Visualizing the HICANN Wafer

Richard Boell

January 10, 2018

Abstract

The mapping of neuronal networks implemented in PyNN onto the HICANN wafer scale system is done with the software Marocco[1]. Visualizing neuron placement and Layer 1 routing is useful for debugging purposes. Previous visualization tools developed within the group are hard to maintain and do not support dynamic interaction.

In this internship, the groundwork was laid for a web-based software that visualizes the data stored in the `marocco::results` container. The web-based approach allows in principle running the software on a large variety of devices, including tablets. After loading the routing data, a graphical representation of the wafer is dynamically built using the JavaScript library PixiJS[2]. Basic pan&zoom input allows navigating through the software, mouse interaction with individual elements is supported. By default the level of detail is determined automatically based on the zoom-level, but the user can also control manually what elements to visualize.

Throughout the project an emphasis was put on writing maintainable code, that could be easily extended to include more data and possibly feature dynamic visualization in future versions.

1 Methods

1.1 Performance

The first step towards building the software was to figure out what details of the wafer the user might want to see, while considering performance related limitations. A single HICANN chip[3] has 512 neurons, 320 Layer 1 buses (128 vertical buses on the left and the right side of the chip each and 64 horizontal buses between the synapse arrays), 224 synapse drivers and 114688 synapses on the two synapse arrays. Hence, visualizing the details of all 384 HICANN chips on a wafer means drawing around 40 Million synapses and 100 000 buses. This number of elements exceeds the number of pixels on a regular screen, so obviously there is a need to decide which features to draw.

I decided on a zoom level dependent visualization, which means that the user can zoom and navigate through the visualization and details are displayed depending on the zoom level, comparable to Google Maps. Apart from being a necessary step, this holds the opportunity to visualize different properties as will be explained later in detail.

There are a lot of different technologies and libraries to use for visualization in JavaScript. The requirements for wafer visualization purposes are a smooth rendering of a couple 100 000 elements but also fairly easy usage and good documentation, so a possible successor can easily dive into the project. Figure 1 gives an overview over some technologies. I tested the

technology	max elements	comments
HTML5 SVG	<100K	Easy access due to xml format.
HTML5 Canvas	<1M	Cumbersome to implement.
PixiJS library	>1M	Fast rendering using WebGL in 2D. Fairly easy to implement.
ThreeJS library	<100K	Good choice for 3D applications, otherwise unnecessary.

Table 1: Breakdown of the performance assessment of visualization technologies. PixiJS is the most performant option for 2D visualization.

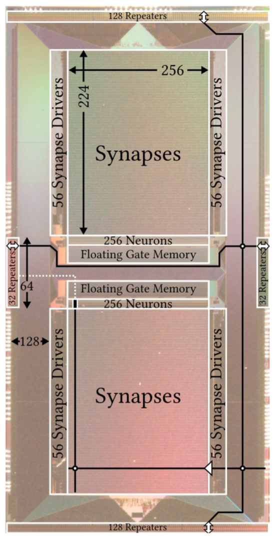


Figure 1: The 5x10mm^2 HICANN microchip. Taken from [1].

performance by implementing mouseover effects and pan&zoom control and then determining the maximum number of elements that still yields a smooth rendering (Figure 2). The JavaScript library PixiJS[2] seems to be the best option for a two dimensional visualization. It supports WebGL rendering and thus can easily handle 1 Million graphic elements. Furthermore the library is well supported and documented and easy enough to use.

1.2 Data Access

The next question at hand was how the software accesses the data from the Marocco results container written in C++, that are to be visualized. Creating a CSV-converted file beforehand to then load with the visualization software would be difficult to maintain. Adding a python wrapping to the C++ code would be a good option, if it was not slow and would not add more dependencies. Apart from using Node.js requests with C++ addons, there is a more elegant way to use the C++ code in JavaScript. Emscripten is a compiler that takes C++ code and outputs JavaScript. Thus, Marocco can be transpiled to JavaScript and used directly in the JavaScript Visualization software. Embind is used to make the functions and and objects available in JavaScript by specifying names for them so they can be used just like the C++ code. The only thing needed is an API in Marocco to access the relevant data. This approach has the huge advantage, that as long as the API stays the same, changes in Marocco will not effect the visualization software.

