the display device.

2.2.1 Canvas2D

The HTML5 `<canvas>` element [W3Schools 2022] defines a rectangular region in a web document which can be drawn into programmatically from JavaScript. Low-level APIs are associated with the rendering context of a particular canvas. The Canvas API [MDN 2022a], referred to here as Canvas2D, is associated with the rendering context `2d` and is used to draw 2d graphics. It supports basic drawing actions such as lines, shapes, filling, and rotation. Furthermore, it offers options for gradient drawing, text and importing images. It is possible to animate with canvas by redrawing the canvas with timeout functions.

2.2.2 WebGL

The Web Graphics Library (WebGL) [Khronos 2022a; MDN 2022b] is associated with the `<canvas>` rendering context `webgl` or `webgl2` and is used to draw both 2d and 3d graphics. The WebGL API is based on OpenGL ES [Khronos 2022b] and is designed to take advantage of any graphics hardware (GPU) which may be present. Drawing with WebGL is typically much faster than drawing with either Canvas2D or SVG injection.

2.2.3 SVG Injection

As mentioned previously, web browsers support the mixing of SVG elements alongside HTML elements inside a web page. Using JavaScript, it is possible to construct and insert SVG elements into the HTML DOM dynamically. This is referred to here as SVG injection, and is a further way of programmatically drawing arbitrary graphics within a web page or application. The D3 library uses this technique to render its graphics.

2.3 Responsive Web Design

Responsive web design is a web design technique whereby a web page or application adapts to the characteristics of the display device [Marcotte 2010; Marcotte 2014]. The same web page should work equally well across many shapes and sizes of devices, taking advantage of their unique characteristics, but does *not* need to look exactly the same on every device [Keith 2016, Chapter 6; Keith 2015, 36:00]. Since the majority of web users now use mobile devices [Enge 2021], responsive web design has become the standard for web pages and applications. In essence, web design *is* now responsive web design. Users consume content on desktop PCs, laptops, tablets, and smartphones. Therefore, web content must be adapted to be usable on all platforms.

The core idea behind responsive web design is for a web page or application to automatically adapt to the characteristics of the display device. Predominantly, a page or application will adjust to the available screen space, and in particular screen width, since vertical scrolling is considered more acceptable than horizontal scrolling. Components on a page may be resized, repositioned, or modified depending on the available space. However, responsive web design is not only about display space. It is important to

support whatever interactivity is supported by the display device. Mouse, keyboard, and touch interaction should all be supported on devices which provide them.

VoroTree is a responsive web application. It adapts its visual representation in respective to window size. All font sizes are also adjusted. If the window size changes, VoroTree will reconstruct the visualization, so it fills the entire available space. VoroTree also adjusts the sharpness of the visualization according to device resolution and interactive zooming of the screen by the user.

2.4 Project Tooling

VoroLib and VoroTree both require specific web technologies and tooling to build and maintain. This includes basics such as compilation, bundling, and project administration.

2.4.1 Node.js

Node.js is an open source JavaScript runtime environment [OpenJS 2022b]. It provides asynchronous input/output and optimizes the scalability and throughput of web applications. Node.js was originally intended to be used for server-side scripting, enabling the use of JavaScript on the server as well as in the client (browser). In the meantime, it has been ported to many other environments and devices.

2.4.2 npm

npm, the Node Package Manager [npm 2022], is a package manager for JavaScript projects built with Node.js. It offers quick maintainable access to thousands of major libraries and packages. Packages are installed from an online database and are organized automatically in the local storage. This saves developers the work of structuring and adapting multiple different libraries to work with one another.

2.4.3 gulp

gulp is a task runner for npm projects [Bublitz and Schoffstall 2022]. It can serialize tasks and aggregate long procedures into single commands, making it easier for developers to maintain a project. Building or serving a project is a simple gulp command once the procedure has been defined. gulp also supports actions like deployment, testing, or copying files.

2.4.4 webpack

webpack is a module bundler for JavaScript [Koppers et al. 2022]. Its main purpose is the bundling of multiple sources to a single software package. Through its configuration file and extra plugins, the process can be highly customized to the needs of the project. It also provides a development server, enabling developers to run the project locally under different settings. webpack is an open-source project built and maintained by numerous developers.

2.5 External Libraries

VoroLib and VoroTree depend on a number of external libraries for some of their functionality. In particular, these dependencies are used for drawing, calculation of the Voronoi treemap structure, and exporting the visualization.

2.5.1 PixiJS

PixiJS is a JavaScript library which provides support for drawing via code [Groves 2022]. It uses either WebGL or Canvas2D to render its graphics. Typically, WebGL is preferred for performance reasons, with a fallback to Canvas2D if WebGL is not supported on the device. PixiJS offers many classes and structures for efficient drawing and animation. It also includes text rendering and accessibility features like support for screen readers and touch devices.

2.5.2 D3

D3 is a JavaScript library for data visualization [Bostock et al. 2011; Bostock et al. 2022]. It uses SVG injection and CSS to draw graphics and charts by binding graphical marks to data points. D3 was initially packaged as a single, monolithic library, but since Version 4 it is split into a number of reusable independent modules. Developers can choose to include only the modules that are relevant to them. The three most important D3 modules used in VoroLib and VoroTree are d3-hierarchy, d3-delaunay, and d3-voronoi-treemap.

2.5.2.1 d3-hierarchy

d3-hierarchy is a D3 module which enables efficient handling of large hierarchies [Bostock 2022]. It includes data import from JSON or CSV files to D3's hierarchical data structure. Once imported, the data can be analyzed and edited through numerous functions. Additionally, there are three methods of visualizing the data in various diagrams built into d3-hierarchy.

2.5.2.2 d3-delaunay

d3-delaunay is a D3 module which constructs a Voronoi diagram from a set of points [Observable 2022]. It is based on Delaunator by Vladimir Agafonkin [Mapbox 2022], which implements Delaunay triangulation. Delaunay triangulations and Voronoi diagrams are duals, given one, the other can easily be computed. Being a D3 module, it works with the D3 data structure created by d3-hierarchy.

2.5.2.3 d3-voronoi-treemap

d3-voronoi-treemap is a D3 module which produces a Voronoi treemap [LeBeau 2022]. It requires a convex polygon and nested weighted data. With this input, it builds the polygons representing the Voronoi treemap of the data within the convex polygon. It works well with other d3 modules, including d3-hierarchy and its data structure.

To construct the Voronoi treemap, the module uses the "Lift to 3-Space" algorithm described in Section 4.1.4.4. For each layer of the treemap, the weighted Voronoi diagram is calculated. Then the positions and weights of all sites are adapted to the result and a new diagram is constructed. This is done until the error in respect to the input data reaches a predefined small value. The algorithm is based on the procedure described by Nocaj and Brandes [2012].

2.5.3 Electron

The VoroTree web application is built using Electron, an open-source framework for creating desktop applications from web applications [OpenJS 2022a]. It uses the Chromium engine to construct binaries from the web project source code, in essence bundling a browser with the web application into one executable package. Electron can build applications for Windows, macOS, and Linux, making it easy to distribute a web application for offline use on any desktop device. Electron is developed and maintained by the OpenJS Foundation.

Chapter 3

Information Visualization

Information Visualization (InfoVis) is part of the computer science field of visualization. As described by Andrews [2021a], InfoVis techniques are used to visualize abstract information spaces to aid the understanding of the underlying data by human viewers. Interactivity is provided to support the exploration of the visualization.

Next to InfoVis, the two other major subfields of visualization are Geographical Visualization (GeoVis) and Scientific Visualization (SciVis). GeoVis visualizations are usually based on 2d or 3d maps from spatial coordinates inherent in the data. Visual information or glyphs are layered on top of the already present map-based geometry. SciVis visualizations are based on concrete objects, either real or simulated. These can be medical scans, air flows around a wing, or processes in a biological cell, for example. SciVis focuses on depicting concrete objects while adding visual elements to make the science more comprehensible.

In InfoVis, the visual representation needs to be carefully chosen and designed, whereas in both GeoVis and SciVis the visual representation is already inherently given in some form [Andrews 2021a]. The combination (union) of GeoVis and InfoVis is often collectively referred to as Data Visualization (DataVis).

3.1 InfoVis

InfoVis deals with *abstract* information structures such as hierarchies, networks, multidimensional tabular data, and feature spaces derived from collections of objects. As these information structures are not easily visualized, the goal of InfoVis is to represent the information in a way that humans can extract insights intuitively from a visualization. This allows for large amounts of data to be comprehended easily and swiftly.

Visualizations facilitate not only the understanding of the input data, but can also highlight unexpected properties about the data. Certain patterns or anomalies that are easy to distinguish through visualizations can lead to new insights about the underlying data. Furthermore, visualizing information can often lead to problems within the information to become apparent. Errors introduced when gathering data can be discovered and therefore quality of the information improved [Ware 2019].

InfoVis works almost exclusively on visual perception. The quirks of human vision can be used to improve visualizations. Color and depth perception can be used to carry information in a visualization. Some visual attributes are processed preattentively (in parallel) by humans and can therefore be used to convey information to a user quickly. Examples for this include color hue, color intensity, line length, line width, two-dimensional spatial positions, orientation, size, and shape. These attributes are understood swiftly by the human brain, which makes them very useful for visualizations [Andrews 2021a].

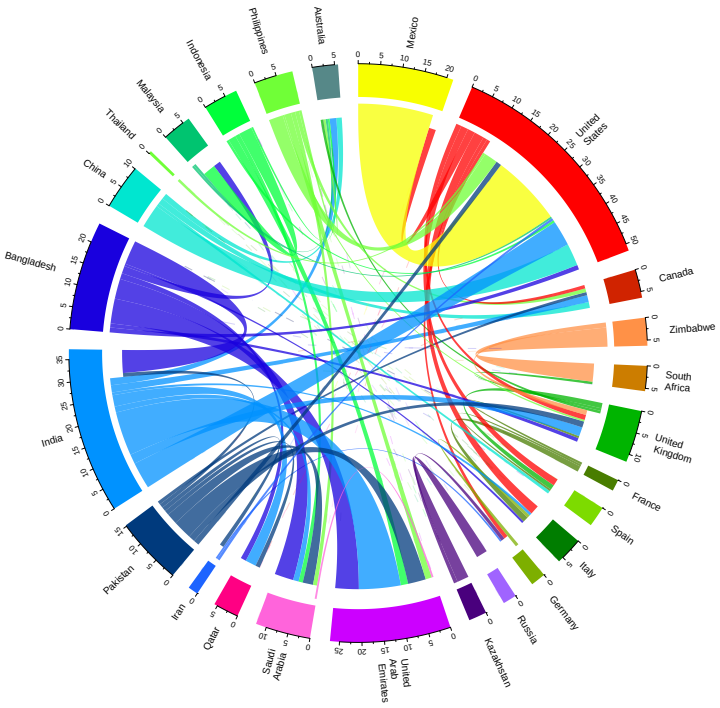

Figure 3.1: A chord diagram is an example of predefined position visualization for network data. Nodes are placed around the perimeter of a circle. Links are drawn as chords between nodes. The data here shows migration between countries. The visualization was constructed using an earlier version of R code by Abel [2021]. [Used with kind permission of Keith Andrews.]

An important part of InfoVis is visual encoding. Characteristics of the data can be mapped to corresponding Visual attributes. Line length and two-dimensional spatial position are perceived very accurately and are well suited to encoding quantitative variables. Other attributes, like color intensity or size, can not be perceived as accurately. It is easier for humans to compare and interpret the relative length of two bars in a bar chart than the relative sizes of two circles. Color hue and shape are the two visual attributes most suitable for encoding categorical data [Andrews 2021a].

The main data structures used for information visualization include linear structures, hierarchies, networks and graphs, multidimensional metadata, and feature spaces. Examples of linear structures are alphabetical or chronological lists, lines of program source code, or ranked search results. Hierarchies are tree structures such as file systems, product taxonomies, or library classification schemes. These are described in more detail in upcoming Sections 3.2 and 3.3.

Networks and graphs comprise a collection of nodes (vertices) and a collection of links (edges) connecting the nodes. Both nodes and links can also have attributes. They can be visualized in many ways. In adjacency matrices, links from one node to another are entered into cells of an $n \times n$ matrix, where n is the number of nodes. With predefined position visualizations, the nodes are laid out in predetermined positions, for example according to a linear, circular, grid, or geographical layout. An example of circular layout, sometimes called a chord diagram, can be seen in Figure 3.1. Layered graph drawing can be used to visualize directed graphs. Nodes are layered according to direction and edges are added between layers. To avoid clutter, nodes are positioned within a layer to reduce edge crossings as far as possible. A further technique to represent networks and graphs is force-based layouts. Nodes are positioned on a plane according to forces of attraction and repulsion working between them, whereby the forces of attraction are typically derived from the strength of edges between the nodes.

Multidimensional metadata refers to classic tabular or spreadsheet-style data. By convention, rows represent records (data items) and columns represent dimensions (variables) in the data. For example,

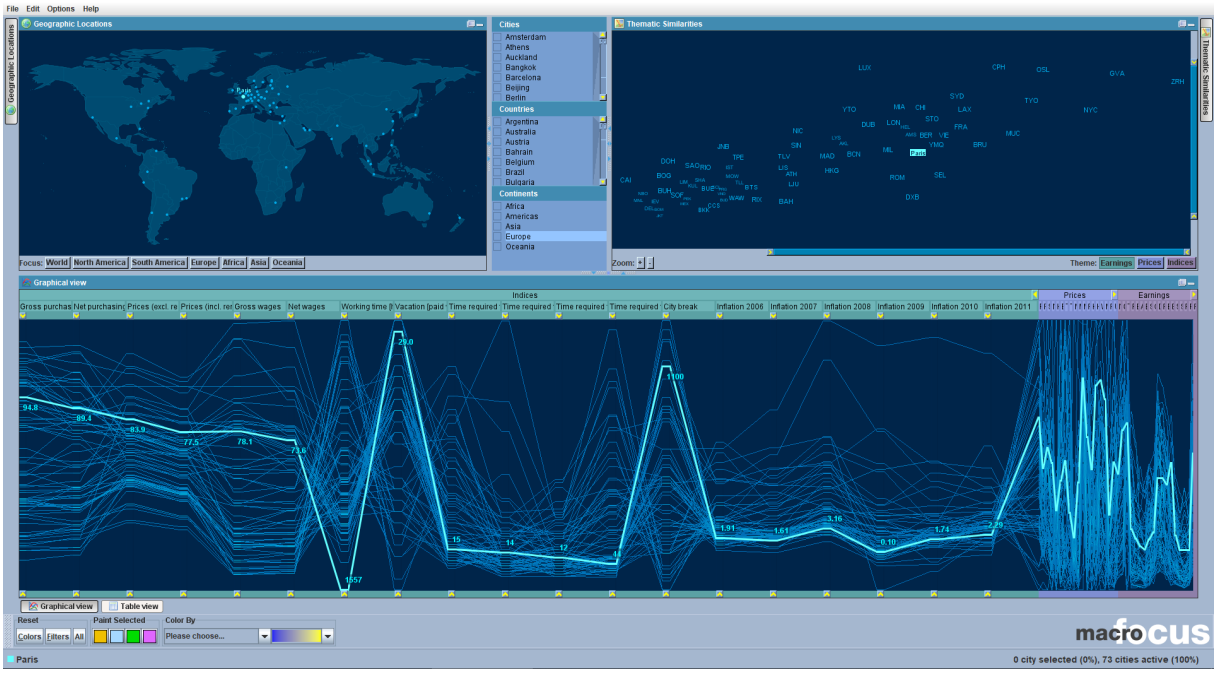

Figure 3.2: InfoScope allows exploration of multidimensional data about 73 cities. Dimensions include data such as net wages and working time (hours per year). In addition to the geographic map at the top left, it provides a similarity map in the top right, and a parallel coordinates view across the bottom. [Used with kind permission of Keith Andrews.]

a dataset about cars might have dimensions such as name, manufacturer, weight, and fuel consumption. Common ways of visualizing multidimensional data include scatter plots, parallel coordinates, and similarity maps. The InfoScope system [Macrofocus 2008], shown in Figure 3.2, visualizes multidimensional data about 73 cities, with dimensions like net wages and working time in hours per year.

Collections of homogeneous items, such as PDF text documents or photographs, can be analyzed, and characteristic features assigned to each item in the collection, forming a *feature space*. For example, for a collection of text documents, features might include counts of how often every word occurs in every document. These feature spaces typically comprise many tens of thousands of features. Visualizations of feature spaces often involve projection of distances in the high-dimensional feature space down to two dimensions, such that similar items are placed close to one another in the visualization. Figure 3.3 shows the VisIslands system which visualizes search result sets using force-directed placement [Andrews et al. 2001].

For further reading, Ward et al. [2015] and Munzner [2014] provide in-depth coverage of the field and techniques of InfoVis.

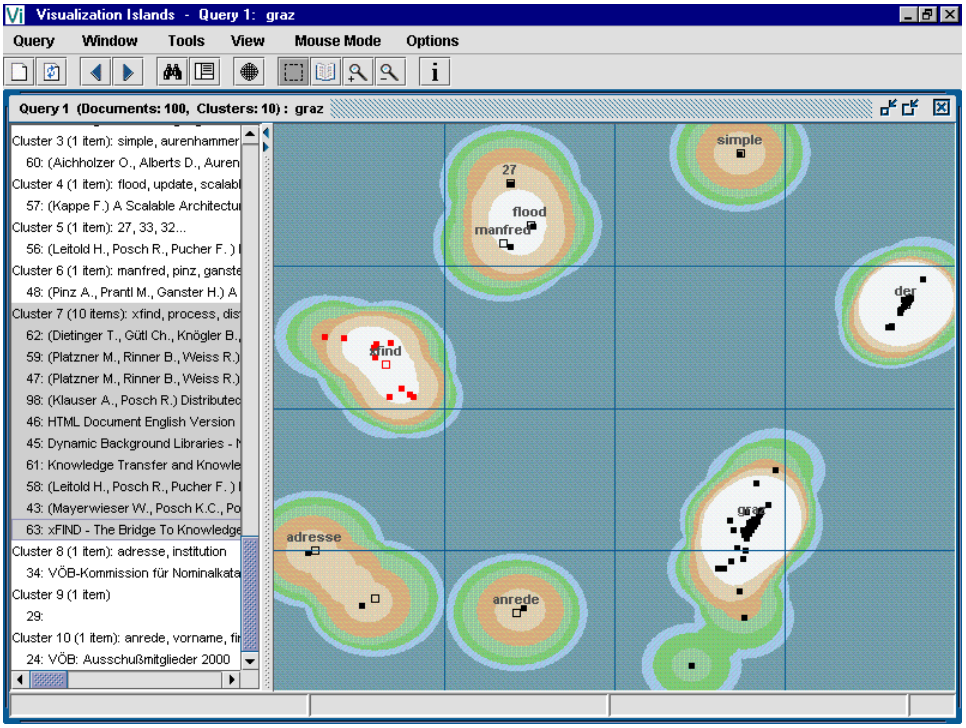

Figure 3.3: VisIslands forms visual clusters of search result sets using force-directed placement [Andrews et al. 2001]. [Used with kind permission of Keith Andrews.]

3.2 Trees and Tree Structures

Hierarchies are extremely common in modern life. They can be found in family trees, product taxonomies, file systems, and various classifications. Tree structures are used to represent hierarchies. They adhere to a specific terminology and can be stored in different file formats.

In information technology, trees are generally modeled as nodes and edges. Each node represents an entry within the data. Nodes are connected via edges, indicating relations to other nodes. These relations can either be to a superior (parent) node or an inferior (child) node, which define the hierarchical nature of the data. Generally, most nodes in a dataset have both parent and child nodes, these nodes are called *inner* nodes. Nodes with no child nodes are called *leaf* nodes. Every tree structure has at least one node which has no parent node, the highest node within the hierarchy, this node is called the *root* node.

Regular tree structures have a single unique root node. However, special trees like multitrees have multiple root nodes [Furnas and Zacks 1994]. Furthermore, hierarchies and tree structures are acyclic, meaning they contain no cycles. A structure contains a cycle if there is a path from and to the same node, in which the only node that is used twice is as the start and end node. By definition, this cannot happen in tree structures.

Tree structures can be stored in various file formats. Most are based on text file formats like XML, JSON, or CSV. TreeML is a markup language based on XML [Fekete and Plaisant 2003; Alencar 2010] designed to represent tree structures. Simple Knowledge Organization System (SKOS) is an XML format for representation of various graph structures like dictionaries, taxonomies, and classifications [W3C 2009], but can also be used to represent tree structures. JSON files can also be used to represent tree structures. D3 uses a JSON format to represent the hierarchy structure used in D3 algorithms and libraries [Bostock 2022]. jsTree, a jquery plugin that enables visualization of interactive trees on web pages, also implements a specific format for JSON files [Bozhanov 2022]. Trees can also be stored in CSV files. Typically, each node takes up one row. In one variant, each column in the data represents one level of the tree, and each row has exactly one entry specifying where in the tree the node lies. In another variant, an

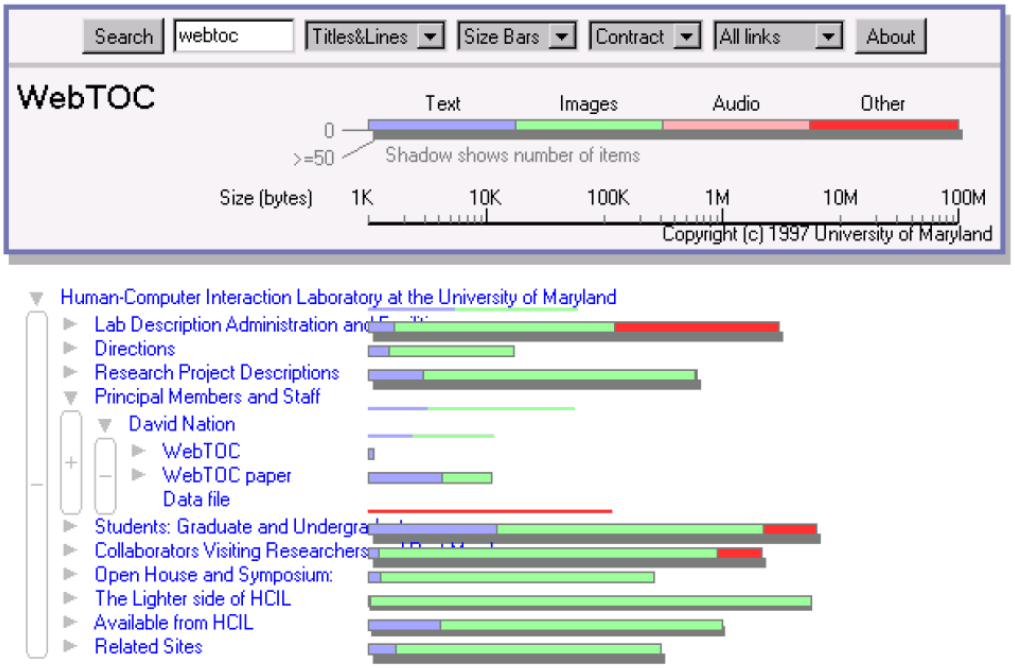

Figure 3.4: A WebTOC table of contents for the University of Maryland's HCI Lab web page. [Used with kind permission of Keith Andrews.]

