# Interactive Voronoi Tree
# Report

Christopher Oser, Lisa Weissl, Markus Ruplitsch, Romana Gruber

Graz University of Technology
A-8010 Graz, Austria

706.057 Information Visualisation SS 2020
Graz University of Technology

28 June 2020

**Abstract**

In this report the process of creating an interactive Voronoi tree is documented. The application is web-based and is fully open source, in contrast to other available applications.

# Contents

# List of Figures

# List of Listings

# Chapter 1

# Introduction

In general, the visualization of hierarchical data is essential in so many fields, as the data can be better understood and evaluated. There are many solutions to visualize such hierarchical data, and one of them is to do it with a Voronoi treemap. Therefore, our goal was to build an application where we visualize hierarchical data as a Voronoi treemap. Of course, there are already numerous applications, which do more or less the same. But the difference is that many of them are not for free or available as open source. As an example, we had a look on *FoamTree* [2020]. FoamTree is an interactive Voronoi treemap visualizer implemented in javascript, but unfortunately not for free and also not available as open source. The offers a free trial and the possibility to look at some examples, to get a chance to see how powerful this tool is. But for permanent use, you have to purchase a licence. A screenshot of a Voronoi treemap of FoamTree can be seen in figure 1.1.

With FoamTree as a working example, our goal was to implement such a similar application that can be used by everyone. In figure 1.2 you can see our interactive Voronoi treemap application.

In the next chapters, we give a short overview of Voronoi diagrams and Voronoi treemaps, which technologies we used for our implementation, how the data proceeds and what the data has to look like. Furthermore, we go a little bit in detail related to the calculation of an entire weighted Voronoi treemap and how we visualized it. Finally, we discuss our results.