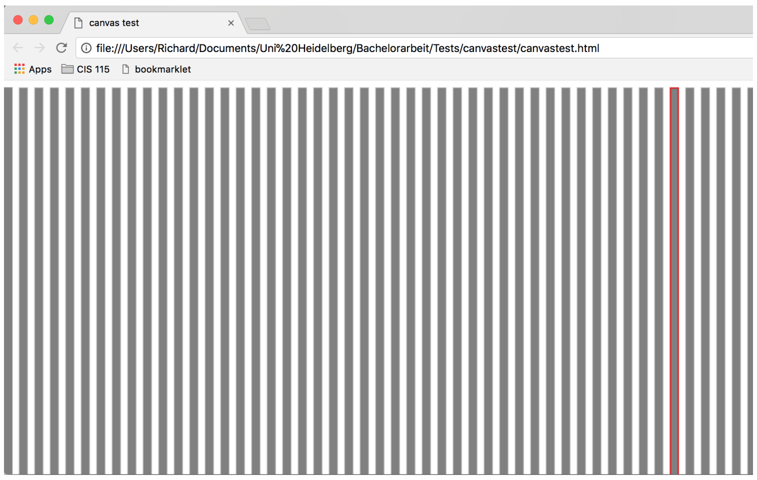

canvas test
file:///Users/Richard/Documents/Uni%20Heidelberg/Bachelorarbeit/Tests/canvastest/canvastest.html
Apps
CIS 115
bookmarklet

Figure 2: An example of a performance test using HTML canvas. A large number of simple stripes was drawn and pan&zoom as well as mouseover effects tested.

```
┌─ build
│    ├─ main.js
│    ├─ main.html
│    ├─ main.css
│    ├─ Marocco
│    └─ libs
│         ├─ pixi.min.js
│         └─ ...
└─ src
     ├─ main.ts
     └─ modules
          ├─ pixiBackend.ts
          └─ ...
```

Figure 3: The filestructure of the software. All TypeScript files are compiled into a single main.js file.

1.3 Maintainability

An important aspect of the project was to write maintainable code. For that reason I switched to TypeScript at some point. TypeScript is a superset of JavaScript maintained by Microsoft. First and foremost it adds static typing so many errors can be avoided right from the start in a large project. TypeScript Modules make it easy to define Namespaces (formerly internal modules) and separate chunks of code. The TypeScript compiler can compile all the .ts files into a single JavaScript file which guarantees both, a better overview for the developer and a small number of files for use in the final software. Figure 3 outlines the filestructure with all the TypeScript files in the src folder and the final software in the build folder.

2 Results

2.1 Features

Starting the software by opening main.html opens up a start screen (Figure 4) where the user can select a network.xml file with the data from the simulation to be visualized. After loading the file, the visualization starts with an overview of the wafer, the wafer is fully visible and centered on the screen (Figure 5). The oval shape of the wafer comes from dynamically calculated HICANN width and height values, to best fit the buses and synapse arrays in the detail-view. Navigation around the wafer works basically like google maps by panning with mouse-clicks and -drags and zooming with the mouse-wheel.

In the overview, the buses and synapse arrays are not displayed in detail. Instead each segment (e.g. vertical left bus) is drawn as a single rectangle in a color representing the number of routes running over this segment. Synapse arrays are omitted altogether, instead the HICANN background is colored to show the number of neurons placed on that HICANN. The number of inputs on each HICANN is graphically represented by a color-coded triangle at the bottom of the HICANN.

Hovering over a HICANN displays its number, enumerated from left to right and top to bottom. Clicking on a HICANN displays its number together with properties in the top part of the info-panel on the right side. Additionally the panel has color-gradients for the properties so the user can see how the colors of the HICANN background for instance correspond to the number of neurons placed on it.