ID column assigns an ID to each node and a parent column specifies the ID of the parent.

3.3 Hierarchy Visualization

Hierarchy visualizations strive to visually convey the parent-child tree structure in an intuitive way. Some hierarchy visualizations also visualize attributes associated with nodes of the tree. Hierarchy visualizations which explicitly draw a connection between parent and child nodes are known as *node-link* (or *explicit*) hierarchy visualizations. Hierarchy visualizations which implicitly indicate the parent-child relationship through placement are known as *space-filling* (or *implicit*) hierarchy visualizations [Andrews 2021a; Schulz 2011].

Hierarchy visualizations can also often be characterized as either *layered*, *radial*, or *inclusive*. Layered visualizations place nodes in specific layers horizontally or vertically. Radial visualizations place nodes on concentric circles. Inclusive visualizations use recursive containment to place nodes.

The outline approach to hierarchy visualization uses expandable and collapsible lists like a table of contents. Familiar examples include file system browsers like Windows Explorer or the Java JTree component. Another example is WebTOC [Nation et al. 1997] for browsing hierarchies on a web page, shown in Figure 3.4.

The Walker tree browser, shown in Figure 3.5, is an example of a layered node-link tree layout. Nodes are placed in vertical layers according to their depth in the tree. The placement is calculated bottom-up according to Walker's algorithm [Walker 1990; Buchheim et al. 2002]. Another example of a layered node-link tree browser is the Harmony Information Landscape [Andrews 1996].

A similar approach to visualizing hierarchies is the radial node-link tree browser, shown in Figure 3.6. Nodes are placed on radial layers (concentric circles) extending from the root node in the center. The tree expands in all directions. Another example of a radial node-link hierarchy browser are hyperbolic browsers such as the D3-Hypertree [Glatzhofer 2021].

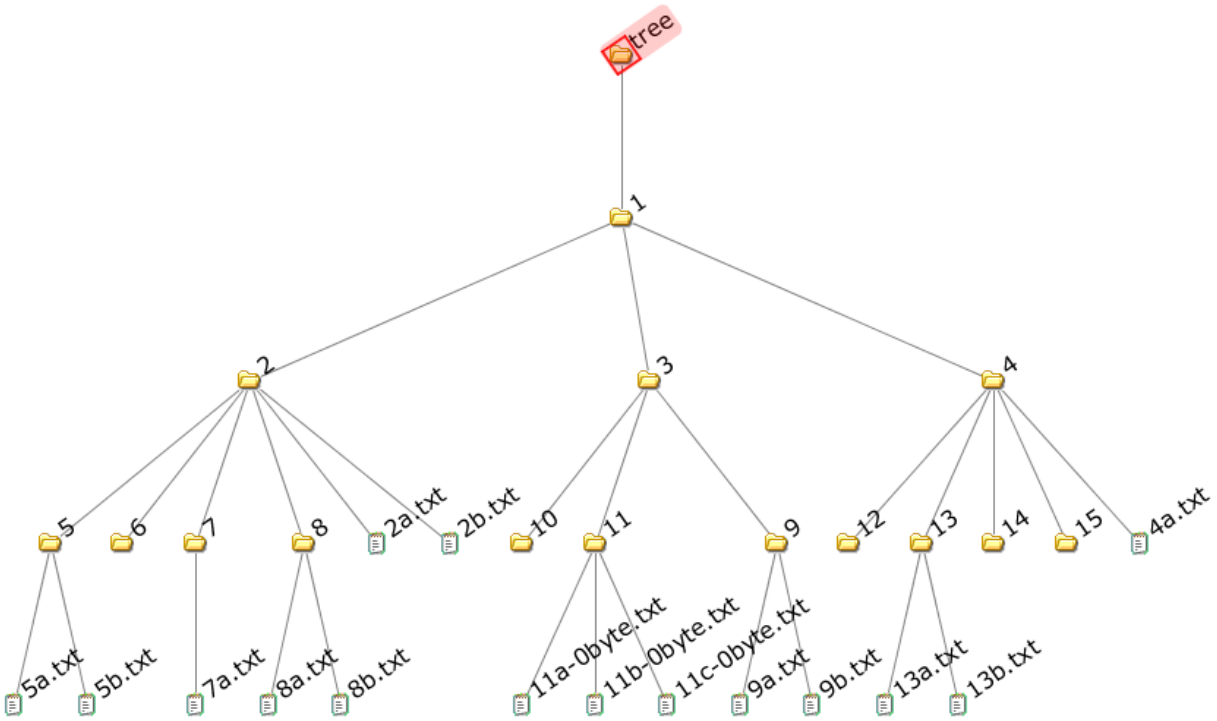

Figure 3.5: The Walker tree browser from HVS [Andrews et al. 2007] is an example of a layered node-link tree layout. [Used with kind permission of Keith Andrews.]

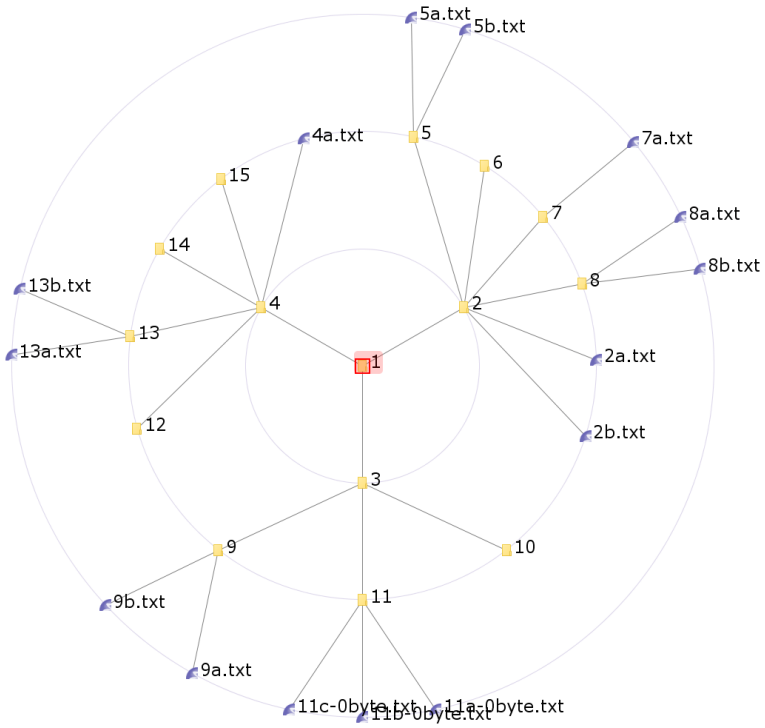

Figure 3.6: The radial tree browser from HVS [Andrews et al. 2007] is an example of a radial node-link tree layout. [Used with kind permission of Keith Andrews.]

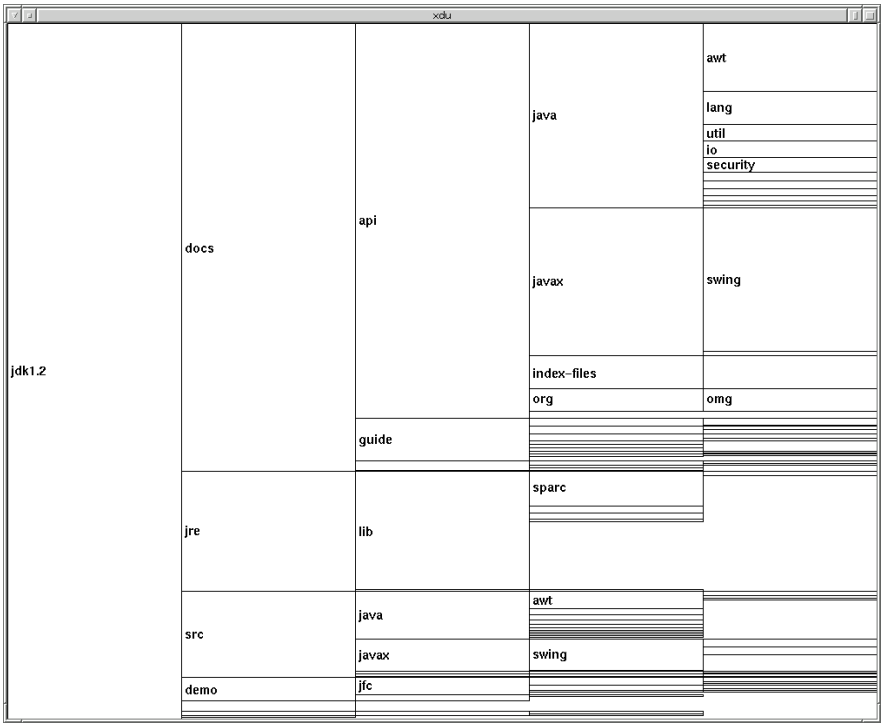

Figure 3.7: A hierarchy visualization from XDU. It is an example for a layered node-link tree. [Used with kind permission of Keith Andrews.]

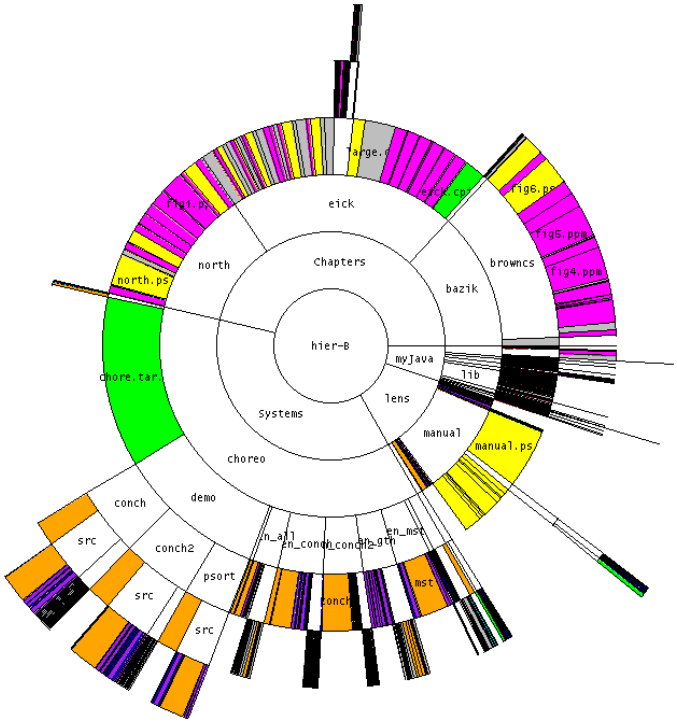

Figure 3.8: A hierarchy visualization from SunBurst. It is an example for a radial space-filling tree. [Used with kind permission of John Stasko.]

Figure 3.9: Hierarchically structured data from the NBA basketball league shown with Treemap 4.1.1. At the top level, the two divisions (Central and Atlantic) are displayed on the left and right. The next level shows the teams in each division. Finally, the colored rectangles show individual players. Here, the size of the respective rectangle corresponds to the minutes per game and the color to the points per game of each player. [Screenshot taken by the author using Treemap 4.1.1 [Shneiderman 2004]]

XDU [Dykstra 1991], shown in Figure 3.7, is an example of a layered space-filling tree browser. Nodes are represented by rectangles placed in layers extending horizontally from left to right. The root node is at the far left. SunBurst [Stasko and Zhang 2000], shown in Figure 3.8, is an example of a radial space-filling tree browser. Here, circular wedges radiate out from the root node at the center.

Treemaps are an example of inclusive space-filling hierarchy visualization. Treemaps were first introduced by Shneiderman [1992] in the 1990s. His team at the University of Maryland then also developed the Treemap application, which was improved and maintained until 2004 [Shneiderman 2004]. Nodes in the hierarchy are represented with nested rectangles. The size and color of the rectangles is determined by attributes of the corresponding tree nodes. The original Treemap application is shown in Figure 3.9. Double-clicking a node drills down into that subtree, right-clicking moves up one level.

For further examples of hierarchy visualizations, Schulz et al. [2011] provides a comprehensive survey. The complementary Visual Bibliography offers a comprehensive online reference [Schulz 2011]. Each technique is represented by a small image. Techniques can be sorted and filtered according to their characteristics.

Finally, Voronoi treemaps are treemaps which use nested Voronoi polygons instead of nested rectangles. This technique, first introduced by Andrews et al. [2002], forms the basis of the work presented later in this thesis and is described in detail in Chapter 4.

Chapter 4

Voronoi Treemaps

A Voronoi treemap is a treemap constructed using Voronoi polygons. Whereas classic treemaps allocate space to child nodes by recursively subdividing rectangular regions, Voronoi treemaps recursively subdivide Voronoi regions. Similar to treemaps, the size of the polygons can be mapped to correspond to a metric of the hierarchical data, such as size in bytes, or number of children. The technique was first described in 2002 by Andrews et al. [2002].

4.1 Voronoi Diagrams

Voronoi diagrams are used to split a plane into regions surrounding a given set of points, called *sites*. In simple terms, a Voronoi diagram indicates which points on the plane are closest to which of the sites. Hence, the plane is divided into regions of influence of the chosen sites. A simple form of the Voronoi diagram for three sites can be seen in Figure 4.1.

4.1.1 Applications

Voronoi diagrams have a wide variety of uses. While some find the diagram itself to be pleasing to the eye, computer scientists use them for many functional applications, including basic distance problems, the classical postman problem, classification, and various clustering algorithms [Aurenhammer et al. 2013, pages 3–5]. Furthermore, it is used in computer graphics to generate textures.

Voronoi diagrams are also used in other fields, often under other names. The Thiessen polygons in geometry or meteorology, Wigner-Seitz zones in chemistry and physics, and domains of action in crystallography are examples of Voronoi diagrams in other fields [Aurenhammer et al. 2013, pages 1–2]. In biology, Voronoi diagrams are used to model cells or bone micro architecture [Bock et al. 2010; Li et al. 2012]. Some uses of Voronoi diagrams, such as nearest neighbor query, span multiple areas of science [Okabe et al. 2000, pages 10–11].

4.1.2 History

As far as is known, the first mentions of Voronoi diagrams were in the 17^{th} century when René Descartes claimed the solar system consisted of vortices. While he did not explicitly define the regions, the process behind his idea is very close to what is nowadays understood to be Voronoi diagrams [Aurenhammer et al. 2013, page 1; Okabe et al. 2000, page 6].

The idea was formalized by the mathematicians Carl Friedrich Gauß in 1840, Gustav Lejeune Dirichlet in 1850, and Georgi Feodosjewitsch Voronoi in 1908. These formalizations led to the structure that is now considered to be a Voronoi diagram [Aurenhammer et al. 2013, page 3; Okabe et al. 2000, page 6].

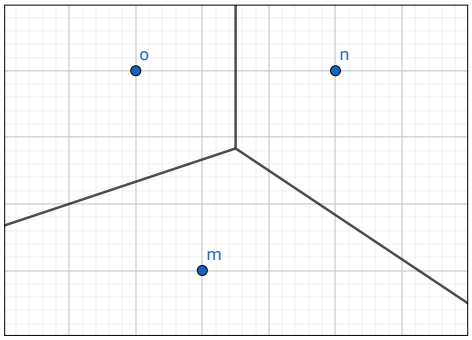

Figure 4.1: The Voronoi diagram for the three points (sites) m, n, and o. [Screenshot taken by the author using GeoGebra [GeoGebra 2021].]

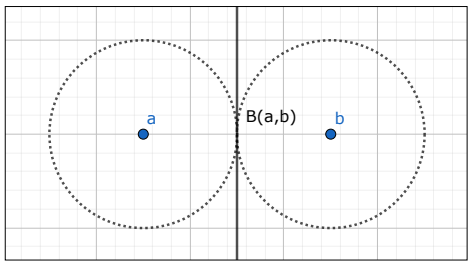

Figure 4.2: The bisector describes the locus of all points at equal distance to two points. [Screenshot taken by the author using GeoGebra [GeoGebra 2021].]

4.1.3 Basic Theory

To construct a Voronoi diagram, a set of at least three points, or *sites*, is required. The term sites is used to better differentiate them from generic points on the plane. A Voronoi diagram of just two sites is simply the bisector, which is the locus of all points at equal distance to these two sites, as can be seen in Figure 4.2.

In its basic form, the Voronoi diagram is constructed on a Euclidean plane which means all distances are given as:

$$d(a,b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2} \tag{4.1}$$

for two sites, a and b. With this, the half plane between two sites can be defined as:

$$H(a,b) = \{x | d(a,x) \leq d(b,x)\}, \tag{4.2}$$

where the bisector of a and b acts as a boundary for this half plane. A Voronoi region is then formed by taking the intersection of all $n-1$ half planes in the set of n sites. Such a region can be seen in Figure 4.3 in the form of the triangle around point g. Such a Voronoi region contains all points closest to its site. In Figure 4.3, this means that all points on the plane within the triangle are closer to g than to any of the other sites (e, f, h).

Two more definitions are necessary to understand a Voronoi diagram. A Voronoi *vertex* is an endpoint within the Voronoi diagram. The edges connecting the Voronoi vertices are known as Voronoi *edges*. Both can be seen in Figure 4.3.

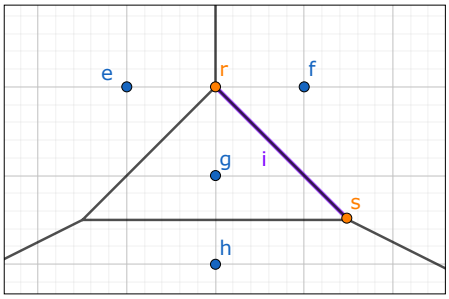

Figure 4.3: The triangle around point g indicates its Voronoi region. The line segment i is a single Voronoi edge. The two points which define the Voronoi edge, r and s, are examples of Voronoi vertices. [Screenshot taken by the author using GeoGebra [GeoGebra 2021].]

4.1.4 Algorithms

There are four types of construction algorithm for Voronoi diagrams. While they do not vary greatly in performance, they are based on very different approaches to building the diagram.

4.1.4.1 Incremental

The most intuitive way of constructing a Voronoi diagram is the incremental method. To construct the diagram, the sites from the set are added to the plane one by one. Every time a new site is added, the diagram is adapted to the new site. This means finding all now faulty edges, deleting them, and adding the new correct edges to accommodate the new site. Once all sites have been added, the final Voronoi diagram has been constructed.

This method has the benefit of being on-line, which means the set of sites does not have to be complete when starting the algorithm as they are read in one by one. Additional sites can be added later on. The downside of this algorithm is that it has a worst case runtime of $O(n^2)$. However, the expected runtime for a randomized algorithm is $O(n \log n)$ [Aurenhammer et al. 2013, pages 18–23].

4.1.4.2 Divide & Conquer

The Divide & Conquer algorithm for Voronoi diagrams requires the entire set of sites to be known right from the start. This is because, as the name suggests, it splits the set into a left half and a right half, according to the placement of the sites on the plane. Once the sites are split up, this is repeated recursively, until the only sets remaining have two or three sites each. For these small sets, simple Voronoi diagrams are easily constructed. Once the diagrams are constructed for all sets, the smaller diagrams are merged back together. After the final two diagrams have been merged, this represents the finished Voronoi diagram for the set of sites.

This algorithm has an upper boundary of $O(n \log n)$ for all cases and therefore performs better than the incremental algorithm in general [Aurenhammer et al. 2013, pages 24–27].

4.1.4.3 Sweep Line (Fortune's Algorithm)

First introduced by Steven Fortune [Fortune 1986], the sweep line algorithm uses a line which moves across one axis of the plane generating events in a queue as it moves. The closer the event is to the sweep line, the earlier it will be handled. There are two types of events: site events and circle events. Site events occur at each site in the set. A circle event is created during a site event and describes a point at which a parabola vanishes from the beach line formed by the site events. During a circle event, a new Voronoi vertex can appear.

As soon as the sweep line has crossed the entire plane, the Voronoi diagram is constructed. This algorithm also has an upper boundary of $O(n \log n)$, which means it is comparable in performance to the

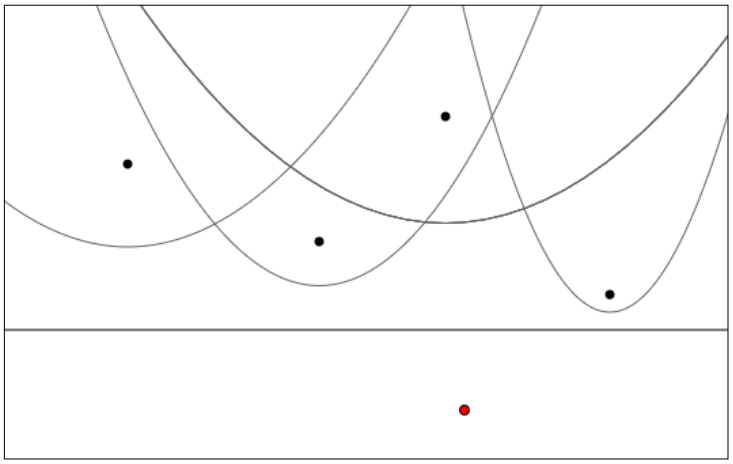

Figure 4.4: The black horizontal line is the sweep line moving downwards. All the black sites and their events have already been handled. The red dot indicates a circle event, through which most likely a new Voronoi vertex is found. [Screenshot taken by the author using a self-constructed demonstrator.]

Divide & Conquer algorithm [Aurenhammer et al. 2013, pages 28–30]. A screenshot of the algorithm in action is shown in Figure 4.4

4.1.4.4 Lift to 3-Space

This construction algorithm works by lifting the sites into the 3rd dimension. This is usually done using a simple projection, for example to a paraboloid like:

$$z = x^2 + y^2. \tag{4.3}$$

Such a projection can be seen in Figure 4.5. Each site is lifted from the 2-dimensional plane to the surface of the paraboloid. Once the projection is done, the convex hull of all sites on the paraboloid is calculated. There already exist many algorithms for computing the convex hull of the sites. These algorithms have the added advantage that they have been used for some time and are optimized and easily accessible.