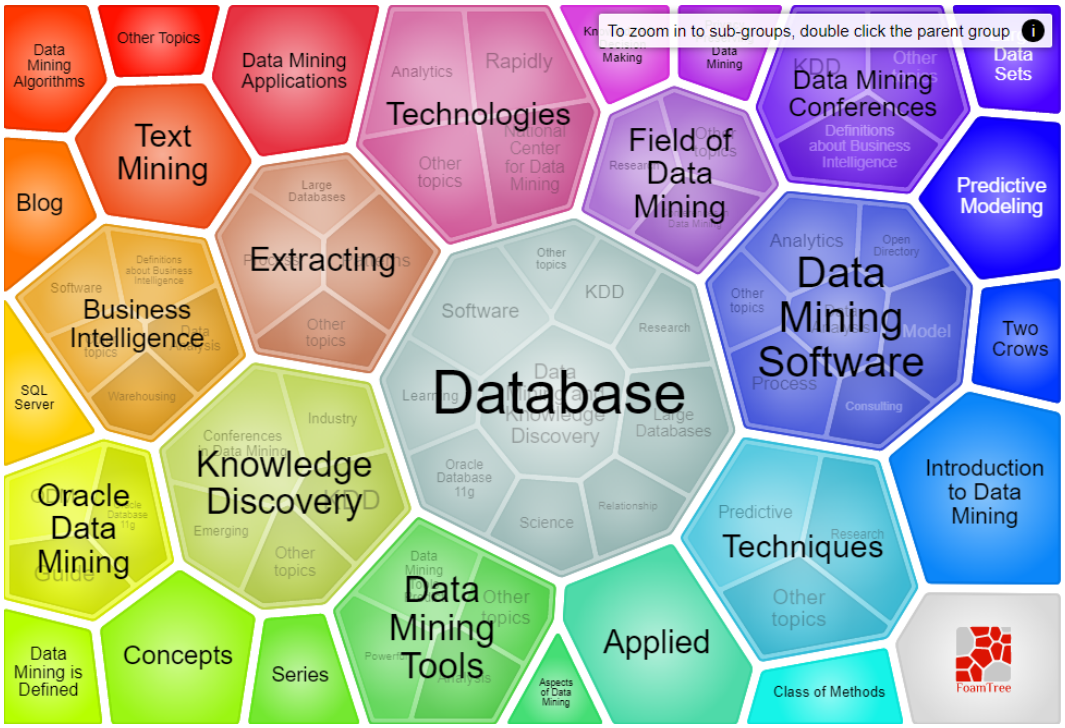


**Figure 1.1:** A visualization of a weighted Voronoi treemap of FoamTree. [Screenshot taken by Romana Gruber.]

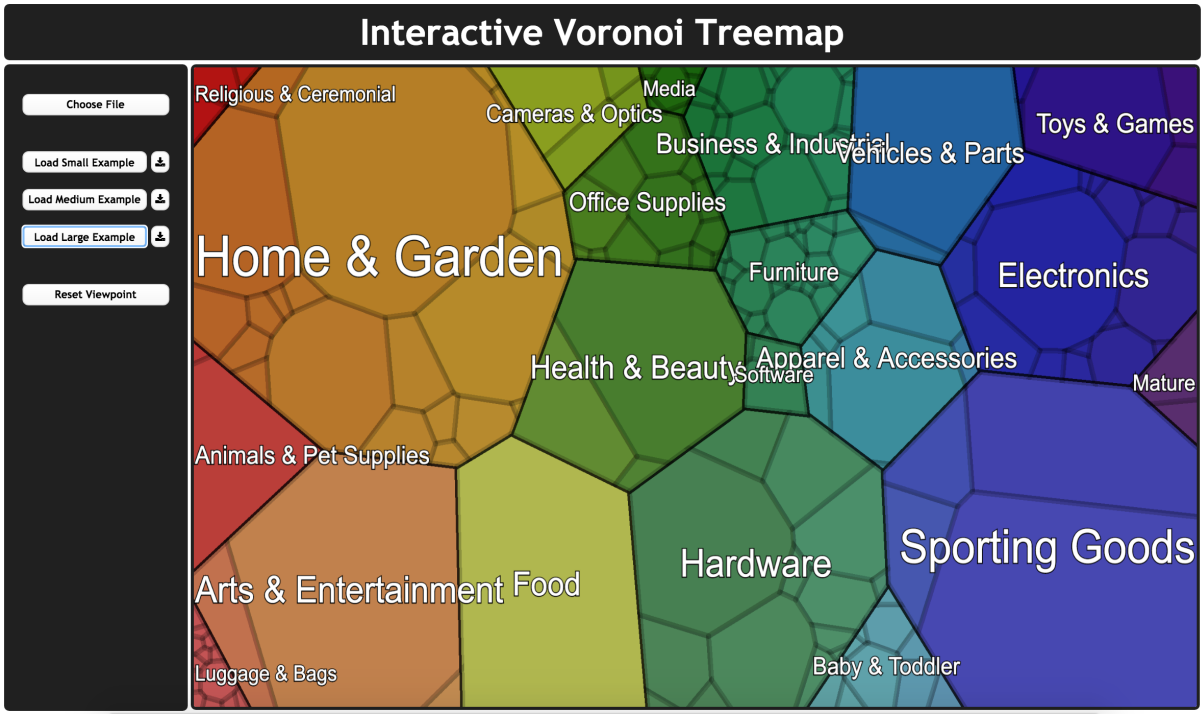


**Figure 1.2:** A visualization of our implemented interactive Voronoi treemap. [Screenshot taken by Romana Gruber.]

# Chapter 2

# Voronoi Theory

This chapter will cover the theory behind voronoi diagrams and voronoi treediagrams. We will briefly explain how to compute them and what characteristics they provide.

## 2.1 Voronoi Diagrams

A voronoi diagram is a subdivision of space into cells that produces no overlaps or holes. The number of cells is defined by the number of input points (also called sites, seeds or generators) and the size and shape of each cell is defined by a distance function. To be more pecise, each cell, with a seed $s$ is defined as the set of all Points $P = \{p_1, p_2, ...\}$, where the distance from the center $d(s, p_i)$ is smaller than the distance to any other seed $d(s_j, p_i)$. While this formulation could be used in higher-dimensional space, for our purpose we are only interested in the two-dimensional case. It is also woth mentioning, that the distance function does not have to be the euclidean distance. Other Distance functions, such as the Manhatten distance or the power weighted distance function, can be used to produce interesting looking patterns, however the euclidean distance is the most common case. Examples for the euclidea distance and the manhatten distance functions being used can be seen in Figure 2.1 and Figure 2.2.

The easiest way to compute a voronoi diagram (using euclidean distances) is by taking the dual graph of the delaunay triangulation. This means that every edge of the delaunay triangulation is flipped 90°and extended to reach the next edge.

Centroidal voronoi diagrams are a special type of voronoi diagram, where the position of a seed is the same as the center of mass of its corresponding cell. By using centroidal voronoi diagrams, we can make sure that the cells do not become too stretched and have similar aspect ratios close to 1:1. Since such a diagram is extremely hard to compute analytically, it is most often created by repeatedly computing voronoi diagrams and moving each seed to the center of its cell.

