Multi-Dimensional Visualisation with Three.js and svelte

Project Report

Valerio Mariani, Sebastian Überreiter, Christoph Söls

706.057 Information Visualisation 3VU SS 2022
Graz University of Technology

13 May 2022

Abstract

Nowadays, there are numerous frontend frameworks and libraries, which make it easier for developers to implement a web application. Because of this large variety, sometimes it can be hard to select, which libraries and frameworks to choose.

In this project report, we look at the specific application of information visualisation and explore a graphics library called Three.js and a frontend framework called svelte. The Three.js library simplifies the creation of interactive 3d graphics. The svelte framework makes is easy to build lightweight reactive applications. The aim of the project is to introduce the graphics library and framework, and explain the necessary workflow. For this, an interactive information visualisation application was created which includes a scatterplot, scatterplot matrix, similarity map, and parallel coordinates plot, as well as components for CSV import and SVG export.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

The goal of this project was to create a proof-of-concept web application to visualise multi-dimensional datasets using Three.js and svelte. It was named 3mvis. The user can import a dataset from a CSV file. Four synchronised visualisations are implemented: scatterplot, scatterplot matrix, similarity map, and parallel coordinates plot.

1.1 Using the svelte Frontend Framework

The basic frontend of the whole project is set upon svelte [svelte 2022a]. Svelte is a free JavaScript frontend framework that builds upon HTML, CSS and JavaScript. Svelte makes it easy to build lightweight reactive applications, where changes in underlying data are reflected immediately and automatically in the user interface. It compiles given code into vanilla JavaScript rather than using a virtual DOM, which leads to a very fast starting and running program.

Svelte provides a useful service to get started on coding with the framework. For novice users, an online sandbox can be used to try out different things, without having to struggle around with setting up the whole project [svelte 2022b].

For users who want to create their own application locally, Node.js [Node.js 2022] should first be installed. Then, it is as easy as issuing the four commands shown in Listing 1.1 to set up an initial svelte environment. The application is started with the command "npm run dev" and can be seen by pointing a modern web browser to localhost:8080. Once the svelte project is fully initialised, the devleoper has an "App.svelte" file and a "main.js" file and is free to start coding their own application.

1.2 Using the Three.js Graphics Library

Three.js is a JavaScript library which supports the creation of 3D computer graphics using WebGL [mrdoob 2022]. It also supports rendering to SVG or to a standard 2d Canvas. The Three.js library can be downloaded in a full or minified version and added to the project (recommended in a /src folder) by importing it. Listing 1.2 shows the two ways a developer might import Three.js for usage.

Three basic components need to be configured in order to draw graphics with Three.js:

- Scene: a collection of 3d objects.
- Camera: a specific view of the scene.
- Renderer: the technology used to draw the scene to the display.

Three.js uses these components to render the scene with a camera. The scene can be created with the simple command shown in Listing 1.3.

This project uses a "PerspectiveCamera", which can be created via a simple function call with four

```
npx degit sveltejs/template mysvelte-project
cd my-svelte-project
npm install
npm run dev
```

Listing 1.1: Svelte: Setting up a svelte environment.

```
import * as THREE from "./three";
<script src="js/three.js"></script>
```

Listing 1.2: Three.js: Importing the library.

parameters: a field of view, aspect ratio, a near plane, and a far plane, as shown in Listing 1.4. The field of view tells the camera how much of the scene should be visible on the screen at all time. The aspect ratio is generally a width divided by height, to achieve the modern standard resolution. The near and far plane tell at what distance objects will no longer be rendered. Everything before the near plane and everything behind the far plane will not be rendered. This parameter can be overseen for beginners, but could be usable for advanced users to increase the efficiency of the program.

The renderer can also be created with a simple function call. This project uses the Three.js WebGL Renderer, as shown in Listing 1.5.

After all three components have been created, the renderer is appended to the document and the user is free to start using the full Three.js functionality. A simple example for novice users can be seen in Listing 1.6.

1.3 CSV Import

The D3 [Bostock 2022b] module d3-dsv [Bostock 2022a] is used for importing CSV files.

```
let scene = new THREE.Scene();
```

Listing 1.3: Three.js: Creating a scene.

```
let camera = new THREE.PerspectiveCamera(
  50,                                   // field of view
  window.innerWidth / window.innerHeight,  // aspect ratio
  0.1,                                  // min. z distance
  1000                                  // max. z distance
);
```

Listing 1.4: Three.js: Creating a camera.

```
let renderer = new THREE.WebGLRenderer();
```

Listing 1.5: Three.js: Creating a renderer.

```
let geometry = new THREE.BoxGeometry( 1, 1, 1 );
let material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
let cube = new THREE.Mesh( geometry, material );
scene.add( cube );
camera.position.z = 5;
```

Listing 1.6: Three.js: Placing a cube in a scene.