By translating the resulting convex hull back to two dimensions, what remains is the Voronoi Diagram from the specified set of sites. As with the previous algorithms, the upper boundary on runtime is $O(n \log n)$ [Aurenhammer et al. 2013, pages 31–34].

4.1.5 Weighted Voronoi Diagrams

In data visualization, it is often desirable to assign higher weight to certain properties. This is also possible for Voronoi diagrams. Weighted Voronoi diagrams can be constructed in a number of ways. The idea remains the same: the higher the weight of a certain site, the larger the resulting Voronoi region should become. This can be seen in Figure 4.6. The standard Voronoi diagram of the four sites is represented by the dashed line. The solid line is a weighted Voronoi diagram, where higher weight has been assigned to sites **a** and **b**, increasing the size of their regions.

One approach to computing weighted Voronoi diagrams is by using either *additively weighted* or *multiplicatively weighted* diagrams. This means the usual distance calculation, as in 4.1, is substituted by either:

$$add_d(a, b) = d(a, b) - weight(a) \tag{4.4}$$

for an additively weighted diagram or by:

$$mult_d(a, b) = d(a, b)/weight(a) \tag{4.5}$$

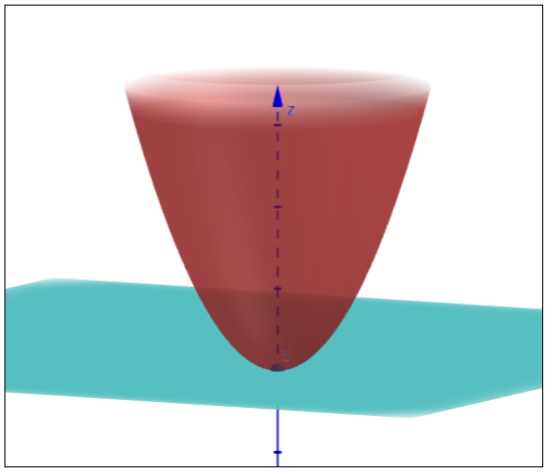

Figure 4.5: Sites on a plane are lifted to 3-dimensional space by being projected onto the surface of a paraboloid, in this case $z = x^2 + y^2$. [Screenshot taken by the author using GeoGebra [GeoGebra 2021].]

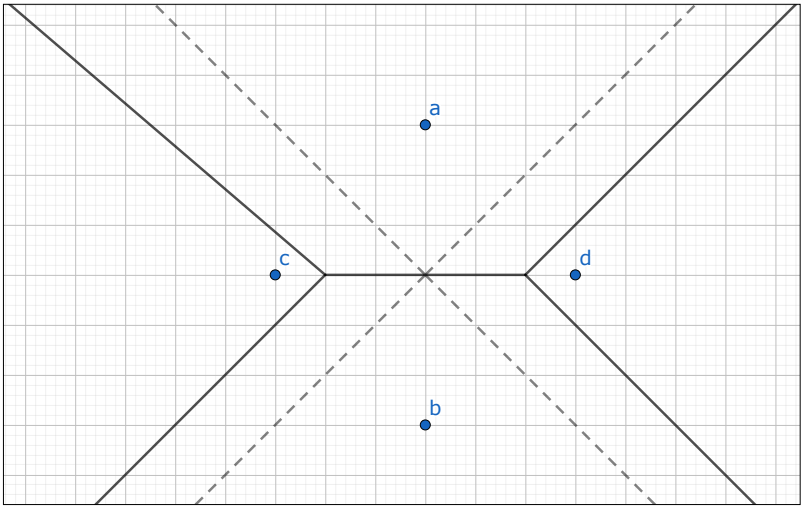

Figure 4.6: The dashed line represents the standard Voronoi diagram of the four sites. The solid line is a weighted Voronoi diagram in which the sites a and b have been assigned higher weight, increasing the size of their respective regions. [Screenshot taken by the author using GeoGebra [GeoGebra 2021].]

for a multiplicatively weighted diagram. The main drawback of these methods is that they yield curved rather than straight edges, which is often undesirable.

Thus, *power diagrams* are often used when weights should be taken into account. In power diagrams, circles or spheres define the area of influence of a site. The circles' or spheres' radius is calculated via the assigned weight. For example for the site **p**:

$$r(p) = \sqrt{weight(p)} \tag{4.6}$$

This method then also yields a power function through which the influence of a site on a point can be calculated:

$$pow(x, p) = (x - p)^T (x - p) - weight(p) \tag{4.7}$$

The function is zero on the outline of the circle or sphere and negative within. For all points outside the circle or sphere it returns a positive value. An example of a power diagram for three points can be seen in Figure 4.7.

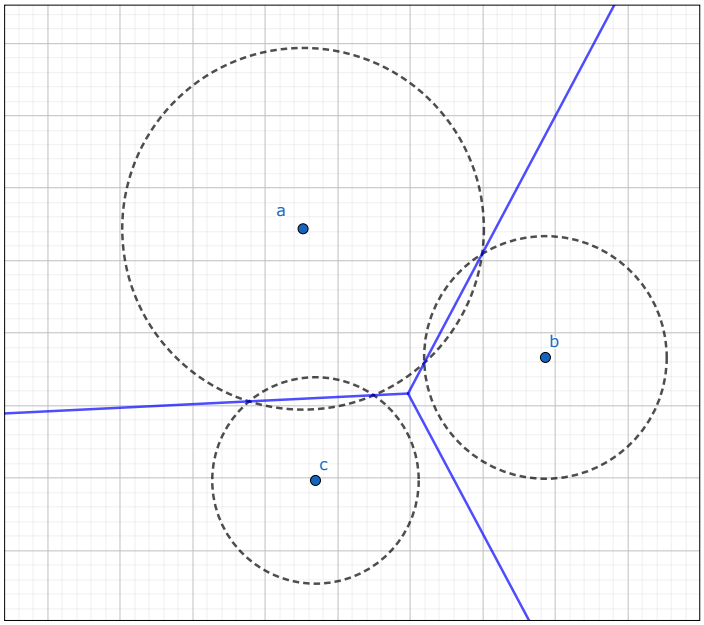

Figure 4.7: The power diagram for the three sites is represented by the blue lines. As can be seen from the dashed circles, site a has a larger weight assigned to it compared to the other two sites. This results in its region being considerably larger. [Screenshot taken by the author using GeoGebra [GeoGebra 2021].]

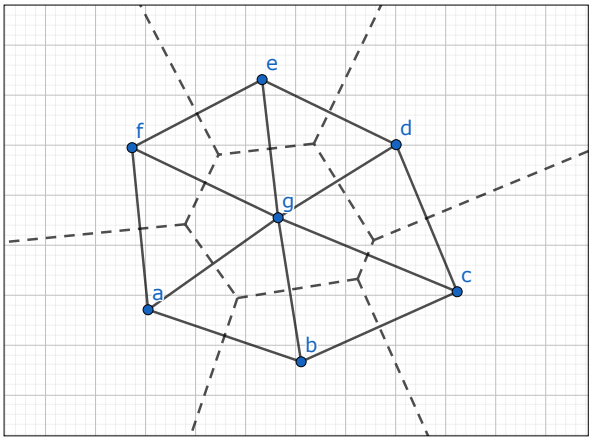

Figure 4.8: The dashed line represents the Voronoi diagram of the set of seven sites. The solid line is the corresponding Delaunay tessellation, which in this case is a Delaunay triangulation, since the sites are in general position. [Screenshot taken by the author using GeoGebra [GeoGebra 2021]]

4.2 Delaunay Tessellation

When discussing the Voronoi diagram, it is important to also talk about the Delaunay tessellation, since they are connected in many ways. While a Voronoi diagram constructs regions around every site in the set, the Delaunay tessellation uses the sites as endpoints for edges which form polygons [Aurenhammer et al. 2013, pages 11–14]. The Delaunay tessellation is the dual of the Voronoi diagram, and one can be derived from the other.

To construct a Delaunay tessellation, all sites which are co-circular, while not including any other sites within the circle, are joined via an edge. So, for every two sites that can be connected via a circle, one edge to the tessellation can be added [Aurenhammer et al. 2013, page 12].

If the sites used for the Delaunay tessellation are in general position, it turns into a Delaunay triangulation, since all polygons in the tessellation are triangles. Sites are in general position, if no three of them are co-linear and no four of them are co-circular. In this case, any three sites that are co-circular and whose circle does not include other sites, can be joined to form a new triangle in the triangulation. Such a Delaunay triangulation can be seen in Figure 4.8 [Aurenhammer et al. 2013, page 12].

The Delaunay tessellation or Delaunay triangulation is the dual of the Voronoi diagram. For every Voronoi edge there is a Delaunay edge, for every Voronoi vertex there is a Delaunay face, and for every Delaunay vertex there is a Voronoi region. This can also be seen in Figure 4.8. This duality is very useful for many algorithms. Whenever one structure is given, the other can be computed, which provides many possibilities to approach the construction of either one of them [Aurenhammer et al. 2013, page 12].

4.3 Voronoi Treemaps

Voronoi treemaps are a combination of treemaps and Voronoi diagrams with the goal of visualizing and interacting with information hierarchies. In standard treemaps, the space allocated to children is divided into strips or rectangles recursively. In Voronoi treemaps, the space is partitioned into Voronoi regions. In interactive treemaps, levels of the tree can be explored by zooming, expanding, and collapsing.

The polygons in a Voronoi treemap are based on Voronoi diagrams. Working top-down, a site is placed for each child and a Voronoi region based on its weight is constructed around it, effectively partitioning the parent's space amongst the children. This continues recursively to the bottom of the hierarchy.

Not all levels of the Voronoi hierarchy have to be displayed at the same time, typically one or two levels are shown at a time, and the hierarchy is explored interactively. One way to make Voronoi treemaps interactive is by monitoring the user's zoom level and automatically opening or closing child cells as the user zooms in or out. This makes for intuitive exploration of the data, as the data show itself without the user explicitly asking for it. Users can zoom by mouse wheel or by pinch-zoom on touch devices. Another way of exploring a Voronoi treemap is by reacting to deliberate choices by the user. The user can click or tap, depending on the device, to open up a cell and display its child cells. This gives the user full control over what is seen.

4.3.1 Andrews et al [2002]

The Voronoi treemaps technique was first introduced in 2002 by Andrews et al. [2002] as part of the InfoSky system, although it was not called Voronoi treemaps. Hierarchy levels were represented as Voronoi regions and every subsequent level was made up of Voronoi regions within the parent region, as can be seen in Figure 4.9. Individual documents in the hierarchy were represented by white dots, resembling stars in a galaxy. Documents at each level were collected into synthetic Voronoi cells. Both Voronoi sites and documents were placed according to their similarity using force-directed placement. Users could zoom by mouse wheel and open child cells by clicking. The system was later further refined and evaluated [Kappe et al. 2003; Andrews et al. 2004; Granitzer et al. 2004].

4.3.2 Balzer and Deussen [2005]

In 2005, Balzer and Deussen [2005] published their paper on Voronoi treemaps, which coined the name for the technique, without referring to the prior work. The paper presents an approach to generate Voronoi treemaps within arbitrary polygons. Such a visualization can be viewed in Figure 4.10. In contrast to Andrews et al. [2002], the paper does not present an interactive application, but rather a new procedure to generate static Voronoi treemaps. This procedure could not be efficiently used for real-time applications, as its construction time was too long. The visualization in Figure 4.10 took 7 minutes and 13 seconds to construct at the time of the paper [Balzer and Deussen 2005].

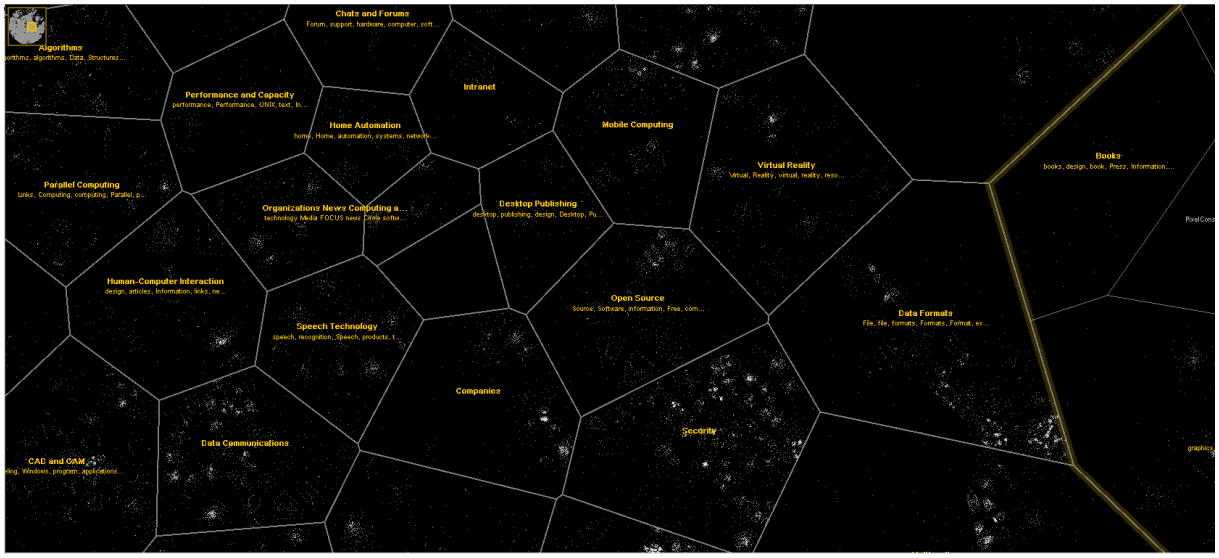

Figure 4.9: The InfoSky explorer is based on the metaphor of stars in galaxies. The hierarchy is recursively divided into Voronoi regions and subregions. Documents are represented as white dots, resembling stars. [Screenshot taken by the author using the InfoSky application [Andrews et al. 2002].]

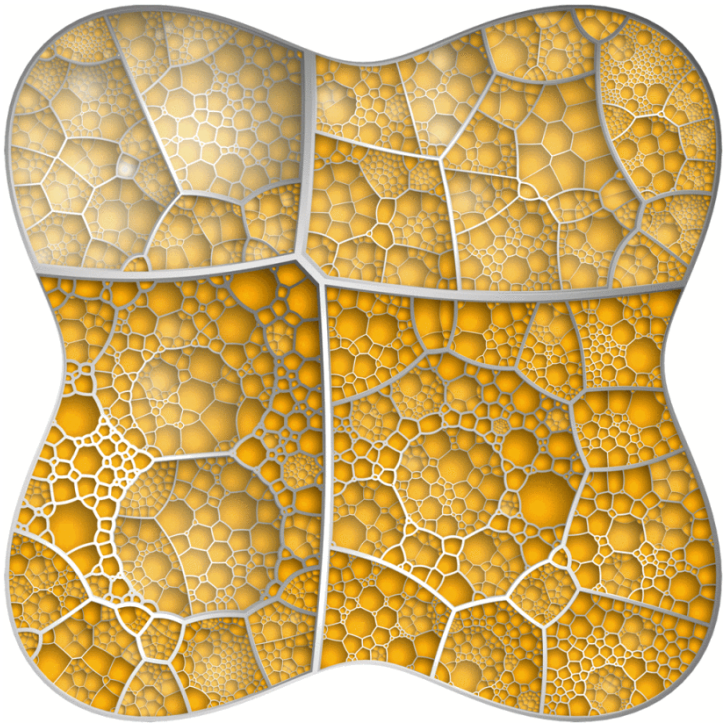

Figure 4.10: A static Voronoi treemap visualization showing all levels. [Extracted from Balzer and Deussen [2005]. © 2005 IEEE, used with permission.]

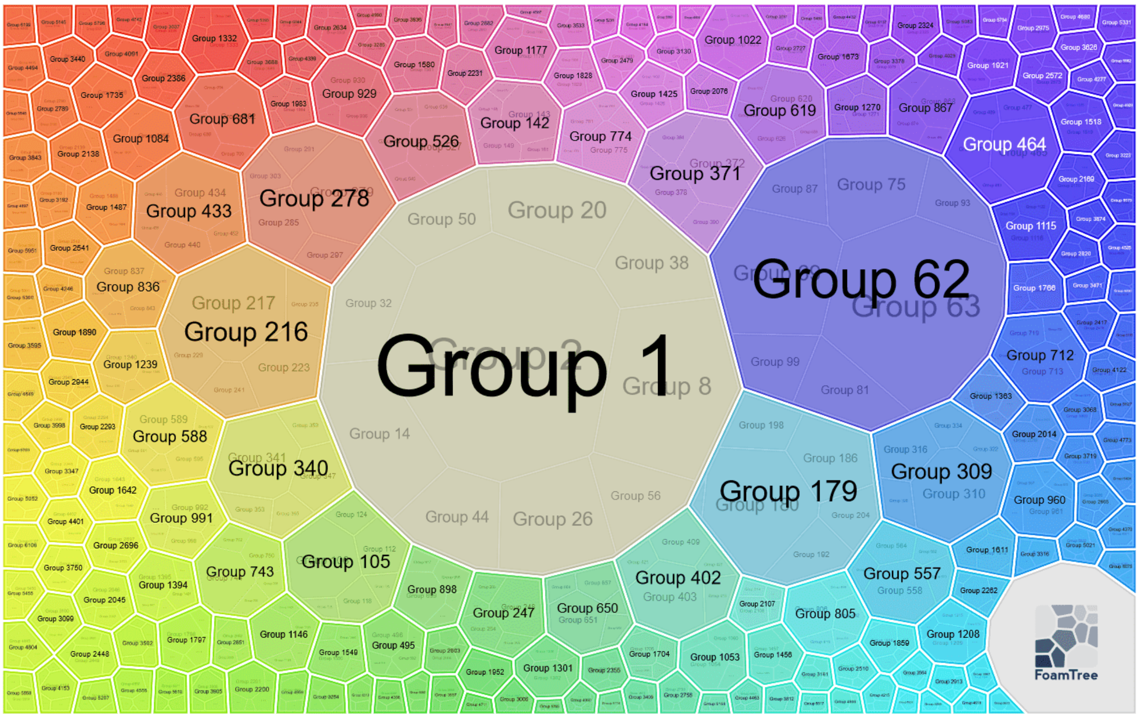

Figure 4.11: A standard demonstration dataset loaded into FoamTree. It displays 2 levels of children at any time. [Screenshot taken by the author using FoamTree [Carrot Search 2021].]

4.3.3 Sud et al [2010]

In 2010, Sud et al. [2010] published their paper "Fast Dynamic Voronoi Treemaps". It presents an efficient algorithm utilizing GPU processing power and enables dynamic updates to the Voronoi treemap. By using their GPU-based implementation they managed to improve the performance by nearly double. As with Balzer and Deussen [2005], the implementation enables the construction of Voronoi treemaps within arbitrary polygons.

4.3.4 Nocaj and Brandes [2012]

Nocaj and Brandes [2012] published an improved algorithm for constructing Voronoi treemaps. They present an algorithm that improves the runtime from $\Omega(n^2)$ to $O(n \log n)$. The algorithm takes the scheme from Balzer and Deussen [2005] and reduces the time needed by processing iterations more efficiently. The d3-voronoi-treemap implementation of Voronoi treemaps, which is used in VoroTree and VoroLib, is based on the algorithm described in the paper [LeBeau 2022].

4.3.5 FoamTree [2012]

FoamTree by Carrot Search is a commercial software library implementing Voronoi treemaps, which was first released in 2012 [Carrot Search 2021]. Figure 4.11 shows FoamTree displaying one of its standard demo datasets. A free-to-use branded version is available, but the software is not open-source. The pricing for an unbranded version is only available upon inquiry.

FoamTree is written in JavaScript and is intended to be integrated into web pages. The application is highly customizable and allows broad customization options. Even though the possibilities are numerous, there is considerable overhead involved in the construction of every FoamTree visualization. Datasets need to be in FoamTree's expected format, standard formats such as JSON or CSV are not supported. Also, any customization requires in-depth knowledge of the application documentation or prior knowledge of

Figure 4.12: A special Voronoi treemap containing images. Clicking on an image yields no children but rather shows a larger version of it. [Screenshot taken by the author using FoamTree [Carrot Search 2021].]

Carrot Search applications. However, it is possible to construct innovative visualizations with FoamTree, once adept at the procedure. An example can be seen in Figure 4.12 which shows a Voronoi treemap filled with pictures.

Interactivity is handled by clicks from the user, once a polygon is double-clicked, it opens up to its child polygons. This also works on touch devices with taps to the screen. FoamTree does not support exploration via zoom, with one exception. It is possible to zoom all the way out, using the mouse wheel or pinching on a touch device, which resets the visualization to its initial state.