Right below the properties section, the settings section allows the user to display or hide all the elements of one type (e.g. all HICANN backgrounds) at once. For the detailed bus segments this is only possible by rendering the single bus segments as sprites beforehand, as explained in section 2.2.1. In addition the user can select which details to show when zooming into detail-view. The FPS/RAM stats are very useful during development, but might be omitted in the final software. Lastly a complete list of HICANNs can be opened up in the info-panel on the left side. Clicking on a HICANN in the list will animate the stage to move the respective HICANN into the center of the screen. The user can also expand each list-item to list all the elements and select which ones to display. This gives the user complete control over what to display on each HICANN if the automatic mode is not sufficient. On the downside, the user is responsible for displaying only the number of elements that will not crash the software.

The automatic zoom automatically draws more details when the user zooms in. Right at the beginning, the software determines the zoom-scale where a HICANN almost fills the screen. Once the user zooms over this threshold the detail-view starts (Figure 6). Instead of the colored elements representing the number of neurons or routes, a more realistic HICANN is drawn. All the vertical and horizontal buses as well as the synapse-arrays are now visible. The synapse-arrays are represented by a grid of 224 x 256 small rectangles.

This large number of elements leads to unpleasant visual effects originating from the quantization to fit the pixels. Video games deal with this issue by using antialiasing. Unfortunately, antialiasing cannot be used on graphics objects with the WebGL renderer. However, PixiJS can render textures from graphics objects with specified resolution and antialiasing enabled. This not only reduces unwanted effects (autorefimg:antialias) but also comes with lower computational cost. Therefore, the automatic zoom effectively has two detail-stages. The first one creating the detailed graphics objects and displaying the textures, the second one displaying the original graphics objects for sharp edges at very high zoom levels. When a HICANN is moved out of scope during detail-view, the neighboring HICANN will be displayed in detail and the details of the previous HICANN removed again.