## 2.2 Voronoi Treemaps

A voronoi treemap is a treemap that uses nested centroidal voronoi diagrams instead of rectangles to visualise hierarchichal data, such that the size of each cell represents the weight of the corresponding datapoint. To achieve the desired cell size, each seed is coupled with a weight that is then used in the power weighted distance function. In each iteration the weights are adjusted depending on the ratio of the current cell size to the desired cell size. Then the seeds are moved to the center of mass, to create a centroidal voronoi diagram, and afterwards, the cells are recalculated. This is repeated until all cells reached the desired size $\pm\epsilon$, where $\epsilon$ is some small number. The cells of the previous layer are then used as the bounding space for its children.

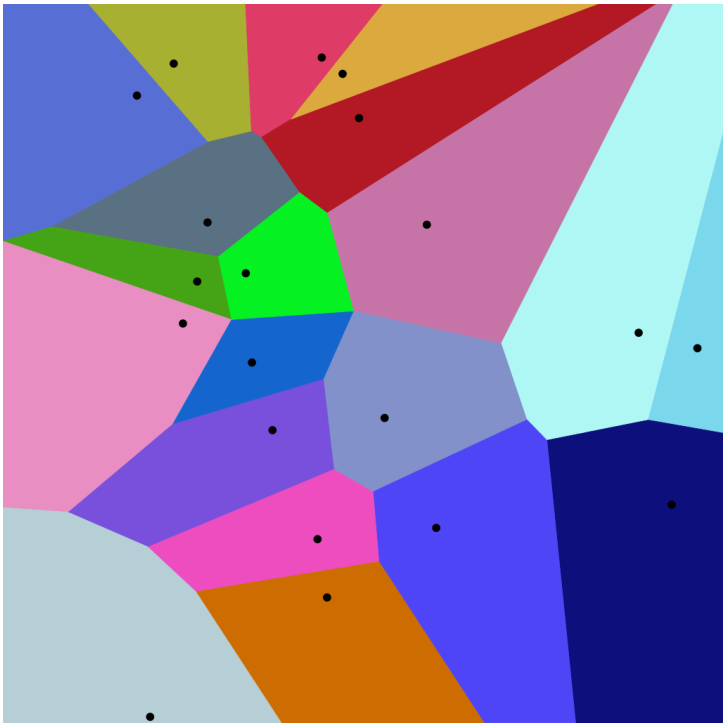

**Figure 2.1:** A Voronoi Diagram using the euclidean distance as distance function
[Diagram taken from Wikipedia]

**Figure 2.2:** A Voronoi Diagram using the manhattan distance as distance function
[Diagram taken from Wikipedia]

# Chapter 3

# Technologies

For the implementation of our interactive Voronoi treemap, several technologies are used. We use the *Node Packet Manager* [2020] package manager to install packages and include external libaries. *Webpack* [2020] is used as a module bundler to simplify the compiling process. The *Electron* [2020] tool was used to transform our web application into a standalone tool. *riot.js* [2020] was also included into our application and it is a simple component-based UI library used to design a web applications user interface.

Furthermore, some Javascript libraries are required, such as *d3* [2020] and *Pixi.js* [2020]. From d3 library we only use a selection of sub-libraries, namely *d3-dsv* [2020], *d3-hierarchy* [2020] and *d3-voronoi-treemap* [2020]. Since we also support CSV files to load the data, d3-dsv converts such a file into a d3-hierarchy. Our other supported file format is JSON, which will directly be handled with d3-hierarchy.

The d3-hierarchy package is needed because we use the d3-voronoi- treemap package to calculate an entire weighted Voronoi treemap. This library has a d3-hierarchy as input and output. Pixi.js is used to visualize our Voronoi treemap. This library offers simple drawing capabilities and uses WebGL or canvas depending on the browser. Finally, we also use the *pixi-viewport* [2020] library, which offers 2D camera capabilities for Pixi.js implementations. With this library, you have the possibility to zoom, click, drag and pinch within a Pixi.js application.

# Chapter 4

# Data-Handling

To display a dataset as a voronoi treemap, the user can upload either a JSON or a CSV file. In both cases the data needs to be in a specific format in order to create a *d3-hierarchy* [2020] which is needed as input to visualize the data. In the following examples especially the properties “name” and “weight” are important. While “name” is used as the label of the respective polygon, the “weight” of the leaves is used to compute the size of the polygon. The calculation is described in more detail in chapter 5.