4.3.6 van Hees and Hage [2015]

Following van Hees Master's thesis [van Hees 2014], van Hees and Hage [2015] published a Voronoi treemap implementation which remains stable to small changes. They highlight that standard algorithms are nondeterministic, meaning if small changes occur, this can lead to large changes in the visualization. As an alternative they present a deterministic algorithm which remains stable. Furthermore, they present a solution using coloring to indicate the differences between two (similar) hierarchies. An example of a Voronoi treemap by van Hees and Hage can be viewed in Figure 4.13

4.3.7 Interactive Voronoi Treemap (IVT) [2020]

In 2020, development on another Voronoi treemap implementation started at Graz University of Technology under the name Interactive Voronoi Treemap (IVT) [Oser et al. 2021]. Like FoamTree, it is a web-based application that offers an interactive Voronoi treemap that can be browsed by the user. Whilst offering similar basic functionality, IVT is free of charge and open source. It works with both standardized JSON and CSV files and also offers example data sets to test the application. For calculation of the Voronoi diagrams, IVT uses the d3-voronoi-treemap library [LeBeau 2022] and PixiJS [Groves 2022] for all visual aspects. A screenshot of the application can be seen in Figure 4.14. Features still to be

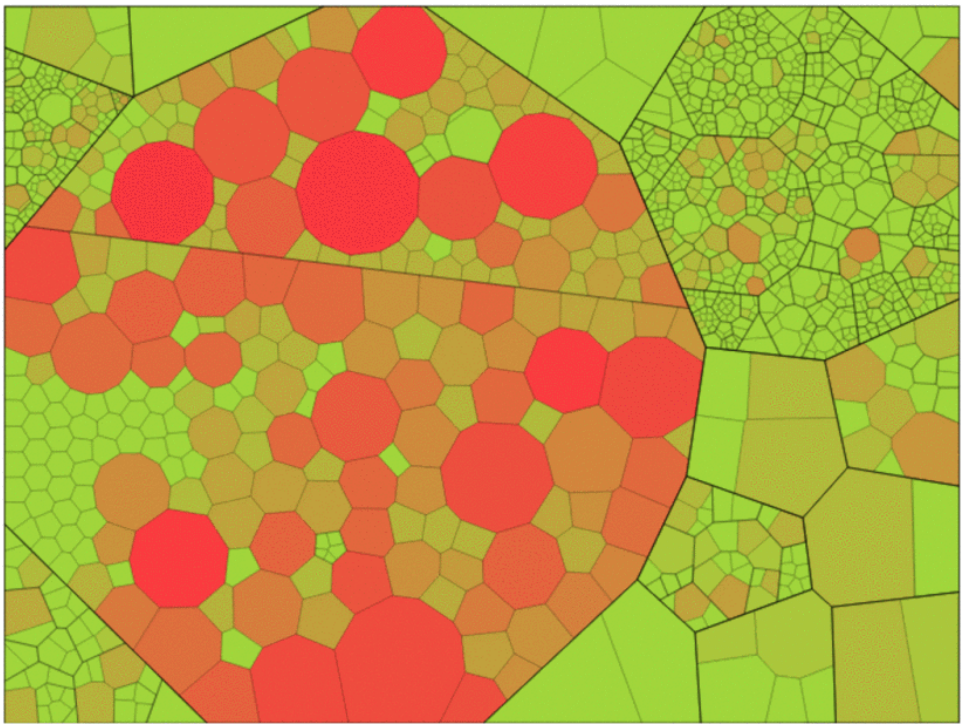

Figure 4.13: Voronoi treemap of a software application. Each leaf node represents a method, and higher levels represent the class and package structure. The color represents the McCabe's complexity in the visualization. [Extracted from van Hees and Hage [2015]. © 2015 IEEE, used with permission.]

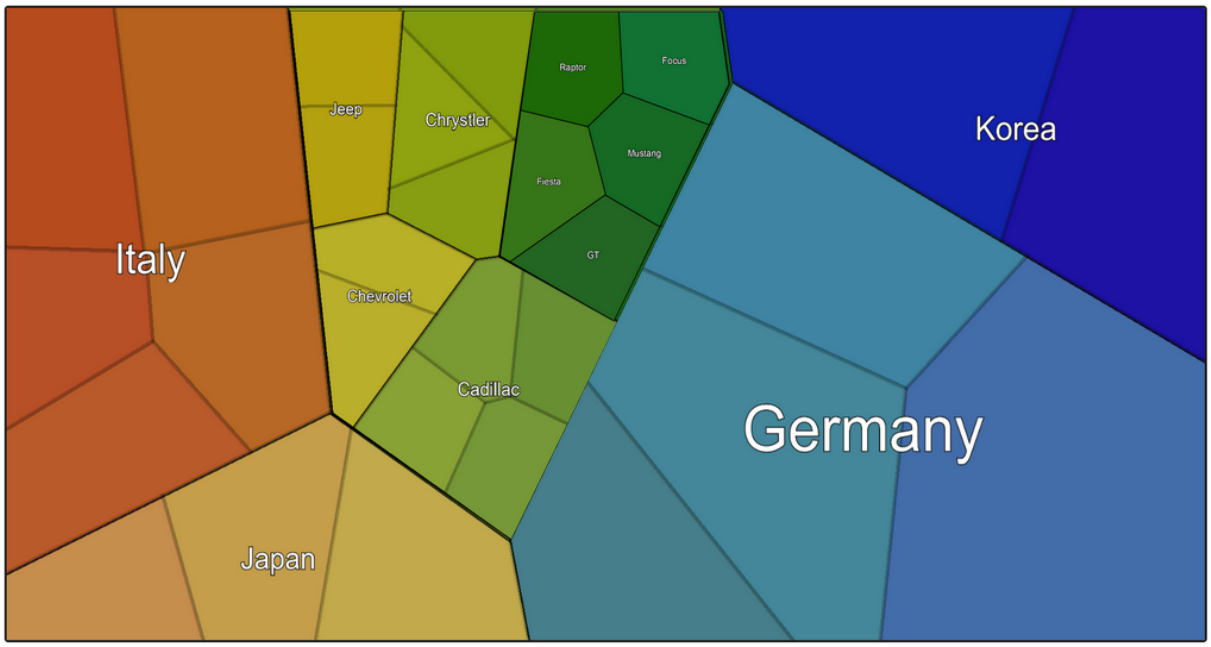

Figure 4.14: A car manufacturer hierarchy loaded into the Interactive Voronoi Treemap (IVT) software. The first level indicates nationality, the second level manufacturer, and the third level model. [Screenshot taken and constructed by the author using IVT [Oser et al. 2021].]

implemented in IVT include: improvement of the UI, support for touch devices, browsing by zooming, and some performance optimization.

Development of IVT was frozen in 2021. However, the source code of IVT was taken to form the basis of the VoroTree and VoroLib software described in the remaining chapters of this thesis.

4.4 Characteristics of Voronoi Treemaps

The various Voronoi treemap systems differ in terms of how they approach and implement certain issues. Sites can be placed in various ways to yield different results. Depending on the placement, algorithms can be deterministic or nondeterministic. There are also variations in leaf node placement and navigation options for the user.

4.4.1 Placing Sites

To construct a Voronoi diagram, and therefore also a Voronoi treemap, sites need to be placed to form Voronoi regions. Many implementations use random placement of sites which are then moved to fit the visualization better [Balzer and Deussen 2005; Nocaj and Brandes 2012; Oser 2022b]. The placement of sites, in this case, can either occur truly random or through a seed. If the placement is done randomly, this makes the algorithm nondeterministic, meaning that the same data will produce a different looking visualization each time it is constructed. If a static seed is used, the algorithm is deterministic, and a visualization will look the same when reconstructed for the same data. VoroTree and VoroLib give the user the option to choose between dynamic and static seeds [Oser 2022b; Oser 2022d].

The implementation of van Hees imposes ordering on the sites [van Hees 2014]. Sites are ordered according to the underlying data, sites occurring later within the data, are placed later onto the plane. This makes the implementation deterministic and stable to changes within the data. Small changes to the data will not greatly change the resulting visualization.

InfoSky uses force-directed placement (FDP) to place the sites, so that similar child nodes are placed close to one another [Andrews et al. 2002]. Each site has a feature vector representing its characteristics and sites at each level of the tree are placed according to similarity.

4.4.2 Representing Leaf Nodes

There are different ways of visualizing leaf nodes in the hierarchy. Consider the example of a file system with folders, subfolders, and documents. Documents can be placed at any level in the file system hierarchy, they are not restricted to being placed only in folders containing no further subfolders.

FoamTree, as well as both VoroTree and VoroLib, create Voronoi cells for each leaf node. An alternative way of visualizing leaves is by collecting all the leaves at a particular level into a specially designated Voronoi cell. This is the approach taken by InfoSky [Andrews et al. 2002] and can be seen in Figure 4.15.

4.4.3 Visual Encoding

Visual encoding is used to increase the amount of information presented by a Voronoi treemap. Some implementations make the size of Voronoi cells represent a particular attribute of the data. InfoSky adjusts the size in respect to the number of documents held by that node [Andrews et al. 2002]. VoroTree and VoroLib allow to choose any numerical value within the data to be used to weight the visualization. If no input is given, the size corresponds to the total number of leaf nodes contained by that Voronoi cell [Oser 2022b; Oser 2022d].

Color is also used to represent information. In InfoSky, document labels are white, while regular tree node labels are yellow [Andrews et al. 2002]. VoroTree offers a color scheme which highlights file

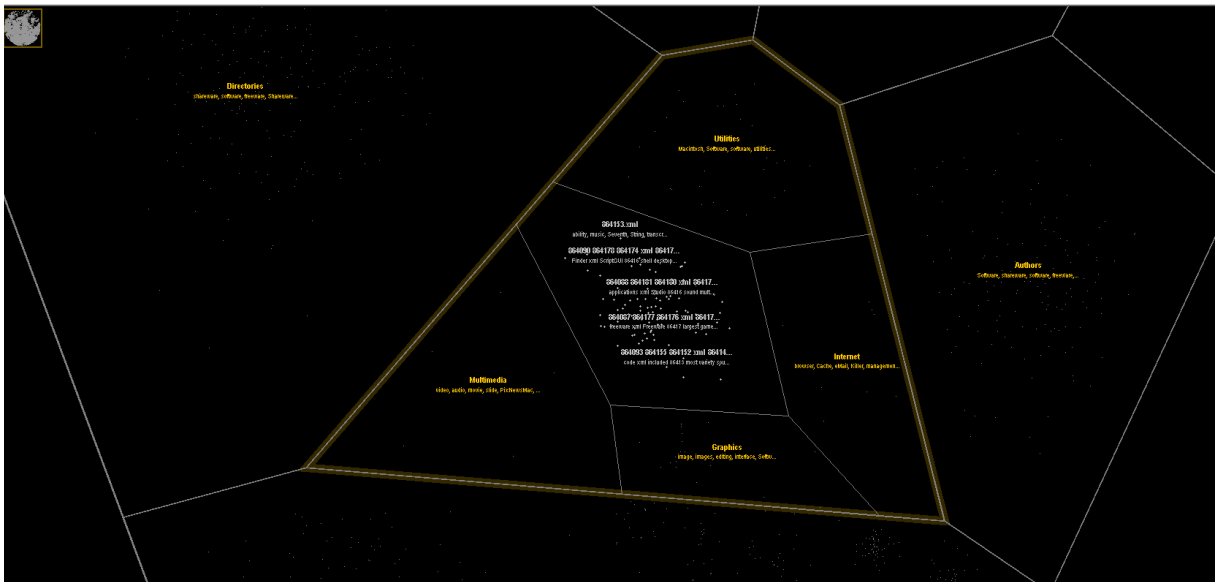

Figure 4.15: InfoSky collects leaf nodes at a particular level into a specially designated Voronoi cell. [Used with kind permission of Keith Andrews.]

types in certain colors and adds icons [Oser 2022d]. It would also be possible to provide the user with a sophisticated visual encoding panel to choose mappings for both size and color, like the original Treemap software does, as shown in Figure 4.16.

4.4.4 Navigational Facilities

User interaction can be handled in various ways. The two main navigation options in current implementations are based on either dedicated selections by the user or by monitoring the users actions and reacting passively. Dedicated selections by the user can be mouse clicks or taps on touch devices. If a Voronoi cell (or its label), is clicked or tapped, the cell is opened and its child Voronoi cells are displayed. This is implemented in FoamTree [Carrot Search 2021], InfoSky [Andrews et al. 2002], VoroTree [Oser 2022d], and VoroLib [Oser 2022b].

Opening and closing cells passively is done by monitoring the user's zoom actions. The user's zoom level is changed by mouse wheel movement or pinching on touch devices. If a user zooms in, the application can automatically open up a Voronoi cell once it fills a certain proportion of the current viewport. When the user zooms out, cells below the threshold can be automatically collapsed again. This is currently implemented by VoroTree [Oser 2022d] and VoroLib [Oser 2022b].

In addition to the Voronoi treemap visualization, InfoSky [Andrews et al. 2002] offers a supplementary outline view of the hierarchy to the left, as can be seen in Figure 4.17. The two views are synchronized. Selecting a folder in the outline view, navigates the user to the respective Voronoi cell and vice versa.

Finally, another idea for browsing a Voronoi treemap is by text search. The user enters a search query which is processed by the application. Matching nodes in the tree can then be highlighted, and potentially navigated to. InfoSky implements this for documents (leaf nodes), color-coding them by search term, as can also be seen in Figure 4.17.

4.4.5 Responsive Design

Voronoi treemaps should be responsive. The visualization should adapt to changes in window size or orientation to take the best advantage of the available space. InfoSky scales the visualization in respect

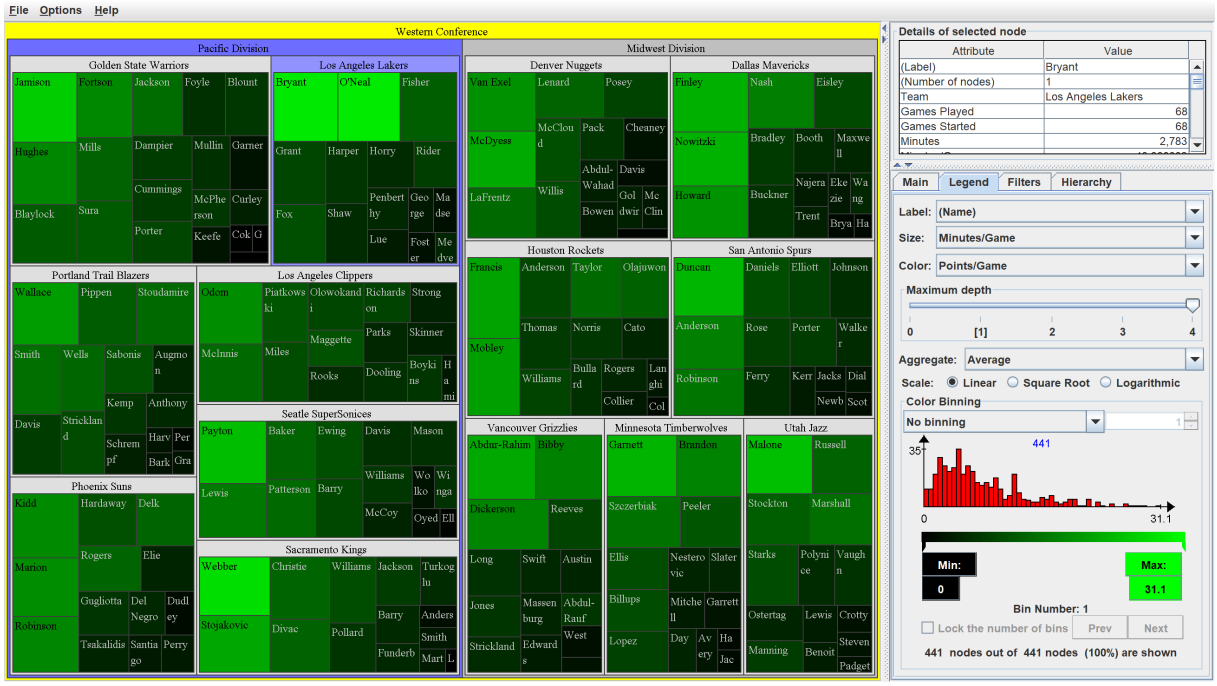

Figure 4.16: The original Treemap software has a sophisticated panel (on the right) through which the user can specify mappings for size and color. [Used with kind permission of Keith Andrews.]

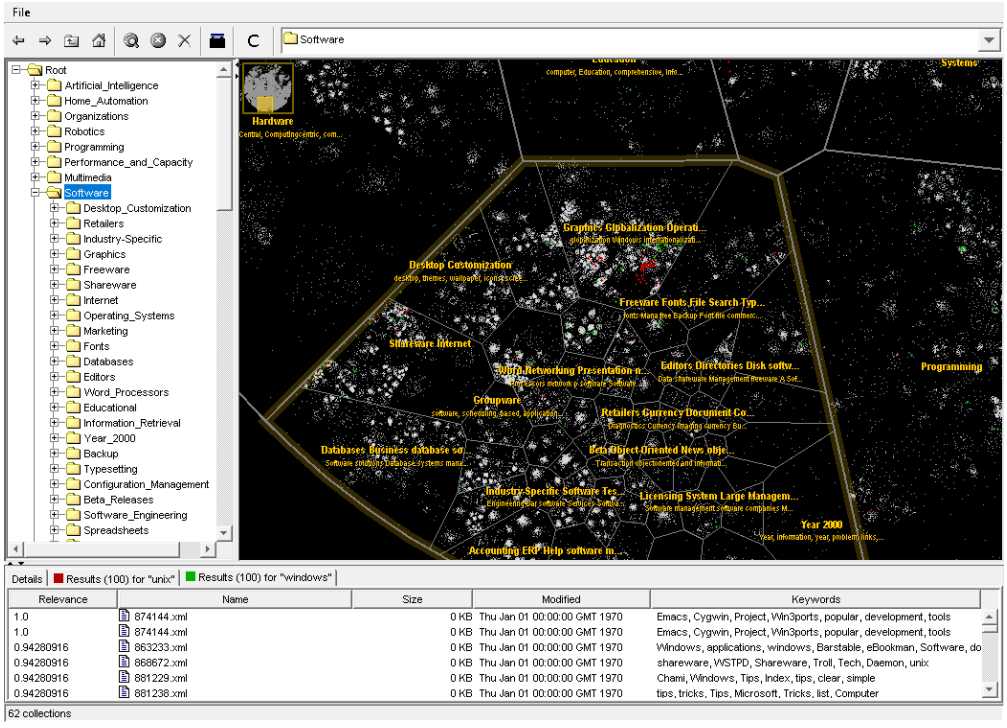

Figure 4.17: InfoSky implements a text search function which highlights matching leaf nodes in the hierarchy. Here, red indicates nodes matching the search term “unix”, and green indicates nodes matching the search term “windows”. [Used with kind permission of Keith Andrews.]

to window size. This means the Voronoi treemap retains its structure, but is scaled in size to fill the available space [Andrews et al. 2002].

FoamTree [Carrot Search 2021], VoroTree [Oser 2022d], and VoroLib [Oser 2022b] all reconstruct the Voronoi treemap upon changes to size or orientation, so the visualization best fits the new dimensions. However, for larger datasets, this can mean significant processing times.

In order to be fully responsive, Voronoi treemap applications should also support all means of interaction which are available on the device. For example, mouse click, mouse drag, mouse wheel, tap, drag, pinch zoom, and keyboard commands, wherever they are available.

Chapter 5

VoroTree

This chapter presents the VoroTree web application [Oser 2022d], with which users can open and explore arbitrary tree structures using Voronoi treemaps. The VoroTree application is shown in Figure 5.1.

VoroTree is based on a previous application called Interactive Voronoi Treemap (IVT) [Oser et al. 2021], which had already implemented the basic features of an interactive Voronoi treemap and had a simple GUI. To start development on VoroTree, all dependencies were updated and optimized, and thereafter the source code was adjusted accordingly. The entire IVT user interface was removed including all its dependencies. A new GUI was developed purely in HTML and CSS. The task runner gulp was integrated to help automate project building and maintenance.

The new GUI is based on a single menu bar at the top with multiple dropdown menus. The Open File menu allows the user to open an arbitrary hierarchy stored in a file. The Examples menu lists a selection of provided example hierarchies. The Settings menu gives access to various options, such as breadcrumbs, color scheme, static or dynamic cell placement, font sizing, and which attribute of the hierarchy to use as the weight for cell sizing. In addition, the SVG Export button in the Settings menu allows a snapshot of the current view to be exported in SVG format. Finally, the Reset View button resets visualization to its original top-level state. The menu bar and the Settings menu can be seen in Figure 5.2.

VoroTree is a self-contained application for interactive Voronoi treemaps. Support for offline use on all major platforms was added through Electron [OpenJS 2022a]. It is possible to build VoroTree as a standalone desktop application for Windows, macOS, and Linux. Electron also saves all desktop application settings, such as window size and position and loads them on application restart.

VoroTree is a responsive web application, meaning it adjusts to different window sizes and resolutions. Also, when a user resizes the application window during execution, the application adjusts its content to the new size, including the Voronoi treemap which is recalculated for the new dimensions on window size changes. In addition to being responsive, the application also supports mobile and touch devices. Taps and pinches are handled by the application and enable the user to intuitively interact with the visualization.

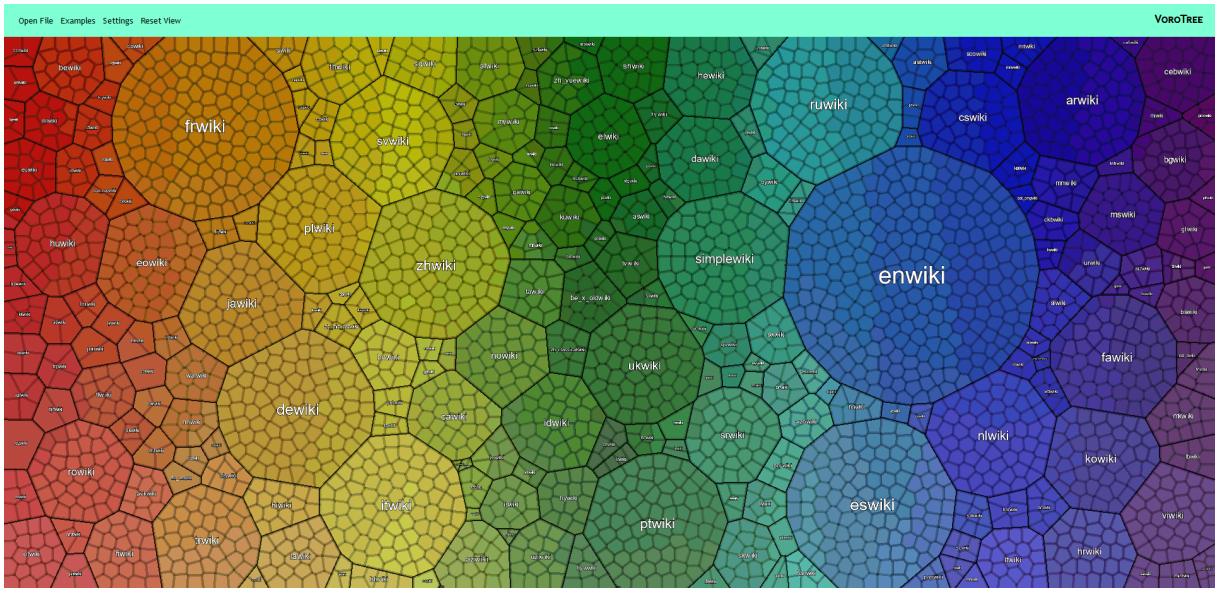

Figure 5.1: The VoroTree web application with a Wikipedia editor dataset and the rainbow color scheme selected. [Screenshot taken by the author using VoroTree.]