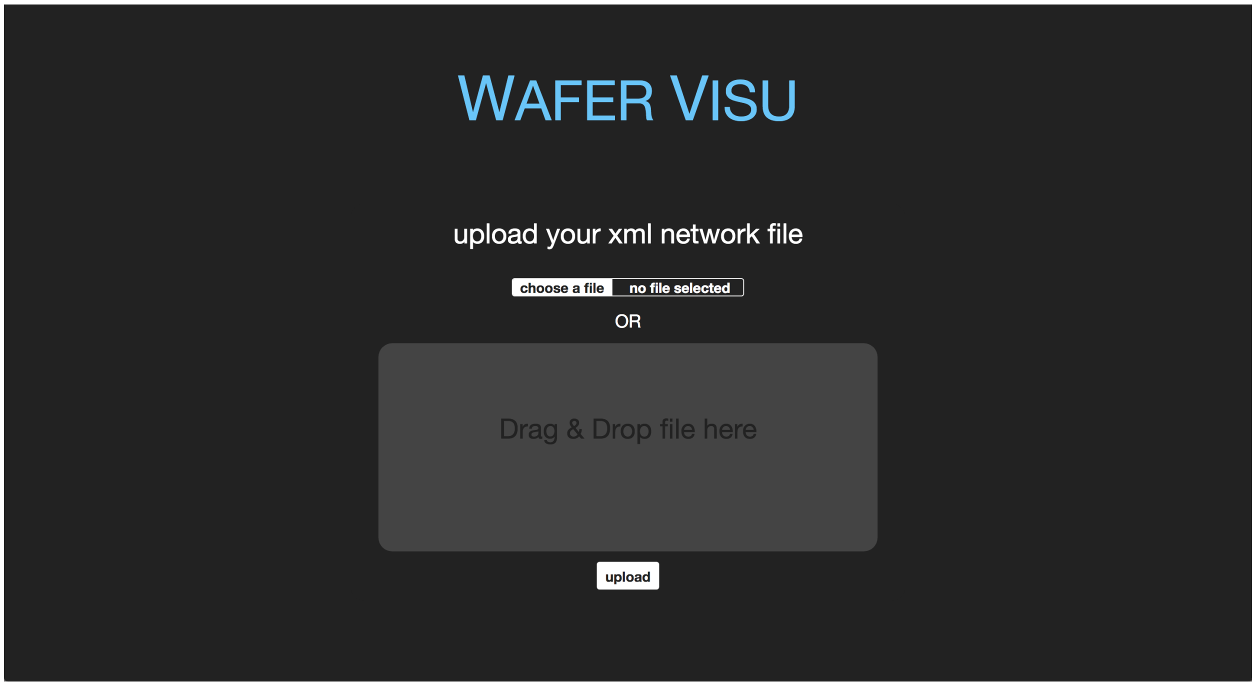


Figure 4: The start screen allows the user to upload a network as a xml-file via the file