## 4.1 JSON

This data is already in a hierarchical format and can be converted simply into a *d3-hierarchy* [2020], where “America” is the root node.

```
{
      "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 4.1:** JSON Dataset [*Global Economy By GDP Dataset* [2020], which was modified by the authors of this paper]

## 4.2 CSV

In comparison to the JSON example above, the following examples are not already in a hierarchical format. Therefore the tabular data needs to be transformed first. This can be accomplished with the d3-stratify function which is included in the *d3-hierarchy* [2020] library.

The first example, which is basically a cars dataset, consists of the columns “name”, “parent” and “weight”. In order to transform it into a hierarchy the columns “name” and “parent” are required. The last column named “weight” however is optional as described above.

```
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 4.2:** CSV Parent Unique Dataset [Dataset from the Github page of *d3-hierarchy* [2020]]

To be able to visualize hierarchies where the parents are non-unique, the columns “id”, “name” and “parentId” are required. Otherwise it would not be possible to differentiate which Alice is the parent of Bob or Doris in the family tree example below.

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

**Listing 4.3:** CSV Non-Unique Parent Dataset [Dataset created by Lisa Weißl]

# Chapter 5

# Calculations

As mentioned in chapter 3, we used d3-voronoi-treemap to calculate the shapes of the cells. d3-voronoi-treemap requires each leave to have a 'weight' parameter which is then proportional to the resulting cell size. If no weight was provided, we assign the same weight to all leaves of the data tree. This results in branches that are larger if they have more children and smaller if they have less. Furthermore, we normalize the weights so that the sum of all leaves is exactly 100. While d3 does not need normalized weights, we use them to scale parts of our visualization.

After the calculation of the polygons via d3, we extract all necessary data from the resulting d3-hierarchy and save it in our own class 'Polygon'. This is done by recursively stepping through the hierarchy. In each step we extract the weight, the cell shape and corresponding center, and add a color. Subsequent polygons are then added as children or parent of other polygons. By using our own class, that extens PIXI.Graphics, it becomes extremely easy to visualize it later on (See chapter 6).

The color for each polygon was calculated via the scale() method from chroma.js. We used a black-to-grey scale for the y axis and a colorful rainbow scale for the x axis. Each polygon was then assigned a color by inserting the x,y coordinates of its center into the two scales and mixing the two colors via the mix() method.

Lastly, we saved the root polygon, the rectangle that defines our applications visualization, as a variable in the model so that the entire hierarchy is accessible through its children.

# Chapter 6

# Visualization

This chapter describes the process of visualizing the voronoi treemap within our application. All drawing was done using *Pixi.js* [2020] as it offers representation in both WebGL and Canvas and provides easy ways to visualize variable polygons. The interaction with the treemap was implemented using *pixi-viewport* [2020], which provides all types of functionalities for PIXI.js applications including zooming, pinching and dragging.

## 6.1 Procedure

As mentioned above, the visualization is handled by *Pixi.js* [2020], the application is fully visualized within a PIXI.Application object that acts like a container that offers options like a stage to display objects, custom sizing and pixel resolution. The stage is the container that holds all objects that are to be rendered. Once the render() function is called all objects on the stage are rendered on screen as they were defined previously.

Once the input data has been read and processed, we receive a single root polygon which contains the rest of the tree. This root polygon, as well as all of its descendants, is a customized object that is extended from the PIXI.Graphics class [*Pixi.js* 2020]. The customized extension serves as a container for all the additional data that is needed for our application. This includes corner points, color, weight, name, children and parent. Using this data the tree, or at least part of it will be visualized for its initial representation.

The visualization of the tree always follows the same procedure, the children and the grandchildren (if they exist of course) of the current root polygon are drawn. The grandchildren have a low opacity, so they do not distract and the children receive labels that are defined by their name in the dataset. Additionally, also no matter how deep into the tree you are, the very first set of children of the original root polygon are also always visualized in order to keep a frame of reference within the tree. The current root polygon is also always highlighted by a thicker outline, for higher distinguishability.

No matter where in the tree your current viewpoint is, the visualization of the next instance can always be done in the same way. All previously drawn graphics are cleared and removed from the stage and all the new active shapes are freshly drawn according to their relevance to the current depth. The relevance always solely depends on the current root polygon, which is set to the polygon to which the viewpoint is currently set on. In this way we only draw what we have to, no unnecessary memory is used and re-draw times are seemingly instant.

All polygons of the upper layer of the currently visualized tree react to hovering actions. Once the cursor touches a polygon its opacity is reduced by 20% to improve the usability and provide the user with instant feedback. This opacity is of course increased back to 100% as soon as the cursor leaves the polygon.