Chapter 2

User Interface

The user interface of the 3mvis software consists of the top bar and four panels containing the visualisations, as shown in Figure 2.1.

2.1 Top Bar

The top bar is used for basic navigation and interaction:

- File Chooser to load a CSV dataset.
- Rendering technology dropdown.
- Scatterplot X and Y dimension chooser.
- SVG Export field.

The CSV input field provides a file explorer where the user can upload a dataset from a local CSV file into the program. Its content will then be parsed by d3.csvParse() and transformed into data arrays which will be passed to the different plotting functions. The scatterplot X and Y drop-down menus let the user decide which dimensions of interest should be plotted against each other on the X and Y axes of the scatterplot. Options will be given according to the dimensions of the input data. The SVG export field lets the user choose one of the 4 given panels and exports the chosen canvas as SVG.

2.1.1 CSV Input Workflow

The user is able to input a CSV file with the help of an HTML <input> field in the header bar (with its accepted data type set to .csv). In the code, at first the import of the d3-dsv package is needed, as it includes the function d3.csvParse(), which was introduced in D3v4 (it was formerly called d3.csv.parse()). After that, the handling of the resulting object is the same as with any JSON encoded object. The relevant parts of the import function are shown in Listing 2.1.

2.1.2 SVG Export Workflow

For the user to download a choosen canvas as SVG, the canvas is first rendered separately with the Three.js SVG renderer. This step is crucial, since the canvas visible to the user is rendered with the Three.js WebGL renderer, and cannot be downloaded as SVG. The canvas is therefore rendered with its given scene and camera and passed to a new SVGRenderer(), as shown in Listing 2.2.

The re-rendered canvas is then passed to a function ExportToSVG, which transforms it into a downloadable SVG. The function uses an XML-Serializer to convert the given DOM element from the rendered SVG canvas into a stream of bytes [IBM 2021]. A Blob (Binary Large Object) [Microsoft 2022] object is further used to store the serialized data with a preface and a type declaring the image to be of type .svg.

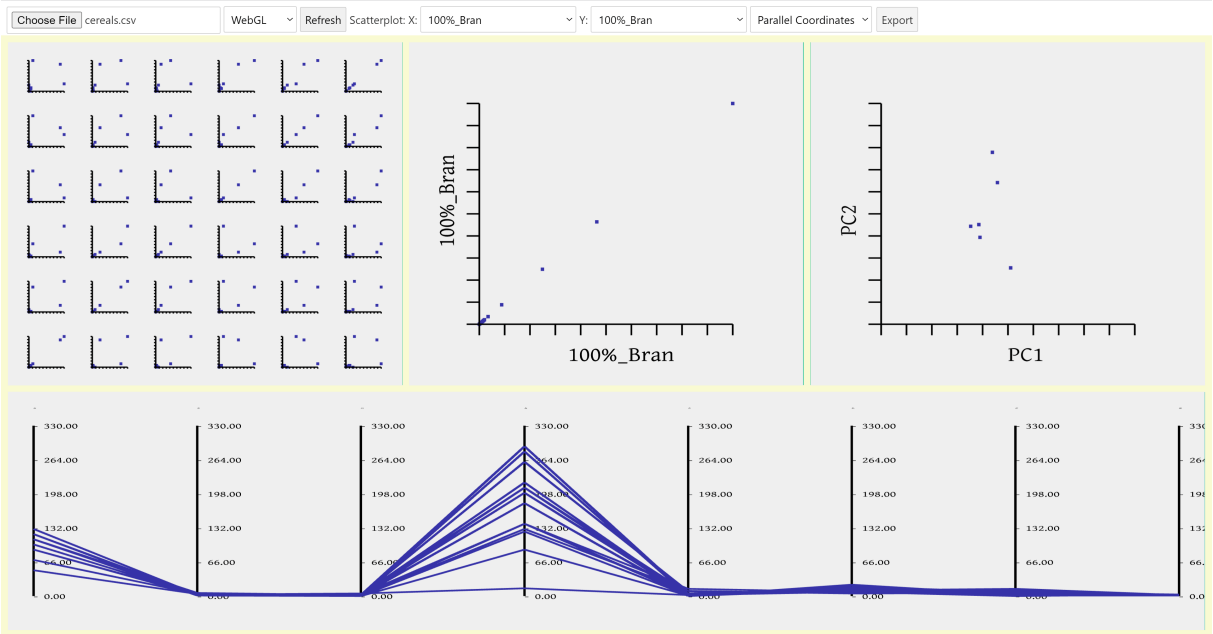

Figure 2.1: The user interface of 3mvis. [Screenshot taken by the authors of this report.]

```
const input = file;

const reader = new FileReader();

reader.onload = function (e) {

  const text = e.target.result;

  var data_csv = d3.csvParse(text);

  // data_csv is a JSON encoded object, which can be used to fill
  // the data arrays for the visualisations

  // render the plots on the canvases.
  // each rendering function returns a record with associated data
  r[0] = render_paracord(canvass[0], divs[0], data_all);
  r[1] = render_scatterplot_matrix(canvass[1], divs[1],
    data_all, scat_x.value, scat_y.value);
  r[2] = render_scatterplot(canvass[2], divs[2],
    data_all, scat_x.value, scat_y.value);
  r[3] = render_similarity_map(canvass[3], divs[3], data_all);
};

reader.readAsText(input);
```