browser or drag-and-drop upon starting the software.

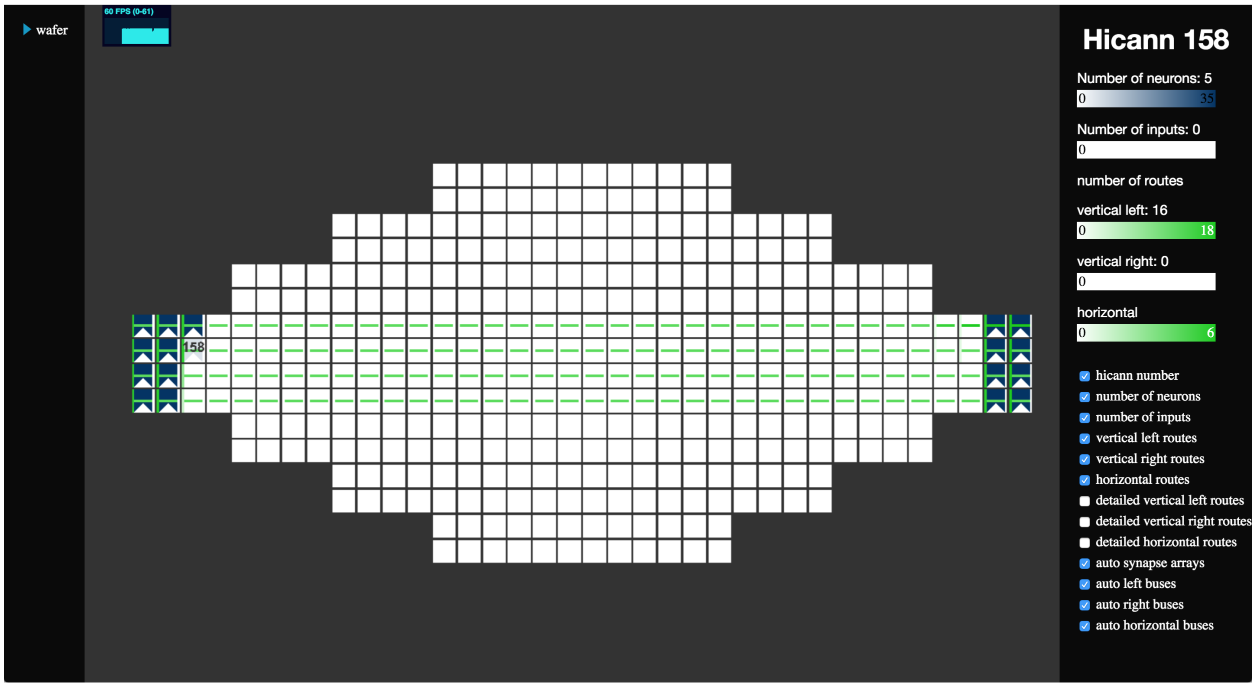
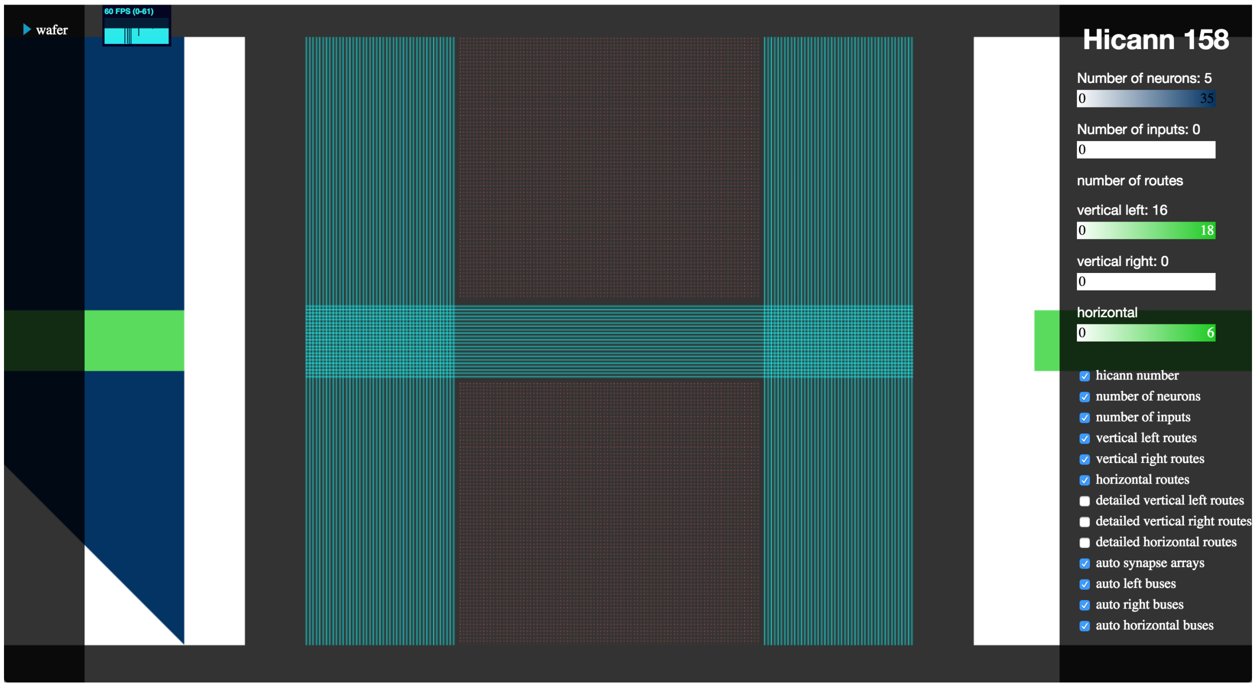