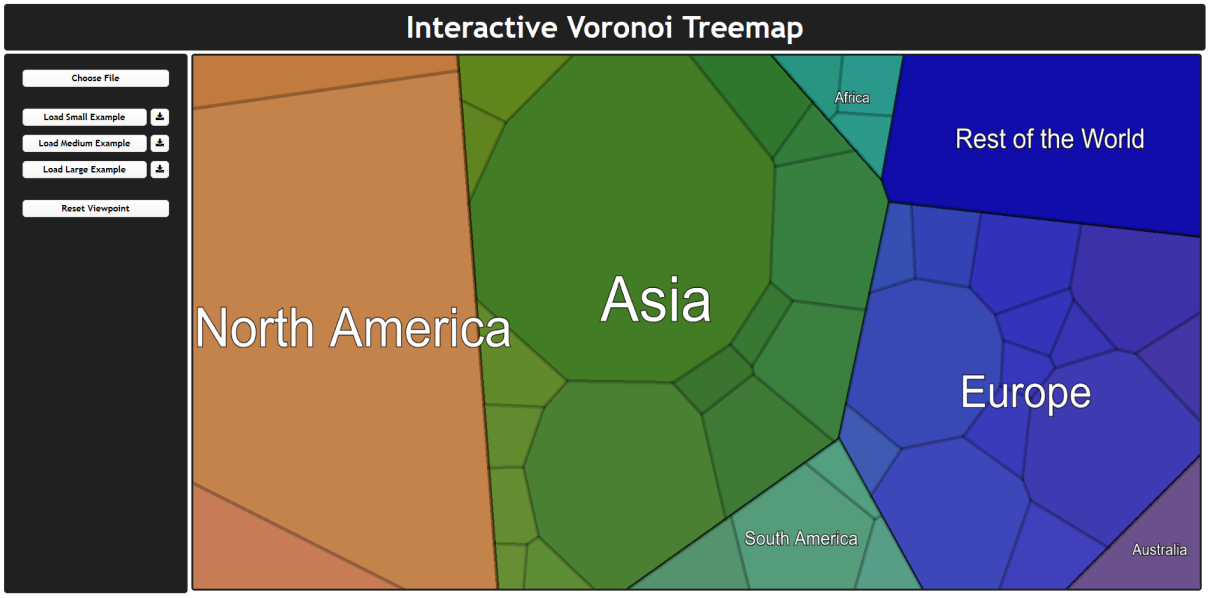


**Figure 6.1:** This shows the initial viewpoint of the small example whereby the root node is the invisible rectangle holding the continents. [Screenshot taken by Christopher Oser.]

## 6.2 Interactivity

All interactive parts of the application are handled by the *pixi-viewport* [2020] library. It is a handy tool that was developed to act as a 2D camera for Pixi.js applications. It offers simple options, like zooming by wheel and click, dragging across the application, as well as touchscreen support with pinching and swiping.

Zooming with the mouse wheel or by pinching the screen, as well as dragging with the cursor or finger is always possible within the treemap. This however does not change the current visualization and is solely a possibility for the user to freely move through the treemap and increase the size of labels or the centering of the viewpoint.

To change the visualization and trigger a re-draw to step through the layers, the user must click the screen. The coordinates of this click are then evaluated and the polygon that contains the coordinates is set as the current root polygon. This then triggers a re-draw for this polygon and zooms to the center of that polygon, while maintaining a zoom ratio so the entire polygon fits within the viewpoint. This ratio is chosen by the widest length within the polygon, which is calculated by comparing the corner points of the polygon.

In case a polygon is clicked, that is outside of the children of the current root polygon, the current root polygon is set to the current root polygon's parent and a re-draw is triggered. In this case the last zoom ratio is reversed and you end up with the previous zooming ratio. Which simply put means, it moves out one layer within the tree. Of course zoom actions are only possible if there is data to be displayed, meaning, no zooming out when viewing the highest layer and no zooming in when viewing the leaves of the tree.

An additional button was added to reset the viewpoint to the original root polygon, to provide users with an option to reset their current application in case they get overwhelmed or want to start over. This button simply resets the current root polygon to the original root polygon, triggers a re-draw and resets the zoom ratio to 1.