Listing 2.1: The code to import a dataset from a CSV file.

```
let select = document.getElementById("export_target");
let selected_r = r[select.selectedIndex];
let camera = selected_r.camera;
let scene = selected_r.scene;
let rendererSVG = new SVGRenderer();
let container = divs[select.selectedIndex];
rendererSVG.setSize(container.clientWidth, container.clientHeight);
rendererSVG.render(scene, camera);
ExportToSVG(rendererSVG, "chart.svg");
```

Listing 2.2: Rendering of the chosen Canvas with a SVGRenderer.

```
var XMLS = new XMLSerializer();
var svgfile = XMLS.serializeToString(rendererSVG.domElement);

var svgData = svgfile;
var preface = '<?xml version="1.0" standalone="no"?>\r\n';
var svgBlob = new Blob([preface, svgData], {
  type: "image/svg+xml;charset=utf-8",
});
var svgUrl = URL.createObjectURL(svgBlob);
var downloadLink = document.createElement("a");
downloadLink.href = svgUrl;
downloadLink.download = filename;
document.body.appendChild(downloadLink);
downloadLink.click();
document.body.removeChild(downloadLink);
```

Listing 2.3: Creating a downloadable SVG.

Then, a URL is created, pointing to the given Blob element, and it is added to a created download link which will then automatically download the SVG for the user. The function can be seen in Listing 2.3.

2.2 Visualisation Grid

The four visualisation are laid out in a grid using CSS Grid. The parallel coordinate plot extends across the bottom of the window for its entire width. The other three visualisations (scatterplot matrix, scatterplot, and similarity map) each take up one third of the width and the remaining height of the window, as can be seen in Figure 2.1.

The whole page resizes when the window size changes. Resize functions in each plot make this possible. The grid is created inside its own class with the CSS directive "display: grid" to indicate that its contents are a CSS Grid. The grid is defined using "grid-template-area", as shown in Listing 2.4. In this project, the proportions of the grid are fixed. It is possible to use flexible values which then resize according to the input.

Inside each grid area, a element holds a canvas which is used to render a visualisation, as shown in Listing 2.5. In CSS, each div is assigned to its respective grid area, as shown in Listing 2.4.

```
display: grid;
grid-template-areas:
  "area1 area1 area1 area2 area2 area2 area3 area3 area3"
  "area1 area1 area1 area2 area2 area2 area3 area3 area3"
  "area1 area1 area1 area2 area2 area2 area3 area3 area3"
  "area4 area4 area4 area4 area4 area4 area4 area4 area4"
  "area4 area4 area4 area4 area4 area4 area4 area4 area4";
grid-gap: 10px;


.item1 {
  grid-area: area1;
}
```

Listing 2.4: Defining the CSS grid layout.

```
<div class="grid-container">
  <div bind:this={divs[1]} class="item1">
    <canvas bind:this={canvass[1]} id="c1" />
  </div>
  <div bind:this={divs[2]} class="item2">
    <canvas bind:this={canvass[2]} id="c2" />
  </div>
  <div bind:this={divs[3]} class="item3">
    <canvas bind:this={canvass[3]} id="c3" />
  </div>
  <div bind:this={divs[0]} class="item4">
    <canvas bind:this={canvass[0]} id="c4" />
  </div>
</div>
```

Listing 2.5: Setting up drawing canvases for each visualisation.

The respective and <canvas> elements are then stored in their own arrays to be used later in the JavaScript code to display the four visualisations.

Chapter 3

Visualisations

The 3mvis prototype implements four different visualisations of a multi-dimensional dataset: scatterplot, scatterplot matrix, similarity map, and parallel coordinates. Graphical output is rendered using Three.js, which has some peculiarities when it comes to drawing text and lines.

3.1 Drawing Text

Text creation was initially intended to be done using a CSS2DObject paired with a CSS2DRenderer. This worked pretty well as a first solution, but became problematic when downloading the plots as SVG, since the SVGRenderer was not able to render the text and so the downloaded SVGs had no text. Three.js text creation was then chosen as a suitable solution and could easily be paired with the SVGRenderer, such that the text was also visible in the downloaded SVG. Creating text with Three.js is straightforward since Three.js has its own "Textgeometry" which can be easily used to create text. First, one creates a "Textgeometry" object which can be given different parameters such as font, size or height of the text, as shown in Listing 3.1.