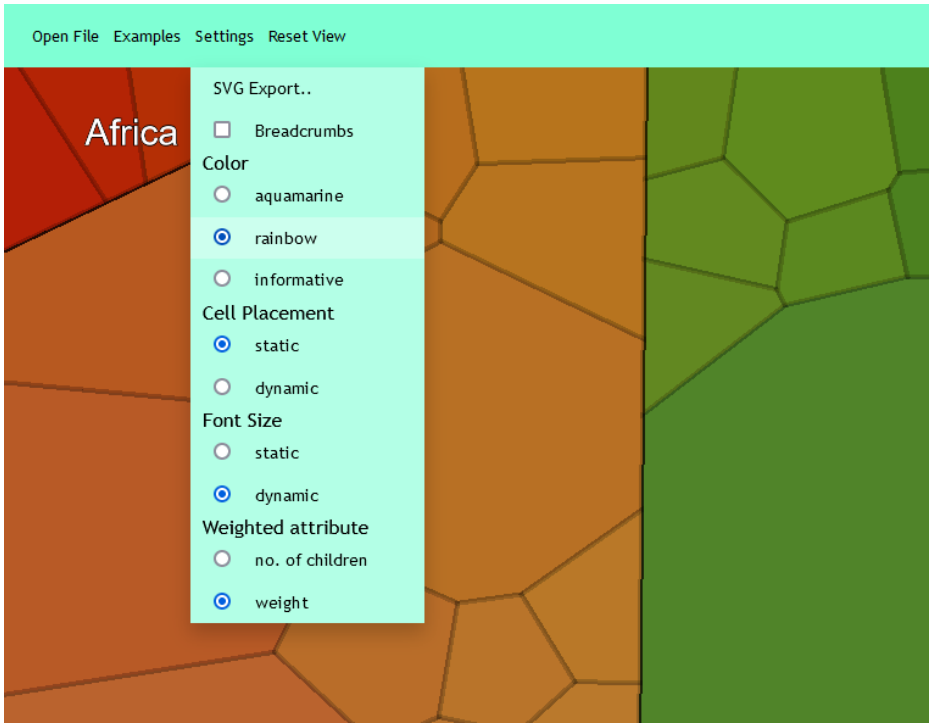

Figure 5.2: All VoroTree options are listed in the Settings menu. This includes visual and structural changes to the visualization. [Screenshot taken by the author using VoroTree.]

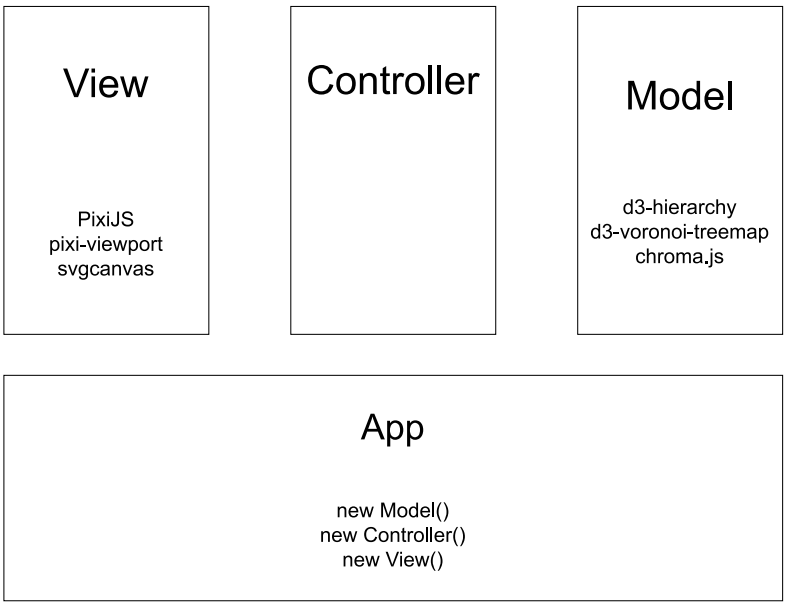

Figure 5.3: The software architecture of VoroTree. The App class calls the constructor of all three main classes to instantiate the application. View and Model both have important external dependencies. [Diagram drawn by the author.]

5.1 Software Architecture

The VoroTree code uses the popular Model-View-Controller paradigm, by which general execution is split into three classes. The Model class holds all knowledge and serves as the application's database. The View class is responsible for all visual functionality of the application. This includes drawing the visualization and any customization procedures. The Controller class handles all interaction by the user. For VoroTree, this is made up of interaction with the visualization and operation of the menu bar. A visual overview on the classes used in VoroTree can be seen in Figure 5.3.

The Model class holds all knowledge and serves as the application's database. The input data and all methods associated with it are held by this class. Initial structuring of input data as well as custom edits to it, like icons for file types, are all done within the Model class. General settings for the visualization are also held here. Examples for this are the weight attribute, font sizing strategy, and cell placement method. The cell placement is defined by a seed which stays constant or is randomized, depending on what the user wants. The Model class also holds the color scheme used by the visualization and the callback function for cells, if one is specified. The path or file used for the input data is also saved in the class, in case the visualization needs to be reloaded or adjusted and reconstructed.

The View class is responsible for all visual functionality of the application. It draws all parts of the visualization: polygons, labels, and colors are all drawn within the class. Any visual adjustments like resetting the viewpoint or redrawing the entire visualization are also possible through methods in the View class. To do this, the View holds all necessary PixiJS objects. On the one hand this means the PixiJS app itself, which holds the renderer and general settings of the visualization, and on the other hand this includes all polygon and label objects needed to draw the visualization. The View class also holds the functionality to draw and export an SVG of the current visualization state.

Finally, the Controller class handles all actions of the user and responds accordingly. Clicks are handled by the Controller and the respective action is then sent on to the Model and View to fulfill the user's expectation. Zoom actions are also handled in the Controller. Corresponding tap and pinch actions are also handled for touch devices. The Controller class defines the zoom ratio that decides at what zoom percentage child polygons are displayed automatically. When the View class has finished constructing an exported SVG,

the Controller class handles the procedure to offer the user the possibility to download the file.

VoroTree has a fourth main class, the App class. This class launches the constructors of the Model, View, and Controller classes and initializes the application. It also sets the resulting variables to be global, so they can be used throughout the application. Since the initial state of VoroTree displays no visualization, it is up to the user to load a dataset after starting the application.

5.2 Project Infrastructure

VoroTree is built using Node.js [OpenJS 2022b] and its package manager npm [npm 2022] to maintain and administrate all external dependencies. Development tasks are automated by the task runner gulp [Bublitz and Schoffstall 2022]. It is possible to build and serve the project with single-line commands. Furthermore, standalone application building, cleaning procedures, and a utility function for constructing folder datasets are all handled by gulp commands. The exact gulp commands are listed in Listing 5.1.

The application is based on a single HTML file. This file holds the general structure of the application, specifically the main menu bar. Button clicks are sent to the respective JavaScript function in the source code. All visual adjustments are done through CSS. A single CSS file sets the visual appearance of all HTML elements in the application. A listing of the VoroTree folder structure can be viewed in Figure 5.4.

```
Gulp tasks for VoroTree
|-- build
|-- serve
|-- default
|-- electron
| |-- <series>
|   |-- build
|   |-- runElectron
|-- buildElectronApp
| |-- <series>
|   |-- build
|   |-- buildElectron
|-- buildElectronWinApp
| |-- <series>
|   |-- build
|   |-- buildElectronWin
|-- buildElectronMacApp
| |-- <series>
|   |-- build
|   |-- buildElectronMac
|-- buildElectronLinuxApp
| |-- <series>
|   |-- build
|   |-- buildElectronLinux
|-- buildElectronAllApps
| |-- <series>
|   |-- build
|   |-- buildElectronWin
|   |-- buildElectronMac
|   |-- buildElectronLinux
|-- clean
|-- cleanAll
| |-- <series>
|   |-- cleanDist
|   |-- cleanNodeModules
|-- deploy
| |-- <series>
|   |-- build
|   |-- deployVoroTree
|-- constructFolderDataset
```

Listing 5.1: All gulp tasks available in VoroTree.

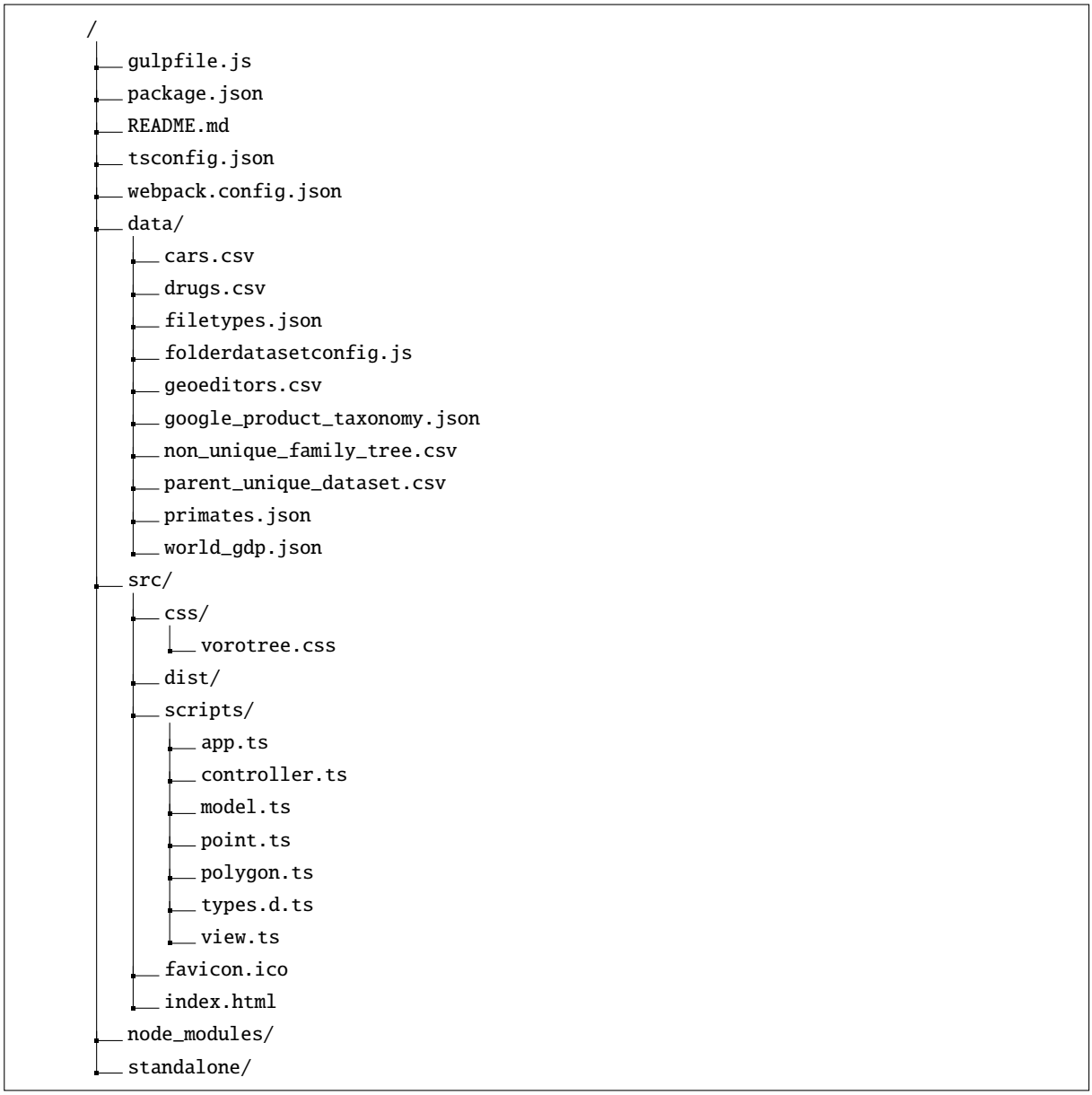

Figure 5.4: The most important files and directories of the VoroTree project. [Figure created by the author.]

5.3 Dependencies

VoroTree requires a number of external dependencies. Listing 5.2 lists all of them as depicted in the package JSON project file.

To construct the Voronoi treemap, VoroTree uses the d3-voronoi-treemap library [LeBeau 2022]. The library uses the "lift to 3-space" algorithm to compute weighted Voronoi diagrams as explained in Chapter 4. The algorithm is looped while adapting positions and weights until the error reaches a reasonably small value. The drawback with this is that the entire treemap must be constructed before being able to visualize the top layer. Therefore, for large datasets, the calculation can require more time than usual visualizations.

Another D3 library used in VoroTree is d3-delaunay [Observable 2022]. While the library's main purpose is Delaunay tessellation and constructing Voronoi diagrams from it, this is not what it is used for in VoroTree. As explained above, VoroTree uses the "lift to 3-space" algorithm for this purpose. The d3-delaunay library contains a polygon data type that is used for the generation of VoroTree's polygon class. After d3-voronoi-treemap constructs the Voronoi treemap, the resulting data structure contains a d3-delaunay polygon. This d3-delaunay polygon needs to be processed to transfer its information to a VoroTree polygon. This is the only purpose of the d3-delaunay library in VoroTree.

A third D3 library used in VoroTree is d3-hierarchy. This library is used to convert input data from CSV or JSON files to a D3 hierarchy data structure. The d3-voronoi-treemap library requires this data structure to start calculation on the Voronoi treemap [Bostock 2022].

All drawing in VoroTree is done via PixiJS [Groves 2022]. If the web browser supports it, the visualization is done in WebGL. If the web browser does not support WebGL, HTML5 canvas is used. All polygons are drawn and saved using the PixiJS graphics object. The polygon data structure defined in VoroTree is built on top of a PixiJS graphics object, to simplify the process of drawing the objects. The polygon's labels are visualized using the PixiJS text object.

All interaction by the user, including clicks, taps, zooming and pinching are handled by pixi-viewport [Figatner 2021], a library developed to be used in conjunction with PixiJS. It offers support for all general interaction types and customizing the viewport of a PixiJS application. For clicks and taps, an animation is triggered that simultaneously moves and zooms the viewport to the center of the desired polygon.

Other libraries used in VoroTree include seedrandom [Bau 2019], chroma.js [Aisch 2022], and SVG-Canvas [Zeng 2021]. The seedrandom library is used for seed management of the initial seed of the Voronoi treemap. For all color management and color scales in VoroTree, chroma.js is used to simplify the task of producing color scales by coordinates. The SVGCanvas library is used to construct SVG files programmatically. It offers HTML5 canvas-like functions which can be used to draw serialized SVG files. An SVG export from VoroTree can be seen in Figure 5.5.

A further library, directory-tree [Dobrescu-Balaur 2021], is used by a utility script to produce a JSON file from a given folder in a file system. This process can be triggered via a gulp command. The JSON file can then be used as input for a VoroTree visualization.

The user interface elements (menu bar, menus, and buttons) of VoroTree are developed entirely in HTML and CSS and require no external dependencies. The entire GUI around the Voronoi treemap is a minimalistic approach to an interface for an application which also works intuitively when run as a standalone application.

As well as being a web application hosted on a web server, VoroTree can also be built as a standalone application for offline use. For this purpose Electron is used [OpenJS 2022a]. Electron is an open-source software framework for creating native desktop applications from web software products. In essence, a headless version of the Chromium browser is packaged with the web application to form a standalone executable. Electron handles construction of binaries for all major operating systems, including Windows, macOS, and Linux.

```
{
  "name": "vorotree-project",
  "productName": "VoroTree",
  "version": "0.0.1",
  "description": "Interactive Voronoi Treemap Visualization",
  "main": "src/main.js",
  "repository": {
    "type": "git"
  },
  "author": "Christopher Oser <christopher.oser@student.tugraz.at>",
  "license": "MIT",
  "devDependencies": {
    "@types/d3-delaunay": "^6.0.0",
    "@types/d3-fetch": "^3.0.1",
    "@types/d3-hierarchy": "^3.0.2",
    "clean-webpack-plugin": "^4.0.0",
    "copy-webpack-plugin": "^10.0.0",
    "copyfiles": "^2.4.1",
    "electron": "^15.3.2",
    "electron-packager": "^15.4.0",
    "gulp": "^4.0.2",
    "gulp-cli": "^2.3.0",
    "ts-loader": "^9.2.6",
    "typescript": "^4.5.2",
    "webpack": "^5.64.4",
    "webpack-cli": "^4.9.1",
    "webpack-dev-server": "^4.6.0"
  },
  "dependencies": {
    "@types/chroma-js": "^2.1.3",
    "chroma-js": "^2.1.2",
    "d3": "^7.1.1",
    "d3-delaunay": "^6.0.2",
    "d3-fetch": "^3.0.1",
    "d3-hierarchy": "^3.0.1",
    "d3-voronoi-treemap": "^1.1.1",
    "d3-weighted-voronoi": "^1.1.1",
    "directory-tree": "^3.0.1",
    "electron-window-state": "^5.0.3",
    "path-browserify": "^1.0.1",
    "pixi-viewport": "^4.34.1",
    "pixi.js": "^6.2.0",
    "seedrandom": "^3.0.5",
    "svgcanvas": "^2.0.5"
  }
}
```

Listing 5.2: VoroTree dependencies as depicted in package.json.

Figure 5.5: An SVG export of the uppermost layer of a Voronoi treemap of a product taxonomy dataset. [Exported from VoroTree by the author and converted to PDF for inclusion here.]

5.4 Purpose and Capabilities

VoroTree is a self-contained web application for interactive Voronoi treemaps. It offers example datasets and customization options which can be adjusted by users. The application offers six example datasets with which the visualization can be tested. One smaller example dataset, containing 50 nodes in two levels, shows the GDP of continents and countries [World Bank 2022]. A second smaller dataset contains models of cars selected by the author in a three-level hierarchy, starting with nationality, then manufacturer, and finishing with car model. There are 100 nodes in total in the tree. The species hierarchy of primates is based on the open tree of life project [OpenTree 2022]. This dataset is a deep largely binary tree, which is not an ideal structure to be visualized with a Voronoi treemap. It contains 1071 nodes with a maximum depth of 20. A medium-sized dataset, containing 1861 nodes in three levels, is based on the USP's drug classification [USP 2021]. Here, different types of drugs are classified by their active substances. This dataset is also used in Figure 5.6. One large dataset, containing 3506 nodes in two levels, is composed of Wikipedia editors by nationality [Wikipedia 2022]. A second large dataset, containing 5595 nodes in up to six levels, is Google's product taxonomy [Google 2021].

Multiple options and functions are offered in the Settings menu. It is possible to choose from three color schemes: aquamarine, rainbow and informative. Aquamarine is the default color scheme and is shown in Figure 5.6. The rainbow color scheme can be seen in both Figures 5.1 and 5.2. The informative color scheme draws basic Voronoi regions in a gray tone, acting as a neutral color scheme. Furthermore, when using a folder dataset, all files are colored in respect to their file type. It is also possible to set the font size to be static across the entire layer or to adjust dynamically to the respective polygon size. The font size in Figure 5.1 is set to dynamic and in Figure 5.6 it is set to static.

There are two structural options. Cell placement can be defined to be either static or dynamic. When static, the visualization remains the same with the same data, but when set to dynamic, reloading the same data initializes it with a new seed and produces a new visualization every time. If the input data permits it, it is also possible to choose the attribute which defines the weight of a polygon and hence its resulting size. If the dataset offers multiple numerical attributes, the same hierarchy can be visualized with differently weighted representations.

To improve usability there is a breadcrumb option. When activated, a breadcrumb trail is displayed beneath the menu bar. All names in the breadcrumb trail can be clicked to navigate back to that node in

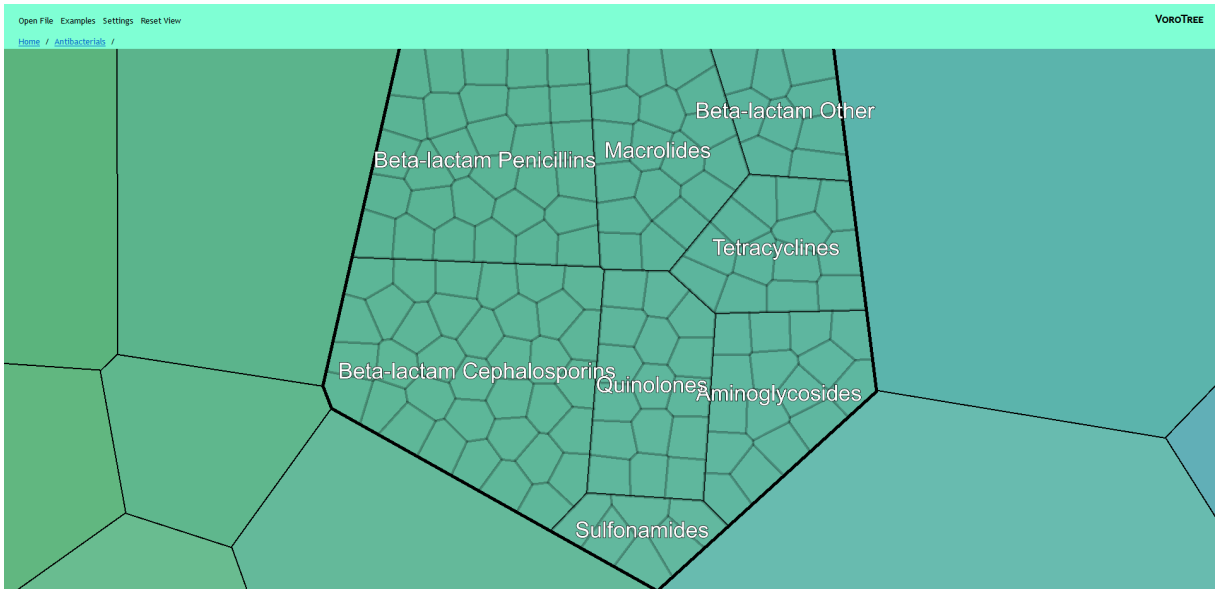

Figure 5.6: VoroTree zoomed in to part of a US drug classification hierarchy. The aquamarine color scheme is used and breadcrumbs are activated. [Screenshot taken by the author using VoroTree.]

the tree. A breadcrumbs trail can be seen in Figure 5.6.

At any point, an SVG export can be triggered from the Settings menu. This will create an SVG file of the current state of the visualization. When zoomed in to a subtree of the data, it will draw the currently active polygon and its children. This has the advantage of being able to produce a scalable export of the visualization at no extra effort.

The VoroTree source code repository also includes a utility function implemented through gulp to create a VoroTree compatible dataset from a given folder in a file system. VoroTree can then visualize this folder, including different icons for different file types.