Since pixi-viewport uses real zoom, sizes as the text font, resolution and line thickness increase dramatically when zooming in. To counter this, all these options were made variable to the current zoom ratio, this way resolution increases, font size decreases and also line thickness decreases on positive

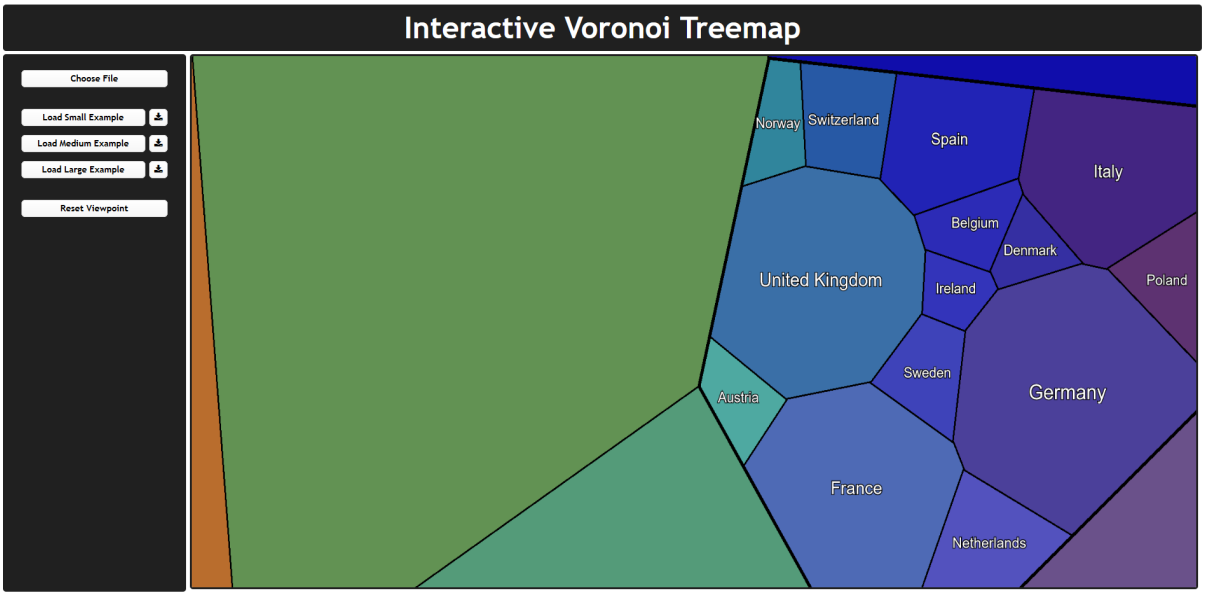


**Figure 6.2:** This shows the zoomed viewpoint of the first layer of the small sized example whereby the root polygon is the "Europe" polygon that can be seen in Figure 6.1. [Screenshot taken by Christopher Oser.]

zooming actions. Naturally, the reverse is true for negative zooming actions.

# Chapter 7

# Results

This chapter presents our final results that were achieved during an effort of creating a web-based interactive Voronoi treemap. It gives a short overview and example screeshots of our final application "IVT", which is short for Interactive Voronoi Treemap.

The application is basically split into two sections, a side navigation bar on the left and the interactive voronoi treemap on the right which occupies most of the space. When loading the application only the side navigation bar is visible whereas the treemap is empty. The user now can either load a own CSV or JSON file in the specified format or has the possibility to load one of our example files. We provide three examples, each in a different size and format which can also be downloaded in order to let the user take a look at it. While loading the file, the word "LOADING..." appears on the side navigation bar, which is a indicator that the application is processing the file. After the file was loaded successfully, the user can interactively step through the hierarchy layers of the treemap by clicking on the polygons. The view of different hierarchy layers of the large example can be seen in figures 7.1, 7.2 and 7.3. When clicking outside the polygon, the user is able to step one layer back. In case the user wants to recover the initial view of the treemap, we also added a "Reset Viewpoint" button, to prevent the user from stepping all way up.

In conclusion it can be said that we have managed to create a simpler version of *FoamTree* [2020], which in contrast is free and open source. With this application we give the user the possibility to view his hierarchical data from a different perspective and since it is open source, to expand also the functionality of the application.