After creating the geometry of the text, a new material is created which gives the text its black color. The material and the geometry is then connected in a mesh which defines the final text object. This object can then be variously translated, rotated, or transformed as desired, which marks another great feature of Three.js. To see the text on-screen, it is simply added to the scene. The code for mesh creation can be seen in Listing 3.2.

3.2 Drawing Lines

Three.js itself provides built-in functionality to draw lines, but there are issues with line thickness and selectability (when hovering or clicking). As a workaround, lines are drawn instead as very thin 2d rectangles. The corresponding function draw_line calculates the length of the input vector and the angle theta it is supposed to be rotated to. Then, a "Boxgeometry" is created with the given length and a "Basicmaterial" with the input color, which are combined to create the mesh representing the line. The mesh is then transformed onto the plot and added to the scene. The whole code can be seen in Listing 3.3.

3.3 Scatterplot

The basic scatterplot is drawn with the two functions draw_scatterplot_axes and draw_scatterplot_points, which are called from the render_scatterplot function. As shown in Listing 3.4, the custom data object is filled with the selected dimensions from the drop-down menu.

The axes tick marks for the scatterplot are drawn with a loop and the custom line creation function. The tick marks are drawn at a uniform distance from each other. An improvement at this point would be

```
const geometry = new TextGeometry(t, {
  font: font,
  size: font_size,
  height: 0,
});
```

Listing 3.1: Three.js: Creating text geometry.

```
const material = new THREE.MeshBasicMaterial({ color: 0x000000 });
const text = new THREE.Mesh(geometry, material);
text.position.x = x;
text.position.y = y;
if (rot) {
  text.rotation.z = rot;
}

scene.add(text);
```

Listing 3.2: Three.js: Creating a text mesh and adding it to the scene.

```
function draw_line(x0, y0, x1, y1, scene, thickness = 0.02, color = "#3632a8") {
  // length of vector from (x0,y0) to (x1,y1)
  let l = Math.sqrt(Math.pow(x1 - x0, 2) + Math.pow(y1 - y0, 2));

  // angle of vector from (x0,y0) to (x1,y1)
  let theta = -Math.atan((x1 - x0) / (y1 - y0));

  const geometryrect = new THREE.BoxGeometry(thickness, l, 0.01);

  const material = new THREE.MeshBasicMaterial({ color: color });
  const rect = new THREE.Mesh(geometryrect, material);

  rect.position.set(0, 0, 0);
  rect.position.x += (x1 - x0) / 2 + x0;
  rect.position.y += (y1 - y0) / 2 + y0;

  rect.rotation.z = theta;

  scene.add(rect);
  return rect;
}
```

Listing 3.3: Three.js: Drawing a line.

```
for (let i = 0; i < _data.length; i++) {
  if (_data[i].name == scat_x) {
    d1 = _data[i].data;
  }
  if (_data[i].name == scat_y) {
    d2 = _data[i].data;
  }
}

let names = [];

for (let i = 0; i < _data.length; i++) {
  let obj = _data[i];
  names.push(obj.name);
}

let dataset_props = {
  dims: [scat_x, scat_y],
  min_x: Math.min(...d1),
  max_x: Math.max(...d1),
  min_y: Math.min(...d2),
  max_y: Math.max(...d2),
};

let chart_props = {
  width: 30,
  height: 30,
  offset_x: 0,
  offset_y: 0,
}

let data = {
  v1: d1,
  v2: d2,
  names: names
}

draw_scatterplot_axes(chart_props, dataset_props, scene);
dots = draw_scatterplot_points(chart_props, dataset_props, data, scene);
```

Listing 3.4: Scatterplot: Drawing a scatterplot.

```
// x-axis
draw_line(off_x - (width / 2), off_y - (height / 2), off_x + (width / 2),
  off_y - (height / 2), scene, .2, 0x000000);

// x-axes tick marks
for (let i = 0; i <= 10; i++) {
  draw_line(off_x - (width / 2) + (i * step_x), off_y - (height / 2),
  off_x - (width / 2) + (i * step_x), off_y - (height / 2) - (height / 20),
  scene, .2, 0x000000);
}

// tick mark labels
if (draw_text) {
  text(off_x - .5 * dims[0].length, off_y - (height / 2) - 5, dims[0],
    scene, 2 * width / 30);
  text(off_x - (width / 2) - 3, off_y - .5 * dims[1].length, dims[1],
    scene, 2 * width / 30, 0x00000, 90 * (Math.PI / 180));
}
```

Listing 3.5: Scatterplot: Drawing the axis tick marks and labels.

to name the axis markings based on the given data. As seen in Listing 3.5, the size of the chart is initially 6×6 (resized later) and the axes are drawn in the full range. The points of the scatterplot are calculated by scaling the real points to a new range, as shown in Listing 3.6.