Chapter 6

VoroLib

This chapter presents VoroLib, a JavaScript library for visualizing interactive Voronoi treemaps [Oser 2022b]. VoroLib provides an API which can be used by web applications to integrate Voronoi treemap functionality.

VoroLib was extracted from VoroTree and shares its Model-View-Controller paradigm and the `Model`, `View`, and `Controller` classes. The only architectural difference between the two, is the fourth class. In VoroTree, the fourth class is the `App` class, which calls the constructors of the other classes and initializes the application. In VoroLib, the fourth class is the VoroTree class, which holds all the API methods exposed to users of the library. To start a VoroLib visualization, the VoroTree constructor must be called. Once a VoroTree object is initialized, it holds all methods used to customize and administrate the visualization.

6.1 Project Infrastructure

Like VoroTree, VoroLib is built using Node.js [OpenJS 2022b] and its package manager npm [npm 2022] to maintain and administrate all external dependencies required to build the library. Development tasks are automated by the task runner gulp [Bublitz and Schoffstall 2022].

The library is written in TypeScript [Microsoft 2022] for its advantages through types and classes. Compilation to JavaScript and the bundling of all dependencies and TypeScript classes are handled by webpack [Koppers et al. 2022]. All these technologies are described in Chapter 2.

6.2 Dependencies

VoroLib shares the same dependencies as VoroTree, described in Section 5.3, with the exception of Electron, since VoroLib is built as a library rather than a self-contained application.

6.3 Purpose and Capabilities

VoroLib is a JavaScript library intended for developers. It is not a finished web application, but is meant to be integrated into one. Developers can embed an interactive Voronoi treemap into their web project by including VoroLib into the source code. It takes very little effort to set it up, if appropriate input data (hierarchy) is available. See Appendix C for more details.

Input data can be given in both JSON and CSV format. It is possible to define a path to a file or to load the input through the HTML input file element. The exact form of the data is described in Appendix C. VoroLib requires certain attributes to be present in the hierarchy, but more attributes can always be added.

If more attributes are present in the data the attribute used for weighting can be changed to any numerical values.

There are various ways of customizing the Voronoi treemap. Labels can be sized according to the area of their polygon or kept at a static value throughout the layer. Colors can be adjusted to any single color or also color scales. A color scheme is defined by a series of color hex values, which is then adjusted by VoroLib to be represented in the visualization. It is also possible to define whether the placement of polygons should be dynamic or static. Depending on which option is chosen, the positions of polygons changes or remains the same on reloads.

VoroLib includes simple administrative procedures for the visualization. At any point it is possible to reset the view to the top level. It is also possible to reload and recalculate the visualization. In case customization option were changed, it is possible to redraw the current state of the visualization. For improved portability and responsiveness, the visualization can be resized to any desired dimensions, which recalculates the Voronoi treemap for the desired size.

It is possible to interact with the data structure used by VoroLib. The hierarchy structure starts at the root polygon and can be navigated via parent and child polygon. Each polygon holds all relevant knowledge like Voronoi vertices, center point, color, weight and name. Polygons can be assigned a callback function via a method in VoroLib. Through these callback functions, developers can define procedures to be triggered when polygons are clicked.

Chapter 7

Outlook and Future Work

Previously the only readily available web application for interactive Voronoi treemaps was FoamTree by Carrot Search [Carrot Search 2021]. Being a commercial software product, it requires a fee to use free of branding, which is not ideal for all use cases. With the development of VoroTree and especially VoroLib, there is now a free, open source alternative to FoamTree.

The work presented here on VoroTree and VoroLib could inspire further work on interactive Voronoi treemap implementations and applications, given its availability in full source code. It would also be possible to integrate interactive Voronoi treemaps into a suite of synchronized visualizations with numerous other types of visualization.

In its current form, VoroTree is a self-contained application, and does not build directly on top of VoroLib, although it could easily be rewritten to do so. In the future, this should be done, so that VoroTree can serve as an example of how to use VoroLib.

Something both software products lack is a mechanism for the end user to specify the visual encoding for color of the Voronoi cells. Provision for binning and selection of color scales could be provided in a configuration panel, like the one available in the original Treemap software described in Section 4.4.3. Furthermore, VoroTree's interface to specify the weight attribute for size encoding is somewhat rudimentary.

Options could be introduced to control the placement of sites based on some criteria, perhaps by providing similarity values between pairs of child nodes to steer a force-directed placement of the sites, in a way similar to InfoSky's approach described in Section 4.4.1.

Options could also be introduced with respect to how leaf nodes are handled, for example by collecting documents at each level into a synthetic cell like InfoSky's approach described in Section 4.4.2, rather than creating a Voronoi cell for each leaf node.

VoroTree introduced a new form of navigation by simply zooming, whereby cells open up or close automatically when they reach or fall below a certain proportion of the display space. It would be useful to conduct a usability study of this and the other forms of navigation (for example by explicit selection, via the breadcrumbs, or via a synchronized outline view) to assess how they perform relative to one another.

Finally, implementation of some form of text search could also be very beneficial for end users, especially when exploring large hierarchical structures.

Chapter 8

Concluding Remarks

This thesis presented the VoroTree application and the VoroLib library for interactive Voronoi treemaps. Both projects are open source. The first part of the thesis, in Chapters 2, 3, and 4 described modern web technologies, the field of information visualization, and the development of Voronoi diagrams and Voronoi treemaps.

The second part of the thesis presented VoroTree and VoroLib. VoroTree, described in Chapter 5, is a self-contained web application for exploring hierarchical data with Voronoi treemaps. VoroLib, described in Chapter 6, is a JavaScript library for integrating Voronoi treemaps into other applications. Finally, Chapter 7 presented some ideas for future work. The four appendices to the thesis provide guides for various pieces of software.

In conclusion, this thesis has produced two software products, both of which are open source and can be used as a starting point for future work. It also reviewed interactive Voronoi treemaps and can serve as a reference for Voronoi web projects. The VoroTree source code is available on GitHub [Oser 2022c]. The VoroTree application is also available as a hosted demo application on the web [Oser 2022d], and can be built as an installable native package for common desktop platforms. The VoroLib library is available in source code on GitHub [Oser 2022b].

Appendix A

VoroTree User Guide

This appendix serves as a user guide for VoroTree. The initial state of VoroTree is shown in Figure A.1. The application always starts without any loaded data, to give the user the option to choose the dataset.

A.1 Web Application

The simplest way to experience VoroTree is by using the hosted web application at https://somestudentcoder.github.io/vorotree/ [Oser 2022d].

A.2 Local Installation

To install locally on a desktop, the VoroTree application can be downloaded from https://github.com/somestudentcoder/vorotree [Oser 2022c]. Standalone executable packages for Windows, Mac, and Linux are listed under the latest release. Download the appropriate package to use VoroTree locally and offline.

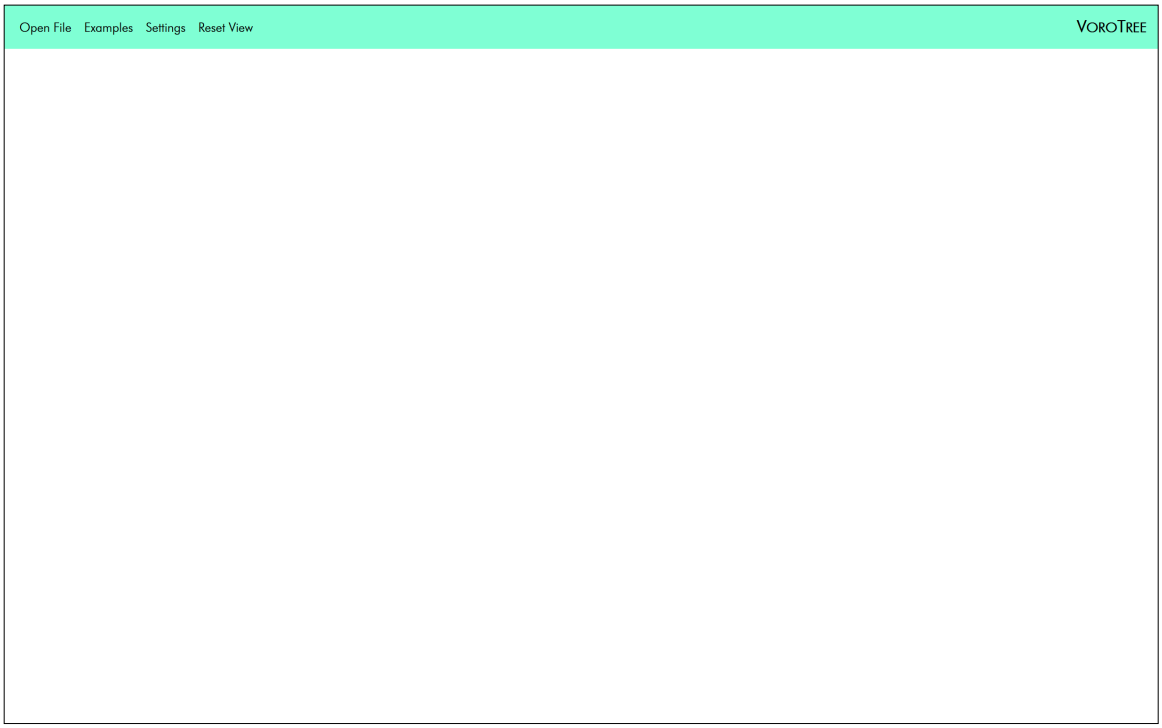

Figure A.1: VoroTree after starting the application. No data is initially loaded. [Screenshot made by Keith Andrews using VoroTree. Used with kind permission.]

A.3 Features and Usage

All VoroTree features can be accessed from the main menu bar, which can be seen in Figure A.1. The Open File button opens the web browser's file browser to select an input file.

The Examples dropdown menu lists all available example datasets, and is shown in Figure A.2. The datasets are described in detail in Section A.4. Each dataset can be selected via click or tap to present the user with different input data. Figure A.3 shows VoroTree with the Google products taxonomy dataset.

The Settings dropdown menu provides access to various visualization settings and to the SVG Export feature, as shown in Figure A.4. The breadcrumb bar can be turned on or off. Breadcrumbs compactly show the path through the hierarchy from the root to the current node, and can be used to jump to any node along the path. Three color schemes (aquamarine, rainbow, and informative) are available for selection. The informative color scheme encodes cells by folder and type: folders are gray and files are colored by their extension. Figure A.5 shows VoroTree with part of a file system hierarchy, breadcrumbs active, and the informative color scheme.

Cell placement can be set to static or dynamic. Static cell placement retains the same seed and produces the same layout every time. The font size can also be set to static or dynamic. Static font sizing uses the same font size for all cell labels, dynamic font sizing adjusts the font size to the polygon size. If the dataset includes quantitative attributes which can be used as the weight to encode cell size, the desired attribute can be chosen under Weighted attribute in the Settings menu. If no other weight attribute is specified, the number of children is used. Finally, the Reset View button resets the visualization to its initial state at the root of the tree. This enables users to start over when getting lost in the visualization.

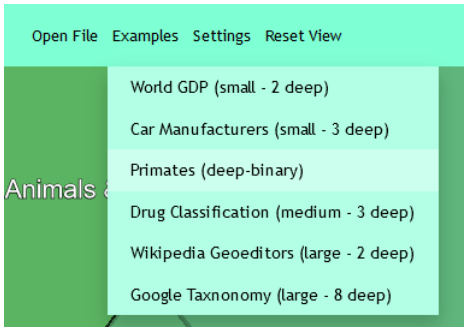

Figure A.2: VoroTree example menu used to load predefined datasets. [Screenshot made by the author using VoroTree.]

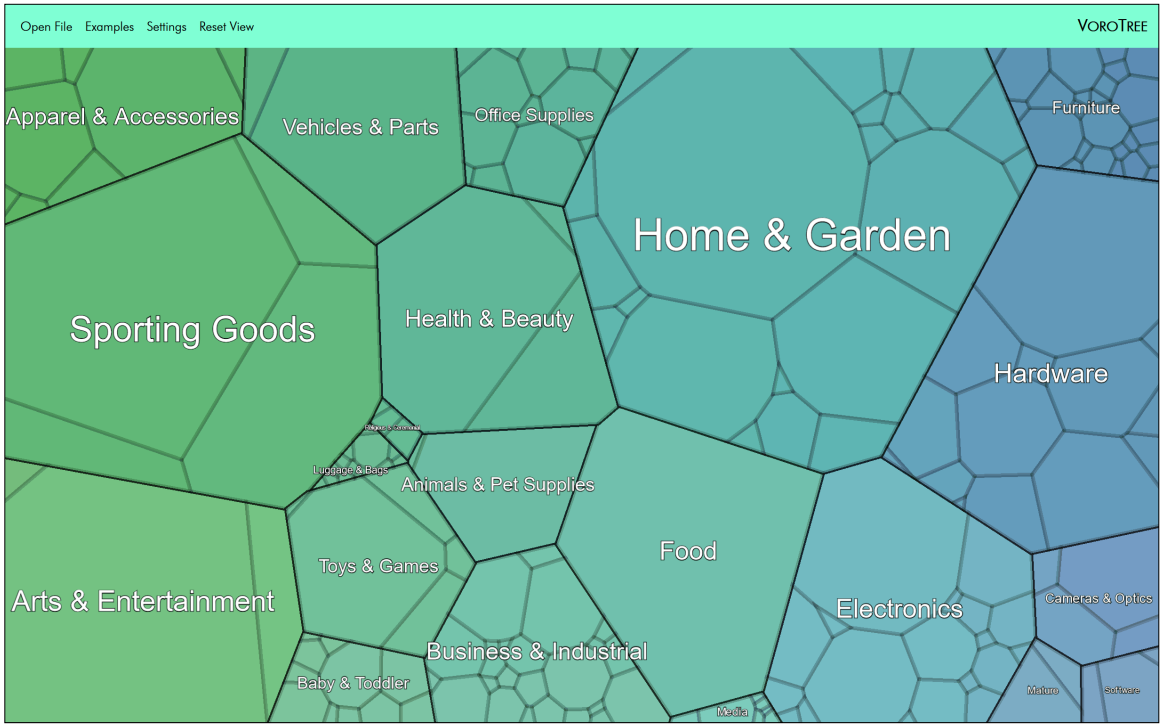

Figure A.3: VoroTree with the Google products taxonomy dataset and the default aquamarine color scheme. [Screenshot made by Keith Andrews using VoroTree. Used with kind permission.]

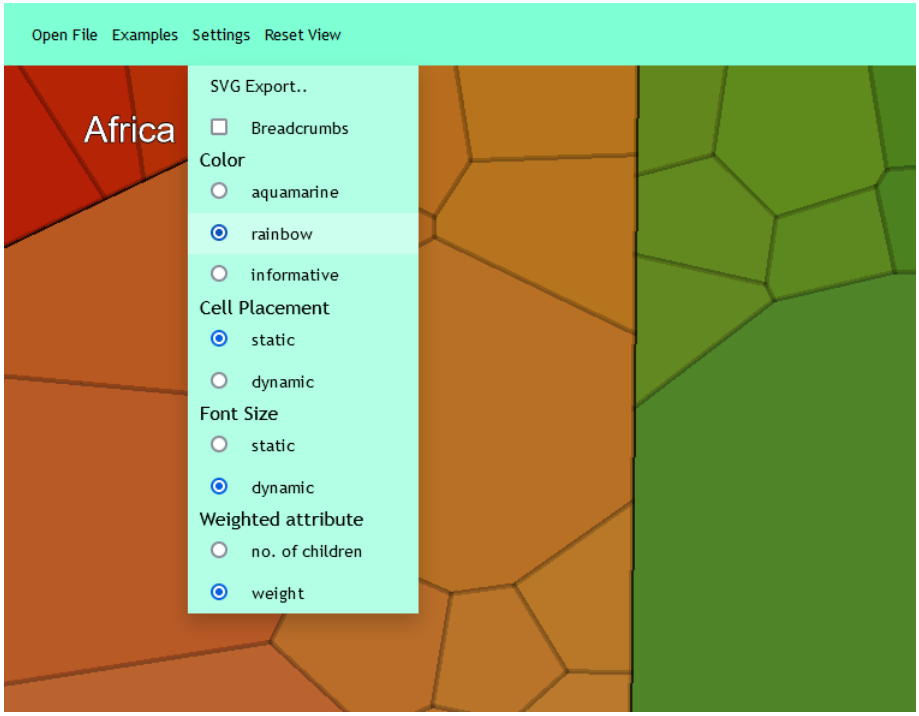

Figure A.4: VoroTree options are listed in the Settings menu. This includes visual and structural changes to the visualization. [Screenshot made by the author using VoroTree.]

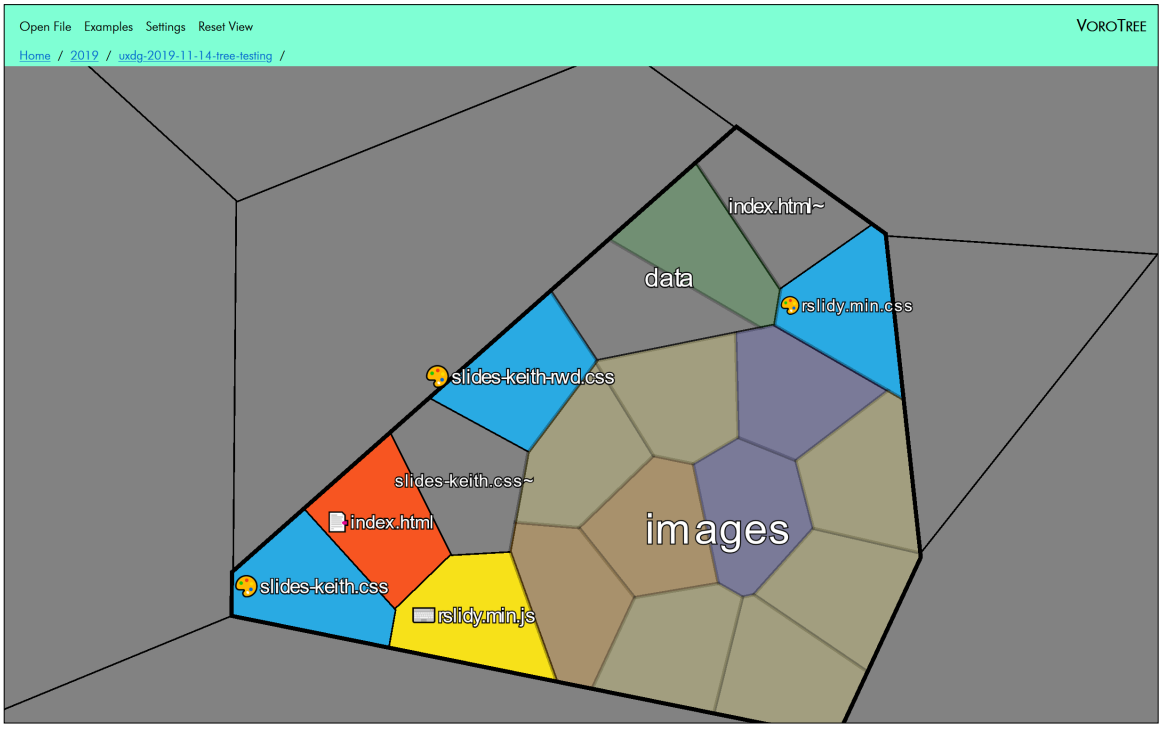

Figure A.5: VoroTree showing part of a file system hierarchy with breadcrumbs active and the informative color scheme. [Screenshot made by Keith Andrews using VoroTree. Used with kind permission.]

A.4 Example Data

Six datasets are included in the VoroTree application for demonstration purposes:

- A highly edited version of the Country, Regional, and World GDP (Gross Domestic Product) dataset [World Bank 2022]. Only some countries were selected to produce a smaller dataset for demonstration purposes. It contains 50 nodes with a constant depth of two.
- Custom car manufacturer dataset constructed by the author. It spans over a selection of car manufacturers divided by country of origin and select models of the brand. It contains 100 nodes with a constant depth of three.
- Adapted primate dataset, originally taken from the Open Tree of Life project [OpenTree 2022]. It represents the tree of species of Primates and is a largely binary tree, containing 1071 nodes with a maximum depth of 20.
- A drug classification scheme from the USA which categorizes drugs according to substances and uses. The dataset is provided by the USP [USP 2021]. It contains 1861 nodes with a constant depth of three.
- The nationalities of Wikipedia page editors. Editors are grouped by nationality per Wikipedia page language. Larger entries represent 100 or more editors, while smaller ones represent between 5 and 99 editors [Wikipedia 2022]. It contains 3506 nodes with a constant depth of two.
- Google product taxonomy, a list of categories used by Google to help departmentalize products in a shopping feed [Google 2021]. It contains 5595 nodes with a maximum depth of six.

A.5 Data Files

VoroTree expects input datasets to be structured in a certain way. Both CSV and JSON files are supported. CSV data files need to have the following format in order to visualize the data properly:

- *name*: Identifies a node and is used as its label.
- *parent*: Identifies the parent of a node.
- *weight*: Influences the size of the polygon. Only leaf nodes should have this property. If the weight is not given, the size of the polygon depends on the amount of children.

Any other numerical attributes of a node are automatically added to the Settings menu as a weight attribute. So any other numerical attributes in the data can also be used to weight the cells in the visualization.

Listing A.1 and Table A.1 show a hierarchical dataset of cars in CSV format. The parent column identifies the name of that node's parent. The weight column specifies a weight for each leaf node. Listing A.2 and Table A.2 show a hierarchical dataset of parent-child relationships, where the name column is not unique and so a specific ID column is necessary. An example of a JSON dataset is shown in Listing A.3.

```
name,parent,weight
cars,,
owned,cars,
traded,cars,
learned,cars,
pilot,owned,40
325ci,owned,40
accord,owned,20
chevette,traded,10
odyssey,learned,20
maxima,learned,10
```

Listing A.1: A hierarchical dataset of cars in CSV format. The names in the name column are unique and are referred to in the parent column.

name	parent	weight
cars		
owned	cars	
traded	cars	
learned	cars	
pilot	owned	40
325ci	owned	40
accord	owned	20
chevette	traded	10
odyssey	learned	20
maxima	learned	10

Table A.1: The dataset from Listing A.1 in tabular form.

```
id,name,parentId,weight
1,Father,,
2,Alice,1,
3,Alice,1,
4,Bob,2,10
5,Doris,3,20
```

Listing A.2: Parent-child relationships where the name column is not unique. A specific id column is required.

id	name	parentId	weight
1	Father		
2	Alice	1	
3	Alice	1	
4	Bob	2	10
5	Doris	3	20

Table A.2: The dataset from Listing A.2 in tabular form.

```
{
"name": "America",
"children": [
{
    "name": "North America",
    "children": [
    {"name": "United States", "weight": 24.32},
    {"name": "Canada", "weight": 2.09},
    {"name": "Mexico", "weight": 1.54}
    ]
},
{
    "name": "South America",
    "children": [
    {"name": "Brazil", "weight": 2.39},
    {"name": "Argentina", "weight": 0.79},
    {"name": "Venezuela", "weight": 0.5},
    {"name": "Colombia", "weight": 0.39}
    ]
}
]
}
```

Listing A.3: A sample JSON dataset for VoroTree.