Figure 5: Overview of the visualization. The single buses are not drawn, instead simple color-coded rectangles represent the number of routes running over that segment. The info-panels show information about a HICANN when selected. All HICANNs are listed in an expandable list in the left panel. All graphical elements can be (un-) selected here.



Hicann 158

Number of neurons: 5
0 35

Number of inputs: 0
0

number of routes

vertical left: 16
0 18

vertical right: 0
0

horizontal
0 6

- [✓] hicann number
- [✓] number of neurons
- [✓] number of inputs
- [✓] vertical left routes
- [✓] vertical right routes
- [✓] horizontal routes
- [ ] detailed vertical left routes
- [ ] detailed vertical right routes
- [ ] detailed horizontal routes
- [✓] auto synapse arrays
- [✓] auto left buses
- [✓] auto right buses
- [✓] auto horizontal buses

Figure 6: The detail-view is automatically loaded when the user zooms past a certain threshold. All Buses and the full synapse array are drawn now.

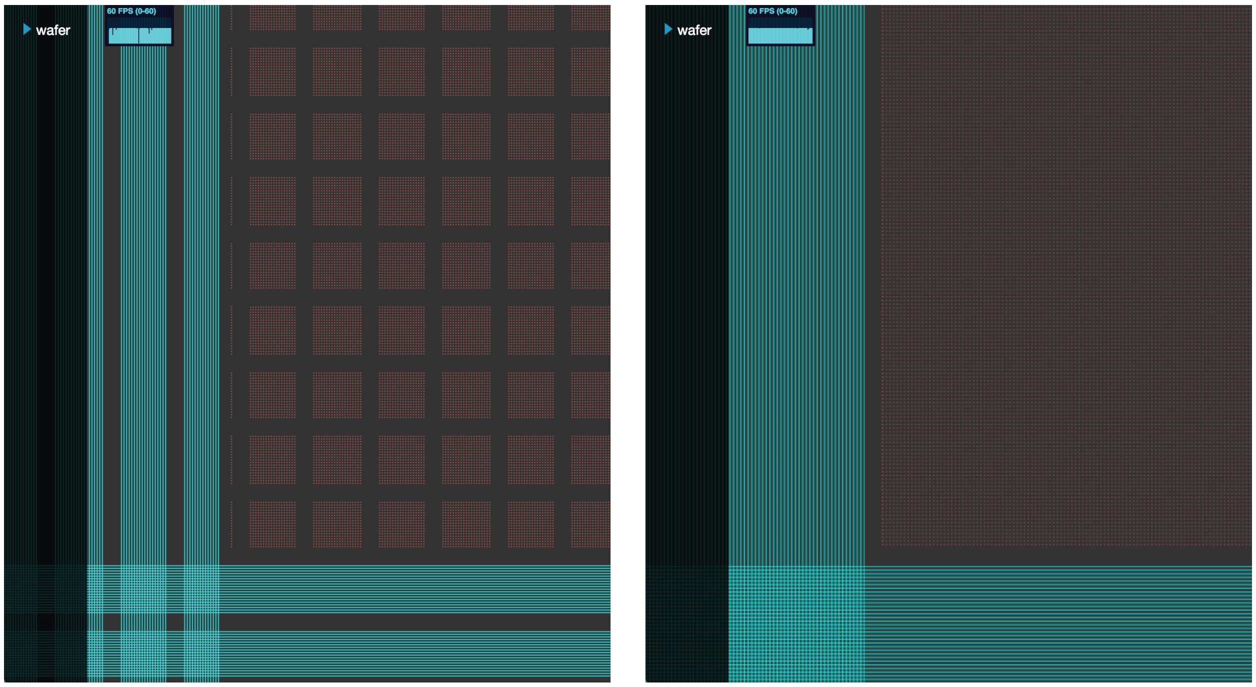


Figure 7: Detail-view with (right) and without (left) antialiasing. The large number of small elements leads to undesirable visual effects when the elements are too small to entirely fit on a pixel. Antialiasing rendered images of the graphics objects with smooth edges minimizes those effects.

2.2 Code details

JavaScripts `FileReader()` object allows to process the text file uploaded via the file-browser or drag and drop. To use the file later with Marocco, it is written into Emscriptens virtual file system, accessible via the object `FS`. After processing the file, the main visualization software is started by invoking the `startVisu()` function.

As mentioned before, writing maintainable code was one of the major goals of this internship. For that reason the code was separated into namespaces as the project grew larger.

2.2.1 PixiBackend

The pixiBackend namespace includes all the code, that directly uses PixiJS such as drawing graphic elements and navigating through the visualization. It exposes a few simple functions to use in the rest of the software. The PixiJS environment is initialized by creating a HTML5 canvas and setting up the container (`PIXI.Container()`) structure to later hold all the different graphic elements (i.e. bus segments, HICANN backgrounds, ...). Drawing simple shapes is straightforward in PixiJS. First, a `PIXI.Graphics()` instance is created, the drawing path and colors specified, and then it is stored in one of the containers. PixiJS even has built-in functionality to add interactive mouse-effects to those graphics objects. However, creating a new graphics object for each element comes with great computational cost, storing multiple elements (e.g. a whole synapse-array) as one graphics object is much more efficient. The `drawRectangles` function takes arrays of x, y, width and height values and draws all those rectangles as one graphics object. The downside is, that mouse interactivity can not be added to the single elements in the graphics object. But storing the element positions in an external array and looping over it for mouse-over and mouse-click effects is still much more performant than creating single graphics objects.

```
function drawRectangles(container, xValues, yValues,
      widthValues, heightValues, color) {

  // create PIXI Graphics instance
  const rectangles = new PIXI.Graphics();

  // loop through passed arrays and draw rectangles
  for (let i=0; i<xValues.length; i++) {
    rectangles.beginFill(color);
    rectangles.drawRect(xValues[i], yValues[i], widthValues[i],
        heightValues[i]);
    rectangles.endFill();
  };

  // add graphics object to container
  container.addChild(rectangles);
}
```