3.4 Scatterplot Matrix

A scatterplot matrix (or SPLOM) is a grid of scatterplots for every pair of dimensions. Once functions for drawing the scatterplot were made, they had to be extended in such a way that they could draw small scatterplots and allow for translation of the origin of their axes. To do that, ability was implemented to add specific parameters to the "chart_props" data structure that the function takes as a parameter: width and height can be specified together with x and y offset values to translate the scatterplot. Also, the optional "draw_text" boolean parameter was added to permit disabling drawing of axis labels. Once the functions were extended, it was possible to draw a matrix of scatterplots as shown in Listing 3.7. The number of dimensions is currently capped at a maximum of seven.

3.5 Similarity Map

For this prototype, the simplest form of similarity map is used: the first two components from a Principal Component Analysis (PCA). An external library is used to calculate the eigenvalues and eigenvectors [bitanath 2022]. The eigenvectors are already sorted by highest eigenvalues, so the first two principal components are calculated and later plotted as PC1 and PC2 in the canvas. The code for calculating the first two principal components can be seen in Listing 3.8.

To display the similarity map on the canvas, a renderer, scene and camera are created using the Three.js library, as shown in Listing 3.9.

Further, the min and max values of both data arrays are filtered, stored and later given with the original array of data points to the external drawing function for plotting the data points. This can be seen in Listing 3.10.

Two functions are used to plot the similarity map axes and the similarity map data points. The function

```
for (let i = 0; i < v1.length; i++) {
  let point = new Float32Array([
    -(width / 2) + (((v1[i] - min_x) / (max_x - min_x)) * width),
    -(height / 2) + (((v2[i] - min_y) / (max_y - min_y)) * height), 0]);
  let dotGeometry = new THREE.BufferGeometry();
  dotGeometry.setAttribute('position', new THREE.BufferAttribute(point, 3));
  let dotMaterial = new THREE.PointsMaterial({ size: 1, color: 0x3632a8 });
  let dot = new THREE.Points(dotGeometry, dotMaterial);
  if (names) {
    // name is associated to the point as an identifier for hover
    dot.name = names[i];
  }
  dots.push(dot);
  scene.add(dot);
  dot.position.x += off_x;
  dot.position.y += off_y
}
```

Listing 3.6: Scatterplot: Drawing the data points.

```
for (let j = 0; j < 7; j++) {
  for (let i = 0; i < 7; i++) {
    d1 = _data[i].data.filter(x => !isNaN(x));
    d2 = _data[j].data.filter(x => !isNaN(x));;
    let data = {
      v1: d1,
      v2: d2,
    }
    let ii = i - 3.5;
    let jj = j - 3.5;
    let chart_props = {
      width: 5,
      height: 5,
      offset_x: 9 * ii,
      offset_y: 9 * jj,
    }
    let dataset_props = {
      dims: [scat_x, scat_y],
      min_x: Math.min(...d1),
      max_x: Math.max(...d1),
      min_y: Math.min(...d2),
      max_y: Math.max(...d2),
    };

    draw_scatterplot_axes(chart_props, dataset_props, scene, false);
    draw_scatterplot_points(chart_props, dataset_props, data, scene);
  }
}
```

Listing 3.7: Scatterplot Matrix: Creating the matrix of scatterplots.

```
import * as PCA from "./pca";

let vectors = PCA.getEigenVectors(test_data);

let psim1 = vectors[0]['vector'];
let psim2 = vectors[1]['vector'];
let datasimmap = [psim1, psim2];
```

Listing 3.8: Similarity Map: Calculating the first two PCA components, using the code from bitanath [2022].

```
renderer = new THREE.WebGLRenderer({ antialias: true, canvas: el });
renderer.setPixelRatio(window.devicePixelRatio);

scene = new THREE.Scene();
scene.background = new THREE.Color(0xefefef);
camera = new THREE.PerspectiveCamera(
  50, // angle of view
  window.innerWidth / window.innerHeight, // pixel ratio
  0.1, // min. seeable z distance
  1000 // max. seeable z distance
);
```

Listing 3.9: Similarity Map: Displaying the similarity map.

```
let dataset_props = {
  min_x: Math.min(...psim1),
  max_x: Math.max(...psim1),
  min_y: Math.min(...psim2),
  max_y: Math.max(...psim2),
};

sim_dots = draw_similarity_map_points(dataset_props, datasimmap, scene, names);
```