Appendix B

VoroTree Developer Guide

This appendix serves as a developer guide for VoroTree.

B.1 Project Installation

For local use, the VoroTree project can be set up on a desktop computer. The project is hosted on GitHub Oser [2022c]. To set the project up locally, follow the example here:

```
$> git clone <url> <new folder>
$> cd <new folder>
$> npm install
```

For this to work, Node and TypeScript need to be installed globally.

B.2 Gulp Commands

To maintain the software project, multiple gulp commands are set up for administration. These commands can be viewed in Listing B.1. The commands should be run in a bash shell.

```
# Builds and runs the project in development mode with Hot Reload.

  $> npx gulp serve

# Builds the project in production mode.

  $> npx gulp build

# Runs the project as an Electron application.

  $> npx gulp electron

# Compiles a standalone Electron application for the operating
# system it was called on.
# You can find the output under /standalone in the project directory.

  $> npx gulp buildElectronApp

# Compiles a standalone Electron application for Microsoft Windows x64.
# You can find the output under /standalone in the project directory.

  $> npx gulp buildElectronWinApp

# Compiles a standalone Electron application for MacOS.
# You can find the output under /standalone in the project directory.

  $> npx gulp buildElectronMacApp

# Compiles a standalone Electron application for Linux operating systems.
# You can find the output under /standalone in the project directory.

  $> npx gulp buildElectronLinuxApp

# Compiles standalone Electron applications for all available platforms.
# You can find the output under /standalone in the project directory.

  $> npx gulp buildElectronAllApps

# Cleans the "dist" folder which includes the bundled source code.

  $> npx gulp clean

# Cleans the "dist" folder which includes the bundled source code,
# as well as the "node_modules" folder.

  $> npx gulp cleanAll

# Constructs a .json dataset from a specified folder.
# This folder can then be viewed and navigated within VoroTree.
# Specify input folder and output name in the
# "/data/folderdatasetconfig.js" file.

  $> npx gulp constructFolderDataset
```

Listing B.1: Definition of the VoroTree gulp commands.

B.3 Example Data

Six datasets are included in the VoroTree application for demonstration purposes:

- A highly edited version of the Country, Regional, and World GDP (Gross Domestic Product) dataset [World Bank 2022]. Only some countries were selected to produce a smaller dataset for demonstration purposes. It contains 50 nodes with a constant depth of two.
- Custom car manufacturer dataset constructed by the author. It spans over a selection of car manufacturers divided by country of origin and select models of the brand. It contains 100 nodes with a constant depth of three.
- Adapted primate dataset, originally taken from the Open Tree of Life project [OpenTree 2022]. It represents the tree of species of Primates and is a largely binary tree, containing 1071 nodes with a maximum depth of 20.
- A drug classification scheme from the USA which categorizes drugs according to substances and uses. The dataset is provided by the USP [USP 2021]. It contains 1861 nodes with a constant depth of three.
- The nationalities of Wikipedia page editors. Editors are grouped by nationality per Wikipedia page language. Larger entries represent 100 or more editors, while smaller ones represent between 5 and 99 editors [Wikipedia 2022]. It contains 3506 nodes with a constant depth of two.
- Google product taxonomy, a list of categories used by Google to help departmentalize products in a shopping feed [Google 2021]. It contains 5595 nodes with a maximum depth of six.

B.4 Data Files

VoroTree expects input datasets to be structured in a certain way. Both CSV and JSON files are supported. CSV data files need to have the following format in order to visualize the data properly:

- *name*: Identifies a node and is used as its label.
- *parent*: Identifies the parent of a node.
- *weight*: Influences the size of the polygon. Only leaf nodes should have this property. If the weight is not given, the size of the polygon depends on the amount of children.

Any other numerical attributes can also be used to weight the cells in the visualization.

Listing B.2 and Table B.1 show a hierarchical dataset of cars in CSV format. The parent column identifies the name of that node's parent. The weight column specifies a weight for each leaf node. Listing B.3 and Table B.2 show a hierarchical dataset of parent-child relationships, where the name column is not unique and so a specific ID column is necessary. An example of a JSON dataset is shown in Listing B.4.

```
name,parent,weight
cars,,
owned,cars,
traded,cars,
learned,cars,
pilot,owned,40
325ci,owned,40
accord,owned,20
chevette,traded,10
odyssey,learned,20
maxima,learned,10
```

Listing B.2: A hierarchical dataset of cars in CSV format. The names in the name column are unique and are referred to in the parent column.

name	parent	weight
cars		
owned	cars	
traded	cars	
learned	cars	
pilot	owned	40
325ci	owned	40
accord	owned	20
chevette	traded	10
odyssey	learned	20
maxima	learned	10

Table B.1: The dataset from Listing B.2 in tabular form.

```
id,name,parentId,weight
1,Father,,
2,Alice,1,
3,Alice,1,
4,Bob,2,10
5,Doris,3,20
```

Listing B.3: Parent-child relationships where the name column is not unique. A specific id column is required.

id	name	parentId	weight
1	Father		
2	Alice	1	
3	Alice	1	
4	Bob	2	10
5	Doris	3	20

Table B.2: The dataset from Listing B.3 in tabular form.

```
{
"name": "America",
"children": [
{
    "name": "North America",
    "children": [
    {"name": "United States", "weight": 24.32},
    {"name": "Canada", "weight": 2.09},
    {"name": "Mexico", "weight": 1.54}
    ]
},
{
    "name": "South America",
    "children": [
    {"name": "Brazil", "weight": 2.39},
    {"name": "Argentina", "weight": 0.79},
    {"name": "Venezuela", "weight": 0.5},
    {"name": "Colombia", "weight": 0.39}
    ]
}
]
}
```

Listing B.4: A sample JSON dataset for VoroTree.

B.5 Constructing a Folder Dataset

It is possible to construct a dataset automatically from a specified folder via a gulp command. First specify the desired folder and file name in the configuration file `data/folderdatasetconfig.js`. To construct the dataset, execute the gulp command `constructFolderDataset`, as described in Listing B.1.

Appendix C

VoroLib Developer Guide

This appendix serves as a developer guide for the VoroLib library.

C.1 Quick Start

VoroLib can swiftly be added to any web page or application. An example can be found in the `docs/` folder of the GitHub repository Oser [2022b]. Please note that the web page must be hosted on an HTTP server, otherwise Cross-Origin protocols will stop VoroLib from loading. To run a VoroLib visualization on a web page the following is required:

- An HTML page which includes `vorolib.js`.
- A CSV or JSON file with data in the correct format (explained below).
- An HTML element to hold the visualization, i.e. a `<div>`.

The most basic form of integrating VoroLib is shown in Listing C.1. All the methods available in VoroLib are described with examples in Listings C.2, C.3, and C.4.

```
// By calling the VoroTree constructor the visualization is initialized:

var myVis = new VoroLib.VoroTree(<string>[data-path],
  <string>[id of HTMLElement], <number>[width], <number>[height])

// As can be viewed in the example call in the repository,
// an example call could look like:

var myVis = new VoroLib.VoroTree("cars.csv", "visualization", 500, 500)

// Produces a standard VoroLib visualization within
// the HTMLElement specified. To further customize, use the
// VoroTree methods listed below.
```

Listing C.1: This illustrates the simplest form of including a VoroLib visualization in a web page.

```
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Opening a File:

myVis.openFile(<any> file);

// Opens the specified file taken from an HTML input object.
// The data will then be loaded into the visualization.
//Example:

const inputObject = <HTMLInputElement> document.getElementById("input");
inputObject.addEventListener("change", function(){
myVis.openFile(inputObject.files);
})

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//Load .CSV from path

myVis.loadCSVFile(<string> file-path);

// Opens the specified .csv file at the specified path.
// The data will then be loaded into the visualization.
// Example:

myVis.loadCSVFile("data.csv");

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//Load .JSON from path

myVis.loadJSONFile(<string> file-path);

// Opens the specified .json file at the specified path.
// The data will then be loaded into the visualization.
// Example:

myVis.loadJSONFile("data.json");

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Get current view as SVG

myVis.getSVG();

// Draws and returns a SVG of the current view in the visualization.
// Example:

let svg = myVis.getSVG();

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Export current view as SVG

myVis.exportSVG();

// Draws a SVG of the current view in the visualization and
// prompt the user to open or download it.
```

Listing C.2: Definition of the VoroLib method structure and presentation of examples.

```
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//Change color scheme

myVis.changeColorScheme(<string[]> color-scheme);

// Takes one or more colors in hex to construct a color scale over the 2D space.
// Example:

myVis.changeColorScheme(['#ff0000', '#ffb300', '#3afa00']);

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Change cell placement

myVis.setCellPlacementStatic(<boolean> value);

// Defines if on reloads of the data, the visualization always remains
// the same (static) or if the visualization varies after each reload.
// Default is set to true.
// Example:

myVis.setCellPlacementStatic(false);

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Change font size

myVis.setFontSizeStatic(<boolean> value);

// Defines if font size is relative to cell size or is to remain the same
// over all cells (static). Default is set to false.
// Example:

myVis.setFontSizeStatic(true);

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Change weighted attribute

myVis.changeWeightAttribute(<string> name of weight attribute);

// Defines which attribute in the data is chosen for weighting
// the visualization. Attribute must be present in the data!
// Default is 'weight'. If weight is not specified, the size is
// based on number of children.
// Example:

myVis.changeWeightAttribute('age');

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//Set a callback function

myVis.setCallbackFunction((polygon: Polygon) => void);

// Sets a callback function to Voronoi cells. The function is called when
// Voronoi cells are tapped/clicked. The polygon parameter is the Polygon
// object of the clicked cell.
// Example:

myVis.setCallbackFunction((polygon) => {window.alert(polygon.name)});
```

Listing C.3: Definition of the VoroLib method structure and presentation of examples.

```
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Activate Emoji Icons

myVis.activateFileIcons(<boolean> value);

// When set to true, adds Icons to nodes with file type endings.

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Reset the View

myVis.resetView();

// Sets the visualization to its initial state with the root polygon being in view.

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Redraw visualization

myVis.redraw();

// Redraws the visualization in its current state. Can be useful when
// changing small things and then refreshing without the user noticing.

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Resize visualization

myVis.resize(<number> width, <number> height);

// Resizes the visualization and redraws it to the selected width and height.

// Example:

myVis.resize(800, 800);

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Get the data in VoroLib format

myVis.getData();

// Returns the root Polygon of the visualization, which includes the entire
// data in VoroLib format. For PIXI methods and members please consult the
// PixiJS API at https://pixijs.download/dev/docs/PIXI.Graphics.html.
// Data structure:

class Polygon extends PIXI.Graphics{
    public center: Point                    // X and Y coordinates of the center
                                            // of the polygon.
    public points: Array<Point>             // The points the polygon is made up of.
    public polygon_children: Array<Polygon> // All children of polygon.
    public polygon_parent: Polygon          // The parent of polygon.
    public color: number[]                  // Color array. The color at color[0]
                                            // is the one that is used for this
                                            // polygon.
    public id: number                       // ID of polygon.
    public name: string                     // Name of polygon.
    public path: string                     // Path of polygon.
    public callbackFunction: Function       // The callback function of polygon.
    public functionFlag: boolean            // Boolean that indicates whether
                                            // cb function is active.
}
```

Listing C.4: Definition of the VoroLib method structure and presentation of examples.

```
# The following command runs a HTTP server under /docs to demonstrate
# the example locally.

  $> npx gulp serve

# The following command builds VoroLib to the dist folder.

  $> npx gulp build

# The following command cleans the "dist" folder,
# which contains the bundled source code.

  $> npx gulp clean

# The following command cleans the "dist" folder,
# which includes the bundled source code,
# as well as the "node_modules" folder.

  $> npx gulp cleanAll

# The following command constructs a .json dataset from a specified folder.
# This folder can then be viewed and navigated within the visualization.
# Specify input folder and output name in the "/data/folderdatasetconfig.js" file.

  $> npx gulp constructFolderDataset
```

Listing C.5: Definition of the VoroLib gulp commands.

C.2 Gulp Commands

The VoroLib project includes numerous gulp commands which can be used to administrate the project. These commands are shown in Listing C.5. The commands should be run in a bash shell.

C.3 Data Files

VoroLib expects input datasets to be structured in a certain way. Both CSV and JSON files are supported. CSV data files need to have the following format in order to visualize the data properly:

- *name*: Identifies a node and is used as its label.
- *parent*: Identifies the parent of a node.
- *weight*: Influences the size of the polygon. Only leaf nodes should have this property. If the weight is not given, the size of the polygon depends on the amount of children.

Any other numerical attributes can also be used to weight the cells in the visualization.

Listing C.6 and Table C.1 show a hierarchical dataset of cars in CSV format. The parent column identifies the name of that node's parent. The weight column specifies a weight for each leaf node. Listing C.7 and Table C.2 show a hierarchical dataset of parent-child relationships, where the name column is not unique and so a specific id column is necessary. An example of a JSON dataset is shown in Listing C.8.

```
name,parent,weight
cars,,
owned,cars,
traded,cars,
learned,cars,
pilot,owned,40
325ci,owned,40
accord,owned,20
chevette,traded,10
odyssey,learned,20
maxima,learned,10
```

Listing C.6: A hierarchical dataset of cars in CSV format. The names in the name column are unique and are referred to in the parent column.

name	parent	weight
cars		
owned	cars	
traded	cars	
learned	cars	
pilot	owned	40
325ci	owned	40
accord	owned	20
chevette	traded	10
odyssey	learned	20
maxima	learned	10

Table C.1: The dataset from Listing C.6 in tabular form.

```
id,name,parentId,weight
1,Father,,
2,Alice,1,
3,Alice,1,
4,Bob,2,10
5,Doris,3,20
```

Listing C.7: Parent-child relationships where the name column is not unique. A specific id column is required.

id	name	parentId	weight
1	Father		
2	Alice	1	
3	Alice	1	
4	Bob	2	10
5	Doris	3	20

Table C.2: The dataset from Listing C.7 in tabular form.

```
{
"name": "America",
"children": [
{
    "name": "North America",
    "children": [
    {"name": "United States", "weight": 24.32},
    {"name": "Canada", "weight": 2.09},
    {"name": "Mexico", "weight": 1.54}
    ]
},
{
    "name": "South America",
    "children": [
    {"name": "Brazil", "weight": 2.39},
    {"name": "Argentina", "weight": 0.79},
    {"name": "Venezuela", "weight": 0.5},
    {"name": "Colombia", "weight": 0.39}
    ]
}
]
}
```

Listing C.8: A sample JSON dataset for VoroTree.

Appendix D

Sweep Line Demonstrator

This final appendix presents the Sweep Line Demonstrator an implementation of Fortune's sweep line algorithm, which was developed by the author during the course of this work. The web demo can be viewed at `https://somestudentcoder.github.io/sweepline_demonstrator/` [Oser 2022a]. The source code is available on GitHub [Oser 2021].

D.1 Description and Usage

The Sweep Line Demonstrator shows how Fortune's sweep line algorithm [Fortune 1986] constructs a Voronoi diagram in practice. It is possible to place points (sites) on a plane and then watch the algorithm build the Voronoi diagram step by step. Also, four predefined datasets are included. Execution can be paused and resumed at any point. The demonstrator has no external dependencies and does not require a HTTP server to be executed. Any modern web browser can be used to simply open the HTML file.

The demonstrator can be seen in Figure D.1. It consists of a framed area where Voronoi sites can be placed and some buttons. The Start Algorithm button starts the demonstrator with the desired sites (and then becomes the Pause/Resume button). The Reset button sets the visualization back to its original state. The four sample datasets can be loaded with the buttons Basic, Small, Large, and Degenerate, in the second row of buttons.

The implementation in the demonstrator is written in TypeScript and is largely based on Raymond Hill's excellent JavaScript source code [Hill 2015], which saved a great deal of work. Many thanks to Raymond Hill.

D.2 Installation and Building

The Sweep Line Demonstrator source can be installed and built locally. The steps necessary are shown in Listing D.1. The source code can be downloaded from Oser [2021]. All commands must be run in a bash shell.

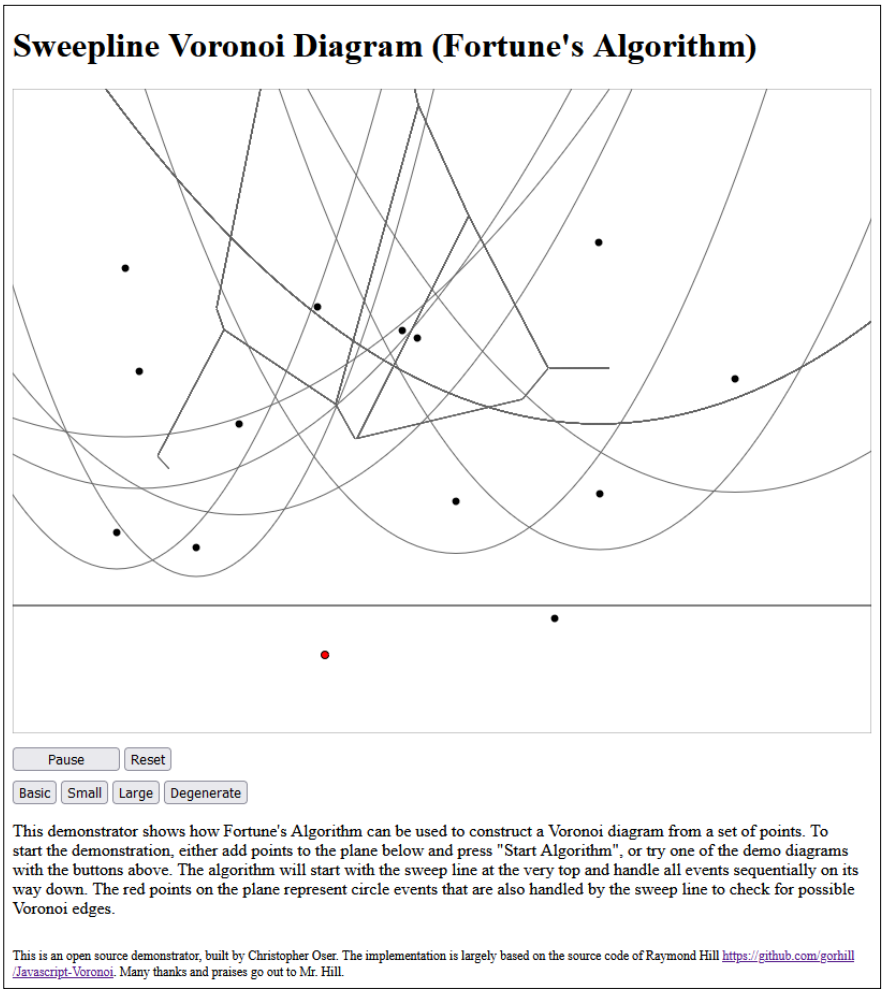

Figure D.1: The Sweep Line Demonstrator illustrates how Fortune's sweep line algorithm [Fortune 1986] to construct a Voronoi diagram works in practice. [Screenshot made by the author using Sweep Line Demonstrator.]

```
# To install the demonstrator from the source code on GitHub,
# execute the following commands:

$> git clone <url> <new folder>
$> cd <new folder>
$> npm install

# The following command builds the project and places the JavaScript
# source code in the /dist folder.

$> npm run build

# The following command builds and watches the TypeScript code
# for changes to recompile.

$> npm run development
```

Listing D.1: A quickstart guide to setting up the Sweep Line Demonstrator locally.

Bibliography

Abel, Guy J. [2021]. *Chord Diagram for Directional Origin-Destination Data*. 02 Dec 2021. https://guyabel.github.io/migest/reference/mig_chord.html (cited on page 10).

Adobe [2022]. *Illustrator*. 02 Feb 2022. https://www.adobe.com/products/illustrator.html (cited on page 6).

Aisch, Gregor [2022]. *chroma.js*. 05 Jan 2022. https://gka.github.io/chroma.js/ (cited on page 39).

Alencar, Aretha [2010]. *CIShell Manual: TreeML (.xml)*. 23 Dec 2010. https://cishell.wiki.cns.iu.edu/2197081.html (cited on page 12).

Andrews, Keith [1996]. *Browsing, Building, and Beholding Cyberspace: New Approaches to the Navigation, Construction, and Visualisation of Hypermedia on the Internet*. PhD Thesis. Graz University of Technology, Austria, Sep 1996. http://ftp.isds.tugraz.at/pub/keith/phd/andrews-1996-phd.pdf (cited on page 13).

Andrews, Keith [2021a]. *Information Visualisation Course Notes*. Graz University of Technology, Austria. 01 Apr 2021. https://courses.isds.tugraz.at/ivis/ivis.pdf (cited on pages 9–10, 13).

Andrews, Keith [2021b]. *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*. Graz University of Technology, Austria. 10 Nov 2021. https://ftp.isds.tugraz.at/pub/keith/thesis/ (cited on page xiii).

Andrews, Keith, Christian Gütl, Josef Moser, Vedran Sabol, and Wilfried Lackner [2001]. *Search Result Visualisation with xFIND*. Proc.2nd International Workshop on User Interfaces to Data Intensive Systems (Zurich, Switzerland). IEEE Computer Society Press, May 2001, pages 50–58. ISBN 0769508340. doi:10.1109/UIDIS.2001.929925 (cited on pages 11–12).

Andrews, Keith, Wolfgang Kienreich, Vedran Sabol, Jutta Becker, Georg Droschl, Frank Kappe, Michael Granitzer, Peter Auer, and Klaus Tochtermann [2002]. *The InfoSky Visual Explorer: Exploiting Hierarchical Structure and Document Similarities*. Information Visualization 1.3/4 (Dec 2002), pages 166–181. doi:10.1057/palgrave.ivs.9500023 (cited on pages 16–17, 23–24, 28–29, 31).