Replacing graphics elements with sprites yields even better performance and minimizes unwanted effects through antialiasing. This in done in PixiJS by generating a texture from the graphics object with specified resolution and storing the texture in a `PIXI.Sprite()`.

Panning and zooming is implemented by manipulating the `scale` and `position` properties of the `stage.transform` object, where stage is the top-level container that contains all sub containers. Changing the stages position effectively moves the whole visualization.

2.2.2 Wafer

The wafer namespace stores the HICANN positions and all the data to be visualized. `loadOverviewData()` is currently used to load the data from the network.xml file selected at the beginning. The idea is, that multiple functions load chunks of data only when needed, once enough data is implemented. `loadOverviewData()` first builds Marocco from the xml file, then `marocco.properties(hicann)` are stored in the `wafer.hicannProperties` object for all HICANNs. `northernhicann()` et cetera determine the indices of the neighboring HICANNs if existent.

```
function loadOverviewData(networkFilePath) {
  // build marocco from uploaded network.xml file
  let marocco = new Module.Marocco(networkFilePath);
  ...
  // read properties from marocco
  for (let i=0; i<384; i++) {

    // create new enum, hicann and properties instances
    let enumRanged = new Module.HICANNOnWafer_EnumRanged_type(i
       )
    let hicann = new Module.HICANNOnWafer(enumRanged);
    let properties = marocco.properties(hicann);

    // hicann position
    this.hicanns[i] = {};
    this.hicanns[i].x = hicann.x().value();
    this.hicanns[i].y = hicann.y().value();

    // hicann properties
    this.hicannProperties[i] = {};
    this.hicannProperties[i].has_inputs = properties.has_inputs
       ();
    this.hicannProperties[i].num_buses_horizontal = properties.
       num_buses_horizontal();
    ...
  }
}
```

2.2.3 Detailview

The detailView namespace is probably the most complex part of the code, as it contains all the logic that takes care of dynamically creating and removing elements in response to panning and zooming. First of all there are methods like `hicannClosestToCenter()` to determine the part of the wafer, that is in the users scope. To start the detail-view, when the user zooms in far enough, `detailView.start()` is called. For the HICANN in scope, the detailed buses and synapse arrays are created but graphics objects are hidden, making use of PixiJS' `object.visible` property. Additionally all the positions of the synapses on the synapse arrays are stored to enable mouse interactivity with those objects. The HICANN number in the visualization is hidden and instead the properties are displayed in the info-panel on the right side of the screen. When the user passes the second zoom-level threshold, the sprites are hidden, and the real graphics objects rendered visible.

```
function start(newHicann) {
  // remove all Hicann numbers
  // show properties of current Hicann in sidebar

  ...

  // remove overView Elements for that HICANN
  this.removeOverview(newHicann)

  ...

  // draw left buses and hide the graphics objects
  this.drawBusesLeft(hicannPosition);

  // display sprites for left bus
  pixiBackend.container.hicannBusesLeft.children[newHicann].
       visible = true;

  ...
}
```