Listing 3.10: Similarity map: Calculating min and max values.

```
let width = 30;
let height = 30;

let step_x = width / 10;
let step_y = height / 10;

// x-axis
draw_line(-(width / 2), -(height / 2), (width / 2), -(height / 2),
  scene, .2, 0x000000);

// y-axis
draw_line(-(width / 2), -(height / 2), -(width / 2), (height / 2),
  scene, .2, 0x000000);

// x-axis tick marks
for (let i = 0; i <= 10; i++) {
  draw_line(-(width / 2) + (i * step_x), -(height / 2),
    -(width / 2) + (i * step_x), -(height / 2) - (height / 20),
    scene, .2, 0x000000);
}

// y-axis tick marks
for (let i = 0; i <= 10; i++) {
  draw_line(-(width / 2), -(height / 2) + (i * step_y),
    -(width / 2) - (width / 20), -(height / 2) + (i * step_x),
    scene, .2, 0x000000);
}

// axis titles
text(0, -(height / 2) - 5, "PC1", scene, 2);
text(-(width / 2) - 3, 0, "PC2", scene, 2, 0x00000, 90 * (Math.PI / 180));
```

Listing 3.11: Similarity Map: Drawing axes, tick marks, and axis titles.

to create the x and y axes uses a fixed width and height along with a fixed step size for the x and y axes to mark the steps. The creation of the axes and the lines can be seen in Listing 3.11.

The second function is used to plot the data points onto the canvas. Its parameters include the pre-calculated min and max values, the data itself, the scene and the names of the data inside the datasets. Plotting the points is rather easy. A Float32Array is created for each point containing the x, y and z values of the point to be drawn. Z is in this project always zero since this project only draws in 2D. The point is then stored in a Three.js "buffergeometry" and is paired with a "dotMaterial", containing the size and color, to create a Three.js point. Lastly, the point is added to the scene and into a separate array for later use. The whole function can be seen in Listing 3.12.

3.6 Parallel Coordinates

The parallel coordinates plot is done using a combination of two functions: draw_parallel_coordinates_axes and draw_parallel_coordinates. The first draws the axes of the parallel coordinates plot, while the second one draws the lines passing through the axes. To draw the axes, the x coordinate of the first axis must be specified in the chart_props parameter data structure through the start_point field. Then, supposing the dataset has n dimensions, and assuming $x_0 = |\text{chart\_props.start\_point}|$, the first axis is positioned at x=$-x_0$, the last axis is positioned at x=x_0, and the remaining n-2 axes are positioned at regular intervals between

```
let point = new Float32Array([
  (((v1[i] - min_x)/(max_x - min_x)) * width),
  (((v2[i] - min_y) / (max_y - min_y)) * height), 0]);

let dotGeometry = new THREE.BufferGeometry();
dotGeometry.setAttribute('position', new THREE.BufferAttribute(point, 3));
let dotMaterial = new THREE.PointsMaterial({ size: 1, color: 0x3632a8 });
let dot = new THREE.Points(dotGeometry, dotMaterial);
scene.add(dot);
dot.name = names[i];
dots.push(dot);
```

Listing 3.12: Similarity Map: Drawing data points.

the first and last axes. Also, y_0 = chart_props.max_height is specified so each axis vertically spans from $-y_0$ to y_0. The code for the draw_parallel_coordinates_axes function can be found in Listing 3.13.

Once the axes are drawn, each polyline of the parallel coordinates plot passes through one of the axes at an height $y' = linear - scaling([dataset\_props.min, dataset\_props.max] \rightarrow [-y_0, y_0], y)$, where $linear - scaling([s1, e1] \rightarrow [s2, e2], y)$ scales the point $y \in [s1, e1]$ from its domain to the the domain $[s2, e2]$ $\forall y \in [s1, e1]$. The code for the draw_parallel_coordinates functions can be found in Listing 3.14.

3.7 Interactivity

Unfortunately, when rendering Three.js graphics with a WebGLRenderer object, built-in interactivity features of HTML related content is lost. This means that if, for example, hovering features are to be used, they need to be implemented from scratch. Here, we demonstrate how to implement hover and selection in Three.js with Raycaster and Vector objects together with JavaScript's built-in "pointermove" event to global "window" variable.

To highlight a selected Three.js object, first the necessary objects need to be initialised and a function needs to be bound to the "pointermove" event, as shown in Listing 3.15.

Next, the function onPointerMove() needs to be implemented. Basically, it needs to set the pointer variable to be positioned where the mouse pointer is, with coordinates normalised w.r.t. the size of the canvas. This can be done with the code in Listing 3.16.

Then, the raycaster object can be used to find which objects intersect a line from the camera to the pointer, and these objects can be highlighted, as shown in Listing 3.17.