Andrews, Keith, Wolfgang Kienreich, Vedran Sabol, and Michael Granitzer [2004]. *The Visualisation of Large Hierarchical Document Spaces with InfoSky*. Proc. CODATA Prague Workshop on Information Visualization, Presentation, and Design (Prague, Czech Republic). Mar 2004, pages 1–4. http://www.cgg.cvut.cz/infoviz04/papers/andrews.pdf (cited on page 23).

Andrews, Keith, Werner Putz, and Alexander Nussbaumer [2007]. *The Hierarchical Visualisation System (HVS)*. Proc. 11th International Conference on Information Visualisation (IV07) (Zurich, Switzerland). IEEE, 02 Jul 2007, pages 257–262. ISBN 0769529003. doi:10.1109/IV.2007.112. https://ftp.isds.tugraz.at/pub/papers/andrews-iv2007-hvs.pdf (cited on page 14).

Anthes, Gary [2012]. *HTML5 Leads a Web Revolution*. Communications of the ACM 55.7 (Jul 2012), pages 16–17. doi:10.1145/2209249.2209256. https://cacm.acm.org/magazines/2012/7/151236-html5-leads-a-web-revolution/ (cited on page 3).

Aurenhammer, Franz, Rolf Klein, and Der-Tsai Lee [2013]. *Voronoi Diagrams and Delaunay Triangulations*. World Scientific, Aug 2013. ISBN 9814447633. doi:10.1142/8685 (cited on pages 17, 19–20, 22–23).

Balzer, Michael and Oliver Deussen [2005]. *Voronoi Treemaps*. Proc. IEEE Symposium on Information Visualization 2005 (InfoVis 2005) (Minneapolis, Minnesota, USA). Oct 2005, pages 49–56. doi:10.1109/INFVIS.2005.1532128. http://graphics.uni-konstanz.de/publikationen/Balzer2005VoronoiTreemaps/Balzer2005VoronoiTreemaps.pdf (cited on pages xiii, 23–25, 28).

Bau, David [2019]. *seedrandom*. 17 Sep 2019. https://github.com/davidbau/seedrandom (cited on page 39).

Baumgartner, Stefan [2021]. *Advanced TypeScript*. 22 Jul 2021. https://fettblog.eu/advanced-typescript-guide/ (cited on page 5).

Berners-Lee, Tim [1992]. *Re: status. Re: X11 BROWSER for WWW*. 03 Nov 1992. https://lists.w3.org/Archives/Public/www-talk/1991SepOct/0003.html (cited on page 3).

Bierman, Gavin, Mart Abadi, and Mads Torgersen [2014]. *Understanding TypeScript*. European Conference on Object-Oriented Programming (Uppsala, Sweden). Springer, 28 Jul 2014, pages 257–281. ISBN 3662442019. doi:10.1007/978-3-662-44202-9_11. https://users.soe.ucsc.edu/~abadi/Papers/FTS-submitted.pdf (cited on page 5).

BitDegree [2020]. *New CSS3 Features: Learn How This Version Improved CSS2*. 23 Jan 2020. https://bitdegree.org/learn/css3-features (cited on page 4).

Bloom, Zack [2016]. *The Languages Which Almost Became CSS*. Eager, 28 Jun 2016. https://eager.io/blog/the-languages-which-almost-were-css/ (cited on page 4).

Bock, Martin, Amit Kumar Tyagi, Jan-Ulrich Kreft, and Wolfgang Alt [2010]. *Generalized Voronoi Tessellation as a Model of Two-dimensional Cell Tissue Dynamics*. Bulletin of Mathematical Biology 72.7 (Oct 2010), pages 1696–1731. doi:10.1007/s11538-009-9498-3. https://arxiv.org/pdf/0901.4469.pdf (cited on page 17).

Bostock, Michael, Vadim Ogievetsky, and Jeffrey Heer [2011]. *D3rd: Data-Driven Documents*. Proc. IEEE Information Visualization Conference 2011 (InfoVis 2011) (Providence, Rhode Island, USA). IEEE. 23 Oct 2011. doi:10.1109/TVCG.2011.185. https://idl.cs.washington.edu/files/2011-D3-InfoVis.pdf (cited on page 8).

Bostock, Mike [2022]. *d3-hierarchy*. 10 Jan 2022. https://github.com/d3/d3-hierarchy (cited on pages 8, 12, 39).

Bostock, Mike, Jason Davies, Jeffrey Heer, and Vadim Ogievetsky [2022]. *D3.js*. 10 Jan 2022. https://d3js.org/ (cited on page 8).

Bozhanov, Ivan [2022]. *jstree*. 09 Feb 2022. https://jstree.com/ (cited on page 12).

Bublitz, Blaine and Eric Schoffstall [2022]. *gulp*. 09 Jan 2022. https://gulpjs.com/ (cited on pages 7, 36, 43).

Buchheim, Christoph, Michael Jünger, and Sebastian Leipert [2002]. *Improving Walker's Algorithm to Run in Linear Time*. Proc. 10th International Symposium on Graph Drawing (GD 2002) (Irvine, California, USA). Springer LNCS 2528. Aug 2002, pages 344–353. http://www.springerlink.com/content/u73fyc4tlxp3uwt8 (cited on page 13).

Carrot Search [2021]. *FoamTree*. 28 May 2021. https://carrotsearch.com/foamtree/ (cited on pages 25–26, 29, 31, 45).

Cederholm, Dan [2015]. *CSS3 for Web Designers*. 2nd Edition. A Book Apart, 24 Feb 2015. 139 pages. ISBN 1937557200. https://abookapart.com/products/css3-for-web-designers (cited on page 4).

Coyier, Chris [2016]. *Practical SVG*. A Book Apart, 27 Jul 2016. 154 pages. ISBN 1937557421. https://abookapart.com/products/practical-svg (cited on page 5).

Dobrescu-Balaur, Mihnea [2021]. *directory-tree*. 17 Nov 2021. https://github.com/mihneadb/node-directory-tree (cited on page 39).

Dykstra, Phillip [1991]. *XDU*. 1991. https://web.archive.org/web/20090310190801/http://sd.wareonearth.com/~phil/xdu/ (cited on page 16).

ECMA [2021]. *Ecma Standards*. Ecma International, 01 Dec 2021. https://ecma-international.org/publications-and-standards/standards/ (cited on page 5).

Enge, Eric [2021]. *Mobile vs. Desktop Usage in 2020*. Perficient, 23 Mar 2021. https://perficient.com/insights/research-hub/mobile-vs-desktop-usage (cited on page 6).

Fekete, Jean-Daniel and Catherine Plaisant [2003]. *TreeML DTD File*. 2003. www.nomencurator.org/InfoVis2003/download/treeml.dtd (cited on page 12).

Figatner, David [2021]. *pixi-viewport*. 22 Dec 2021. https://github.com/davidfig/pixi-viewport (cited on page 39).

Fortune, Steven [1986]. *A Sweepline Algorithm for Voronoi Diagrams*. Proc. Second Annual Symposium on Computational Geometry (SCG '86) (Yorktown Heights, New York, USA). ACM, 02 Jun 1986, pages 313–322. ISBN 0897911946. doi:10.1145/10515.10549. https://semanticscholar.org/paper/A-sweepline-algorithm-for-Voronoi-diagrams-Fortune/6dae4431abda4c3334995e554d442855076c03a3 (cited on pages 19, 71–72).

Furnas, George W. and Jeff Zacks [1994]. *Multitrees: Enriching and Reusing Hierarchical Structure*. Proc. SIGCHI Conference on Human Factors in Computing Systems (Boston, Massachusetts, USA). New York, NY, USA: Association for Computing Machinery, 24 Apr 1994, pages 330–336. ISBN 0897916506. doi:10.1145/191666.191778. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.1789&rep=rep1&type=pdf (cited on page 12).

GeoGebra [2021]. *GeoGebra*. 28 May 2021. https://geogebra.org/ (cited on pages 18–19, 21–22).

Glatzhofer, Michael [2021]. *D3-Hypertree*. Graz University of Technology, 15 Mar 2021. https://github.com/glouwa/d3-hypertree (cited on page 13).

Google [2021]. *Google Product Taxonomy*. 21 Sep 2021. https://google.com/basepages/producttype/taxonomy.en-US.txt (cited on pages 41, 53, 59).

Granitzer, Michael, Wolfgang Kienreich, Vedran Sabol, Keith Andrews, and Werner Klieber [2004]. *Evaluating a System for Interactive Exploration of Large, Hierarchically Structured Document Repositories*. Proc. IEEE Symposium on Information Visualization (InfoVis 2004) (Austin, Texas, USA). Oct 2004, pages 127–134. doi:10.1109/INFOVIS.2004.19 (cited on page 23).

Groves, Mat [2022]. *PixiJS*. 10 Jan 2022. https://pixijs.com/ (cited on pages 8, 26, 39).

Hill, Raymond [2015]. *Javascript-Voronoi*. 23 Apr 2015. https://github.com/gorhill/Javascript-Voronoi (cited on page 71).

Inkscape's Contributors [2022]. *Inkscape*. The Inkscape Project, 02 Feb 2022. https://inkscape.org/ (cited on page 6).

Kappe, Frank, Georg Droschl, Wolfgang Kienreich, Vedran Sabol, Jutta Becker, Keith Andrews, Michael Granitzer, Klaus Tochtermann, and Peter Auer [2003]. *InfoSky: Visual Exploration of Large Hierar-*

chical Document Repositories. Proc. HCI International 2003 (Crete, Greece). Volume 3. Lawrence Erlbaum Associates, Jun 2003, pages 1268–1272. ISBN 0805849327 (cited on page 23).

Keith, Jeremy [2015]. *Enhance!* WebExpo Prague 2015. 20 Sep 2015. https://slideslive.com/38894415/enhance (cited on page 6).

Keith, Jeremy [2016]. *Resilient Web Design*. Dec 2016. https://resilientwebdesign.com/ (cited on page 6).

Keith, Jeremy and Rachel Andrew [2016]. *HTML5 for Web Designers*. 2nd Edition. A Book Apart, 17 Feb 2016. 92 pages. ISBN 1937557243. https://abookapart.com/products/html5-for-web-designers (cited on page 3).

Khronos [2022a]. *Low-Level 3D Graphics API Based On OpenGL ES*. The Khronos Group, 05 Feb 2022. https://khronos.org/webgl/ (cited on page 6).

Khronos [2022b]. *OpenGL ES Overview*. The Khronos Group, 05 Feb 2022. https://khronos.org/opengles/ (cited on page 6).

Koppers, Tobias, Sean Larkin, Johannes Ewald, Juho Vepsäläinen, and Kees Kluskens [2022]. *Webpack*. 09 Jan 2022. https://webpack.js.org/ (cited on pages 7, 43).

LeBeau, Franck [2022]. *d3-voronoi-treemap*. 10 Jan 2022. https://github.com/Kcnarf/d3-voronoi-treemap (cited on pages 8, 25–26, 39).

Li, Hui, Kang Li, Taehyong Kim, Aidong Zhang, and Murali Ramanathan [2012]. *Spatial Modeling of Bone Microarchitecture*. Three-Dimensional Image Processing (3DIP) and Applications II (SPIE 2012) (Burlingame, California, United States). Edited by Atilla M. Baskurt and Robert Sitnik. 8290. International Society for Optics and Photonics. SPIE, 30 Jan 2012, pages 232–240. doi:10.1117/12.907371. https://cse.buffalo.edu/DBGROUP/bioinformatics/papers/Hui_SPIE2012.pdf (cited on page 17).

Lie, Håkon Wium [1994]. *Cascading HTML style sheets – a proposal*. 10 Oct 1994. https://w3.org/People/howcome/p/cascade.html (cited on page 4).

Lindley, Cody [2013]. *DOM Enlightenment: Exploring JavaScript and the Modern DOM*. O'Reilly, 19 Mar 2013. 180 pages. ISBN 1449342841. http://domenlightenment.com/ (cited on page 5).

Macrofocus [2008]. *InfoScope*. 22 Feb 2008. https://macrofocus.com/installers/infoscope/pad_file.htm (cited on page 11).

Mapbox [2022]. *Delaunator*. 10 Jan 2022. https://github.com/mapbox/delaunator (cited on page 8).

Marcotte, Ethan [2010]. *Responsive Web Design*. A List Apart, 25 May 2010. https://alistapart.com/article/responsive-web-design/ (cited on page 6).

Marcotte, Ethan [2014]. *Responsive Web Design*. 2nd Edition. A Book Apart, 02 Dec 2014. 153 pages. ISBN 1937557189. http://abookapart.com/products/responsive-web-design (cited on page 6).

Marquis, Mat [2016]. *JavaScript for Web Designers*. A Book Apart, 28 Sep 2016. 135 pages. ISBN 1937557464. https://abookapart.com/products/javascript-for-web-designers (cited on page 5).

MDN [2022a]. *Canvas API*. 05 Feb 2022. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API (cited on page 6).

MDN [2022b]. *WebGL*. 05 Feb 2022. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API (cited on page 6).

Microsoft [2022]. *TypeScript*. 09 Jan 2022. https://typescriptlang.org/ (cited on pages 5, 43).

Munzner, Tamara [2014]. *Visualization Analysis and Design*. CRC press, 26 Nov 2014. ISBN 1466508914 (cited on page 11).

Nation, David A., Catherine Plaisant, Gary Marchionini, and Anita Komlodi [1997]. *Visualizing Websites using a Hierarchical Table of Contents Browser: WebTOC*. Proc. 3rd Conference on Human Factors and the Web (Denver, Colorado, USA). US WEST. 12 Jun 1997. doi:10.1016/B978-155860915-0/50026-3. https://cs.umd.edu/hcil/trs/97-10/97-10.html (cited on page 13).

Netscape [1995]. *Netscape and Sun announce JavaScript*. 04 Dec 1995. https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html (cited on page 5).

Nocaj, Arlind and Ulrik Brandes [2012]. *Computing Voronoi Treemaps: Faster, Simpler, and Resolution-independent*. Computer Graphics Forum. Volume 31. Wiley, 25 Jun 2012, pages 855–864. doi:10.1111/j.1467-8659.2012.03078.x. https://d-nb.info/1114662852/34 (cited on pages 8, 25, 28).

npm [2022]. *npm*. npm, 09 Jan 2022. https://npmjs.com/ (cited on pages 7, 36, 43).

Observable [2022]. *d3-delaunay*. 10 Jan 2022. https://github.com/d3/d3-delaunay (cited on pages 8, 39).

Okabe, Atsuyuki, Barry Boots, Kokichi Sugihara, and Sung Nok Chiu [2000]. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. 2nd Edition. Wiley, 2000. ISBN 0471986356. doi:10.1002/9780470317013 (cited on page 17).

OpenJS [2022a]. *Electron*. OpenJS Foundation, 09 Jan 2022. https://electronjs.org/ (cited on pages 8, 33, 39).

OpenJS [2022b]. *Node.js*. OpenJS Foundation, 20 Feb 2022. https://nodejs.org/ (cited on pages 7, 36, 43).

OpenTree [2022]. *Open Tree of Life*. 18 Jan 2022. https://tree.opentreeoflife.org/ (cited on pages 41, 53, 59).

Oser, Christopher [2021]. *Voronoi Diagram Sweep Line Source Code*. Graz University of Technology, 28 Jul 2021. https://github.com/somestudentcoder/sweepline_demonstrator (cited on page 71).

Oser, Christopher [2022a]. *Sweepline Voronoi Diagram (Fortune's Algorithm)*. Graz University of Technology, 20 Jan 2022. https://somestudentcoder.github.io/sweepline_demonstrator/ (cited on page 71).

Oser, Christopher [2022b]. *VoroLib*. Graz University of Technology, 19 Jan 2022. https://github.com/somestudentcoder/vorolib (cited on pages 1, 28–29, 31, 43, 47, 63).

Oser, Christopher [2022c]. *VoroTree*. Graz University of Technology, 20 Jan 2022. https://github.com/somestudentcoder/vorotree/ (cited on pages 1, 47, 49, 57).

Oser, Christopher [2022d]. *VoroTree Demo Application*. Graz University of Technology, 19 Jan 2022. https://somestudentcoder.github.io/vorotree/ (cited on pages 1, 28–29, 31, 33, 47, 49).

Oser, Christopher, Lisa Weissl, Markus Ruplitsch, and Romana Gruber [2021]. *Interactive Voronoi Treemap*. Graz University of Technology, 06 Apr 2021. https://github.com/somestudentcoder/IVT (cited on pages 26–27, 33).

Peyrott, Sebastian [2017]. *A Brief History of JavaScript*. auth0, 16 Jan 2017. https://auth0.com/blog/a-brief-history-of-javascript/ (cited on page 5).

Schulz, Hans-Jörg [2011]. *treevis.net: A Tree Visualization Reference*. IEEE Computer Graphics and Applications 31.6 (20 Oct 2011), pages 11–15. doi:10.1109/MCG.2011.103. https://treevis.net/ (cited on pages 13, 16).

Schulz, Hans-Jörg, Steffen Hadlak, and Heidrun Schumann [2011]. *The Design Space of Implicit Hierarchy Visualization: A Survey*. IEEE Transactions on Visuaiaztion and Computer Graphics 17.4 (Apr 2011), pages 393–411. ISSN 1077-2626. doi:10.1109/TVCG.2010.79 (cited on page 16).

Shneiderman, Ben [1992]. *Tree Visualization with Tree-Maps: 2-d Space-Filling Approach*. Transactions on Graphics 11 (Jan 1992), pages 92–99. doi:10.1145/102377.115768. https://cs.umd.edu/~ben/papers/Shneiderman1992Tree.pdf (cited on page 16).

Shneiderman, Ben [2004]. *Treemap 4.1.1*. Human-Computer Interaction Lab, University of Maryland, 17 Feb 2004. https://cs.umd.edu/hcil/treemap/ (cited on page 16).

Stasko, John and Eugene Zhang [2000]. *Focus+Context Display and Navigation Techniques for Enhancing Radial, Space-Filling Hierarchy Visualizations*. Proc. IEEE Symposium on Information Visualization (InfoVis 2000) (Salt Lake City, Utah, USA). IEEE, Oct 2000, pages 57–65. ISBN 0769508049. doi:10.1109/INFVIS.2000.885091. https://faculty.cc.gatech.edu/~stasko/papers/infovis00.pdf (cited on page 16).

Sud, Avneesh, Danyel Fisher, and Huai-Ping Lee [2010]. *Fast Dynamic Voronoi Treemaps*. 2010 International Symposium on Voronoi Diagrams in Science and Engineering (Quebec City, Quebec, Canada). IEEE. Jun 2010, pages 85–94. doi:10.1109/ISVD.2010.16. https://microsoft.com/en-us/research/wp-content/uploads/2010/04/Fast-Dynamic-Voronoi-Treemaps.pdf (cited on page 25).

USP [2021]. *USP Drug Classification 2021*. The United States Pharmacopeial Convention, 05 Jul 2021. https://www.genome.jp/kegg-bin/get_htext?htext=br08302.keg (cited on pages 41, 53, 59).

Van Hees, Rinse [2014]. *Stable Voronoi treemaps for software system visualization*. Master's Thesis. Utrecht University, The Netherlands, 14 Aug 2014. https://webspace.science.uu.nl/~hage0101/downloads/rinsevanhees-msc.pdf (cited on pages 26, 28).

Van Hees, Rinse and Jurriaan Hage [2015]. *Stable Voronoi-Based Visualizations for Software Quality Monitoring*. IEEE 3rd Working Conference on Software Visualization (VISSOFT 2015) (Bremen, Germany). 27 Sep 2015, pages 6–15. ISBN 146737525X. doi:10.1109/VISSOFT.2015.7332410 (cited on pages xiii, 26–27).

W3C [2004]. *Document Object Model (DOM) Level 3 Core Specification*. W3C Recommendation. 07 Apr 2004. https://w3.org/TR/DOM-Level-3-Core/ (cited on page 5).

W3C [2009]. *SKOS Simple Knowledge Organization System*. World Wide Web Consortium (W3C), 18 Aug 2009. https://w3.org/TR/skos-reference/ (cited on page 12).

W3C [2021a]. *CSS Snapshot*. World Wide Web Consortium (W3C), 31 Dec 2021. https://w3.org/TR/CSS/ (cited on page 4).

W3C [2021b]. *MathML*. World Wide Web Consortium (W3C), 10 Aug 2021. https://w3.org/Math/ (cited on page 3).

W3C [2022]. *SVG*. World Wide Web Consortium (W3C), 10 Jan 2022. https://w3.org/Graphics/SVG/ (cited on pages 3, 5).

W3Schools [2022]. *HTML Canvas Reference*. Refsnes Data, 02 Feb 2022. https://w3schools.com/tags/ref_canvas.asp (cited on page 6).

Walker II, John Q. [1990]. *Node-Positioning Algorithm for General Trees*. Software: Practice and Experience 20.7 (Jul 1990), pages 685–705. doi:10.1002/spe.4380200705. https://www.cs.unc.edu/techreports/89-034.pdf (cited on page 13).

Ward, Matthew O., Georges Grinstein, and Daniel Keim [2015]. *Interactive Data Visualization: Foundations, Techniques, and Applications*. 2nd Edition. CRC press, 29 May 2015. ISBN 1482257378 (cited on page 11).

Ware, Colin [2019]. *Information Visualization: Perception for Design*. 4th Edition. Morgan Kaufmann, 19 Dec 2019. 560 pages. ISBN 0128128763. http://ifs.tuwien.ac.at/~silvia/wien/vu-infovis/articles/book_information-visualization-perception-for-design_Ware_Chapter1.pdf (cited on page 9).

WHATWG [2022a]. *DOM Living Standard*. Web Hypertext Application Technology Working Group (WHATWG). 02 Feb 2022. https://dom.spec.whatwg.org/ (cited on page 5).

WHATWG [2022b]. *HTML - Living Standard*. Web Hypertext Application Technology Working Group, 05 Jan 2022. https://html.spec.whatwg.org/multipage/ (cited on page 3).

Wikipedia [2022]. *Wikipedia Geoeditors*. 18 Jan 2022. https://dumps.wikimedia.org/other/geoeditors/readme.html (cited on pages 41, 53, 59).

World Bank [2022]. *Country, Regional and World GDP (Gross Domestic Product)*. The World Bank Group, 20 Jan 2022. https://datahub.io/core/gdp (cited on pages 41, 53, 59).

Zeng, Zeno [2021]. *SVGCanvas*. 23 Nov 2021. https://github.com/zenozeng/svgcanvas (cited on page 39).