As explained before, the software detects automatically, when the user moves the currently displayed HICANN out of scope during detail-view and loads the detail-view of the neighboring HICANN (if existent). To prevent unwanted switches and give the user some space to move around without experiencing annoying changes of the detailed HICANN, the switching does not occur when the center of the display is exactly between two HICANNs but half a HICANN width later (controlled with the `detailView.edge` property). When zooming out past the detail-view threshold, the detailed graphics objects and sprites are removed, the synapse position arrays emptied and the overview restored again.

```
function recoverOverview(hicannIndex) {

  // set overview elements to visible
  pixiBackend.container.hicannNumberText.visible = true;
  pixiBackend.container.hicannBackgrounds.children[hicannIndex
       ].visible = true;
  ...
}

function removeDetailView(hicannIndex) {

  // remove graphics objects for detailed elements
  pixiBackend.removeChild(pixiBackend.container.
       detailBusesVerticalLeft, 0);

  // check if detailed bus segment was manually selected
  if (this.displayedBuses.left.indexOf(hicannIndex) === -1) {
    pixiBackend.container.hicannBusesLeft.children[hicannIndex
       ].visible = false;
  };
  ...

  // reset synapse array position arrays
  this.synapseArrayOne.x = [];
  ...
}
```

This automatic zoom mode was the first mode implemented in the software and the ability to select manually the exact elements to be displayed followed later. So when adding this functionality, unexpected behavior when selecting some elements manually and using the automatic zoom occurred. First attempts to fix this issue led to a complex and error-prone code. Therefore, following the internship, two completely separate display modes will be implemented. The auto mode that automatically selects the details to be displayed according to pan and zoom action, without the ability to take any influence on that. And secondly the manual mode that disables the automatic functionality and leaves it completely up to the user to select the elements and detail-level to be displayed. However, in this mode it is the users alone responsibility to limit the number of detailed elements, otherwise the software will crash.

2.2.4 Global space

Apart from the namespaces described above, there are a number of global functions and objects. All the configuration tasks that need to be performed before the actual start of the visualization software are implemented in global space at this point. Also the dynamic setup of the HICANN tree (left panel in the GUI) is done inside a global function: All the items in the list are HTML `<li><\li>` "list item" tags in an unordered list (`<ul></ul>`). Inside those list-items, a hidden checkbox is controlled by a label that is styled with the little triangle as background to indicate whether the list is opened or not. Another label (or button in some cases) shows the name of the list item. If the list-items themselves should contain multiple sub elements, they are unordered lists again containing their child elements as list items. This setup function for the whole tree is unfortunately very lengthy and confusing, but so far I did not find a way to simplify it significantly.

The overview (i.e. the functions to draw the elements in the overview) is implemented in global space as well and so are all the event handlers for mouse, keyboard and checkbox interactivity.

3 Next steps

As a first step towards optimizing the code, TypeScript modules can be used to further separate codes and create independent, well maintainable units. The mentioned implementation of two different modes to either automatically view details or select everything manually will create a better and more intuitive user interaction and makes it easier to extend the code.

Probably the next goal for this visualization is to implement detailed routes, a crucial feature for debugging. The user should be able to click on bus segments in the detail-view and get all the routes running over that segment displayed. Alternatively a list of all routes should be provided with the ability to click on them and display them on the wafer. It could be potentially useful to also list all routes that run over a route segment as well as all the routes that start or end at on HICANN in the already existing elements list in the left info-panel.

Once a solid foundation is implemented, it will be fairly easy to add further features. When creating a population in PyNN, an unambiguous id is assigned to it. A nice feature would be to highlight those HICANNs hosting the population and visualizing the connection between them. Apart from including further static data, a dynamic visualization of spikes, weights and membrane potentials would be great for demonstration purposes. One could reproduce the dynamics during a simulation in slow motion.

A rather extensive addition would be to visualize the biological model as well and draw a connection between them. A complete restructuring of the user interface would be necessary, though. A three dimensional representation of the biological model using a different JS library should be considered, provided that computational power is sufficient.

References

[1] Sebastian Jeltsch. “A Scalable Workflow for a Configurable Neuromorphic Platform”. PhD thesis. Universität Heidelberg, 2014.

[2] Mat Groves and the PixiJS team. *PixiJS - The HTML5 Creation Engine*. 2017. URL: `http://pixijs.download/release/docs/index.html`.

[3] J. Schemmel et al. “A wafer-scale neuromorphic hardware system for large-scale neural modeling”. In: *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. 2010, pp. 1947–1950. DOI: `10.1109/ISCAS.2010.5536970`.