Unfortunately, this is not yet sufficient to fully implement hovering functionality. When the pointer passes out of an intersected object, the object needs to be unhighlighted again. While a pointer is moved, a list of currently hovered objects is maintained. Note that this actually involves keeping two lists. When a pointer moves, an object known as old_intersects contains the objects that were intersected by the ray-caster the last time the pointer moved. A second object called intersects is updated to contain the currently intersected objects. Then, the difference between the two lists can be computed and these objects unhighlighted. This is shown in Listing 3.18.

To achieve brushing across the fout different views (cross-canvas hovering), a string identifier was attached to every record (data point) in the dataset, so that graphical elements representing the same data point could be retrieved by looking for identifiers. Brushing is shown in Figure 3.1, where the hovered

```
function draw_parallel_coordinates_axes(chart_props, dataset_props, scene) {
  // number of dimensions
  let n = dataset_props.n;

  // vector containing labels of the various dimensios
  let dims = dataset_props.dims;
  let min = dataset_props.min;
  let max = dataset_props.max;

  // x position of the first axis. The last one specular w.r.t. the y=0 point
  let start_point = chart_props.start_point;

  // y position of the end of the axis
  let max_height = chart_props.max_height;

  // distance between one axis and the other
  let step = chart_props.step;

  for (let i = 0; i < n; i++) {
    draw_line(start_point + (i * step), -max_height, start_point + (i * step),
      max_height, scene, .5, 0x000000);
    text(start_point + (i * step), max_height + 6, dims[i], scene);

    const y_step = (2 * Math.abs(max_height)) / 5;
    for (let j = 0; j <= 5; j++) {
      let tick_num = ((max - min) / 5) * j;
      text(start_point + (i * step), -max_height + (j * y_step) - 1,
        `- ${tick_num.toFixed(2)}`, scene, '2', 'gray');
    }
  }
}
```

Listing 3.13: Parallel Coordinates: Drawing the parallel axes.

element is highlighted in red in each of the views. All of the code described in this section can be found in the file `hovering.js` inside the project's repository.

```
export function draw_parallel_coordinates(
  pts,
  chart_props,
  dataset_props,
  scene,
  name
) {
  // number of dimensions
  let n = dataset_props.n;
  let min = dataset_props.min;
  let max = dataset_props.max;

  // x position of the first axis. The last one specular w.r.t. the y=0 point
  let start_point = chart_props.start_point;

  // y position of the end of the axis
  let max_height = chart_props.max_height;

  // distance between one axis and the other
  let step = chart_props.step;

  /** this will contain references to the drawn points.
   * Those will be needed by the hovering function to
   * scale their size when needed
   */
  let lines = [];

  let prev_x = start_point + (0 * step);
  let prev_y = scale_linear(pts[0], max_height, min, max);

  for (let i = 1; i < n; i++) {
    let new_x = start_point + (i * step);
    let new_y = scale_linear(pts[i], max_height, min, max);

    lines[i] = draw_line(prev_x, prev_y, new_x, new_y, scene, 0.6);

    // name is associated to the point as an identifier
    // in order for the hovering function to handle related data
    // that need to be highlited togheter
    lines[i].name = name;

    prev_x = new_x;
    prev_y = new_y;
  }
  return { name: name, lines: lines };
}
```

Listing 3.14: Parallel Coordinates: Drawing polylines for data points.

```
const raycaster = new THREE.Raycaster();
const pointer = new THREE.Vector2();

window.addEventListener('pointermove', onPointerMove);
```

Listing 3.15: Interactivity: Initialisation.

```
let canvasBounds =
  renderer.getContext().canvas.getBoundingClientRect();
pointer.x = (
  (event.clientX - canvasBounds.left) /
  (canvasBounds.right - canvasBounds.left)
  ) * 2 - 1;
pointer.y = - (
  (event.clientY - canvasBounds.top) /
  (canvasBounds.bottom - canvasBounds.top)
  ) * 2 + 1;
```

Listing 3.16: Interactivity: Tracking pointer coordinates.

```
raycaster.setFromCamera(pointer, camera);
const intersects = raycaster.intersectObjects(scene.children);

intersects.forEach(element_hovered_now => {
  highlight(element_hovered_now);
});
```

Listing 3.17: Interactivity: Highlighting hovered objects.

```
\begin{lstlisting}
let difference = old_intersects.filter(
  x => !intersects.find(xx => xx.object.name = x.object.name)
);

// unhighlight objects no longer hovered
if (difference.length) {
  reset_intersects(difference);
}

// copy currently hovered objects to old_intersects
old_intersects = intersects.slice();
```

Listing 3.18: Interactivity: Unhighlighting objects.