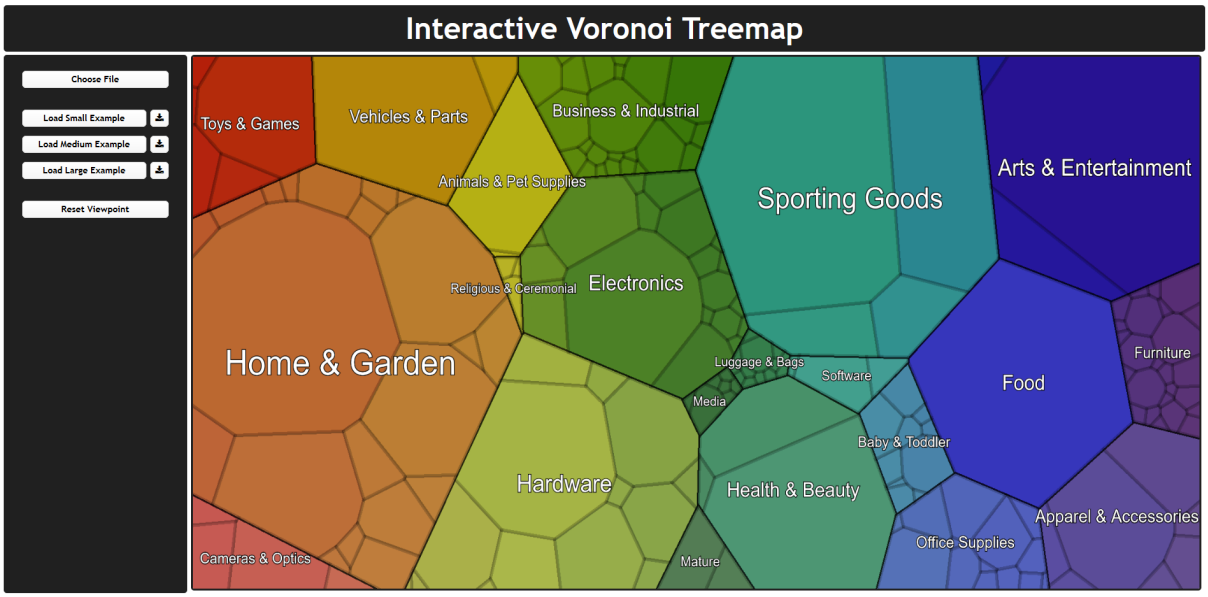


**Figure 7.1:** IVT right after loading the large example. [Screenshot taken by Christopher Oser.]

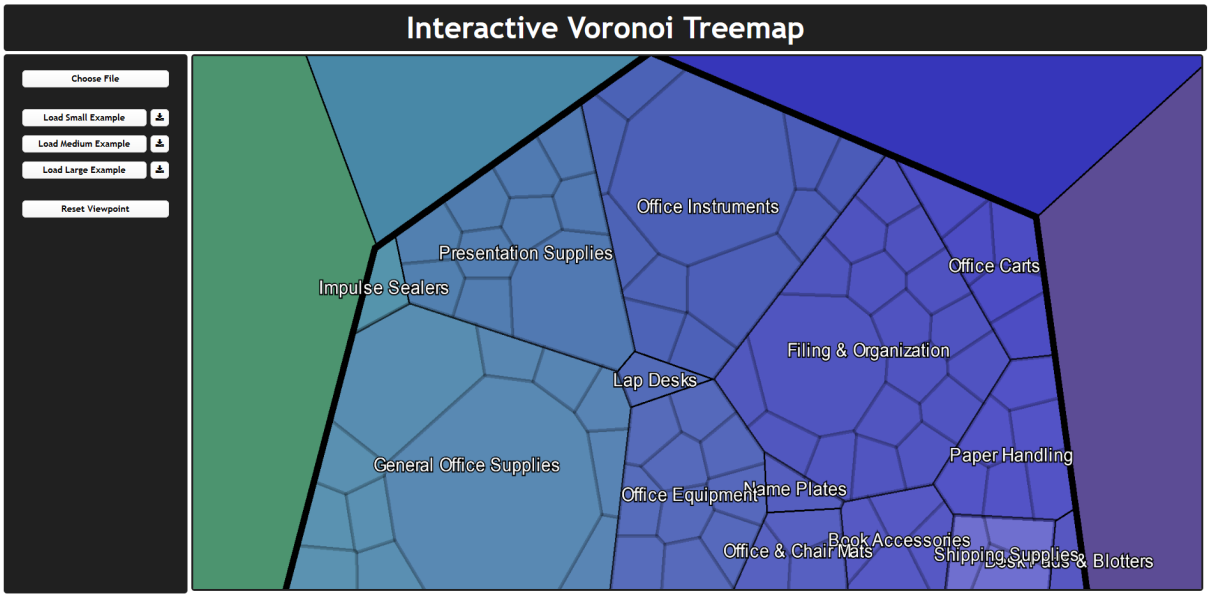


**Figure 7.2:** The changed view of the large example, after clicking on the “Office Supplies” polygon. Now only the labels of the current hierarchy layer and the clicked polygon are visible. [Screenshot taken by Christopher Oser.]