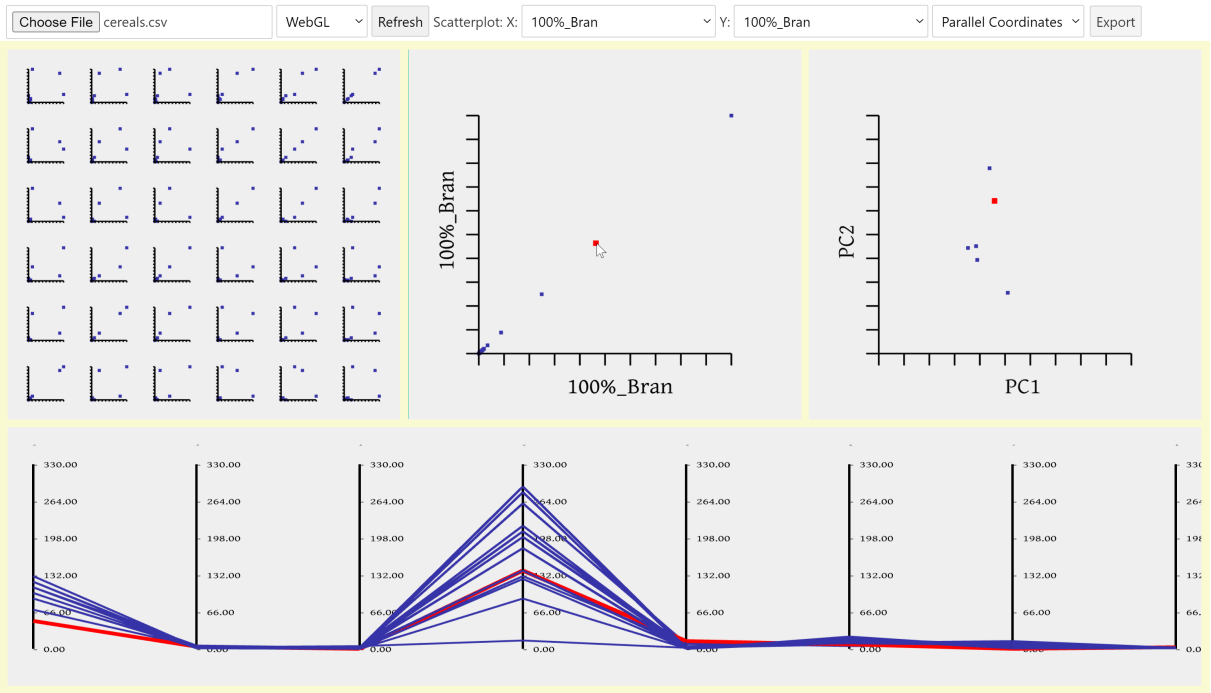

Figure 3.1: 3mvis: Hovering over an element in one visualisation highlights the same element in the other visualisation (brushing). [Screenshot taken by the authors of this report.]

Chapter 4

Concluding Remarks

As a general conclusion, it is certainly possible to create a multidimensional visualisation web application using Three.js in combination with svelte. Svelte makes it beginner friendly to create reactive frontend web applications, and Three.js already supports many useful rendering functions. Importing D3's DSV module makes importing CSV straightforward. SVG export can be done by using the SVGRenderer provided by Three.js, which makes this part also easily approachable. Currently the only problem left over is the size of the files which is pretty large (couple of megabytes). Interactivity between the different visualisations (brushing) is not supported by either svelte or Three.js, so a hand-made solution had to be developed. A Raycaster applied to each plot is a good workaround for this problem. Finally, the scaling of the plots is not easy out of the box, but is achievable by using some resizing functions for each plot in combination with a CSS grid.

Bibliography

bitanath [2022]. *PCA*. 27 Jun 2022. https://github.com/bitanath/pca (cited on pages 12, 14).

Bostock, Mike [2022a]. *d3-dsv*. 12 Jan 2022. https://github.com/d3/d3-dsv (cited on page 2).

Bostock, Mike [2022b]. *D3.js – Data Driven Documents*. 27 Jun 2022. https://d3js.org/ (cited on page 2).

IBM [2021]. *XML serialization*. 2021. https://ibm.com/docs/en/db2/11.5?topic=functions-xml-serialization (cited on page 5).

Microsoft [2022]. *Interface Blob*. 2022. https://microsoft.github.io/PowerBI-JavaScript/interfaces/_node_modules_typedoc_node_modules_typescript_lib_lib_dom_d_.blob.html (cited on page 5).

mrdoob [2022]. *Three.js*. 27 Jun 2022. https://threejs.org/ (cited on page 1).

Node.js [2022]. *Node.js*. 27 Jun 2022. https://nodejs.org/ (cited on page 1).

svelte [2022a]. *svelte*. 27 Jun 2022. https://svelte.dev/ (cited on page 1).

svelte [2022b]. *svelte sandbox*. 27 Jun 2022. https://svelte.dev/repl/hello-world?version=3.48.0 (cited on page 1).