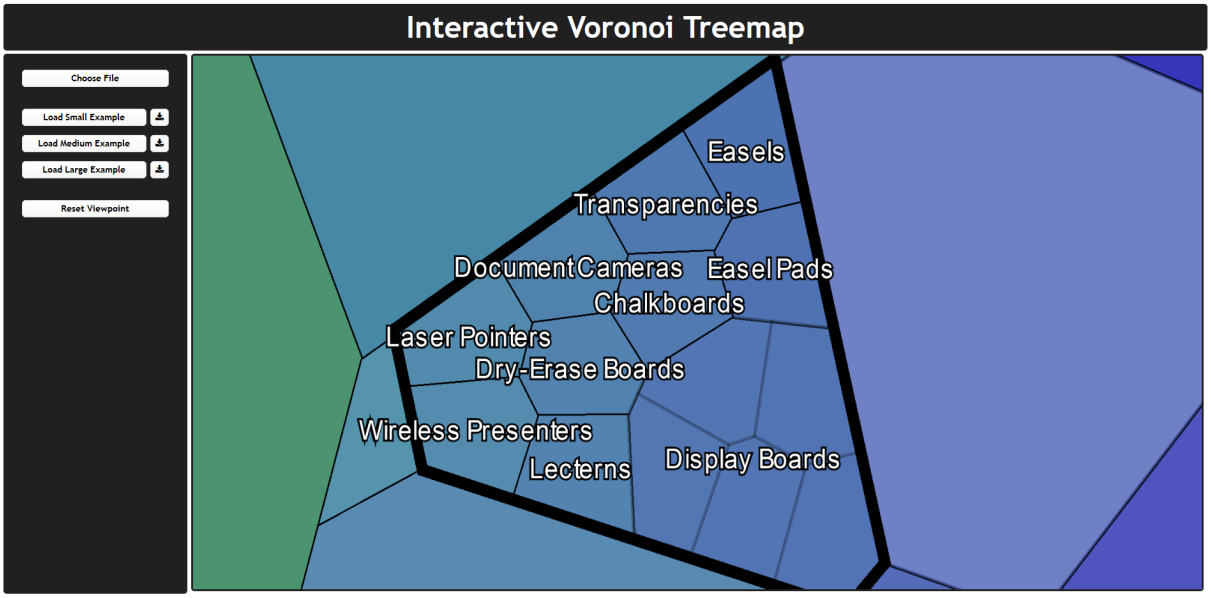


**Figure 7.3:** The view of an deeper layer after clicking on the "Presentation Supplies" polygon in Figure 7.2. [Screenshot taken by Christopher Oser.]

# Bibliography

*d3* [2020]. https://d3js.org/ (visited on 27 Jun 2020) (cited on page 5).

*d3-dsv* [2020]. https://github.com/d3/d3-dsv (visited on 27 Jun 2020) (cited on page 5).

*d3-hierarchy* [2020]. https://github.com/d3/d3-hierarchy#stratify (visited on 27 Jun 2020) (cited on pages v, 5, 7–8).

*d3-voronoi-treemap* [2020]. https://github.com/Kcnarf/d3-voronoi-treemap (visited on 27 Jun 2020) (cited on page 5).

*Electron* [2020]. https://www.electronjs.org/ (visited on 27 Jun 2020) (cited on page 5).

*FoamTree* [2020]. https://carrotsearch.com/foamtree/ (visited on 27 Jun 2020) (cited on pages 1, 15).

*Global Economy By GDP Dataset* [2020]. https://github.com/WYanChao/Orthogonal-Voronoi-Treemap/blob/master/data/globalEconomyByGDP.json (visited on 27 Jun 2020) (cited on pages v, 7).

*Node Packet Manager* [2020]. https://www.npmjs.com/ (visited on 27 Jun 2020) (cited on page 5).

*pixi-viewport* [2020]. https://github.com/davidfig/pixi-viewport (visited on 18 Jun 2020) (cited on pages 5, 11–12).

*Pixi.js* [2020]. https://www.pixijs.com/ (visited on 18 Jun 2020) (cited on pages 5, 11).

*riot.js* [2020]. https://riot.js.org/ (visited on 27 Jun 2020) (cited on page 5).

*Webpack* [2020]. https://webpack.js.org/ (visited on 27 Jun 2020) (cited on page 5).