Department of Physics and Astronomy
University of Heidelberg

Bachelor Thesis in Physics
submitted by

Richard Boell

born in Stuttgart

April 2018

Visualization of Mapping and Routing of the BrainScaleS System

This Bachelor Thesis has been carried out by Richard Boell
at the
Kirchhoff Institute for Physics
Ruprecht-Karls-Universität Heidelberg
under the supervision of
Prof. Dr. Karlheinz Meier

Abstract

To emulate neural networks on the BrainScaleS neuromorphic wafer-scale system, the biological network first needs to be translated into a corresponding hardware configuration. The ability to visualize this configuration process and particularly the map-and-route step is important for software debugging as well as commissioning of the system. Additionally, experiment monitoring benefits from visualization of dynamic properties such as hardware usage and spike rates.

This thesis implements an interactive browser-based visualization of individual BrainScaleS wafers that allows a deep understanding of network configurations. The visualization provides an intuitive overview of the utilization of different parts of a wafer by implementing colormaps. Further, routing across the wafer can be studied in detail. The software is designed to be highly maintainable and easily extendible. Due to the large number of elements on a wafer, careful considerations about the level of detail and the method of drawing are inevitable. In a benchmark, drawing methods are examined and conclusions about further improvement in terms of performance are drawn.

The software presents the foundation for an adaptable and versatile visualization of the BrainScaleS system.

Zusammenfassung

Um neuronale Netzwerke auf dem neuromorphen BrainScaleS System nachzubilden, muss das biologische Netzwerk zunächst in eine entsprechende Hardwarekonfiguration übersetzt werden. Die Möglichkeit, diesen Konfigurationsprozess und insbesondere den map-and-route Schritt zu visualisieren, ist sowohl für die Untersuchung der Software auf Fehler, als auch die Inbetriebnahme des Systems wichtig. Zusätzlich ist die Visualisierung von dynamischen Eigenschaften wie die Verwendung der Hardware oder Spike-Raten für die System-Überwachung von Vorteil.

Diese Arbeit realisiert eine interaktive, browserbasierte Visualisierung einzelner Wafer des BrainScaleS Systems, die ein tief gehendes Verständnis von Netzwerk Konfigurationen ermöglicht. Eine farbkodierte Übersicht verschafft einen intuitiven Einblick in die Verwendung verschiedener Wafer Komponenten. Routen können außerdem im Detail verfolgt und untersucht werden. Die Software wurde gezielt wartungsfreundlich gestaltet und ist einfach zu erweitern. Aufgrund der großen Anzahl an Elementen auf einem Wafer müssen die Detailstufe und Zeichenmethode sorgfältig erwogen werden. In einem Benchmark-Test werden verschiedene Methoden zum Zeichnen von Elementen untersucht und Schlussfolgerungen über mögliche Leistungssteigerungen gezogen.

Die vorgestellte Software bildet die Grundlage für eine anpassungsfähige und vielseitige Visualisierung des BrainScaleS Systems.

Contents

1. Introduction

Simulating the brain has the potential to give valuable insights into learning and development of the brain. Simulation of large-scale networks on traditional computer clusters is time- and energy-consuming. Dedicated neuromorphic systems could offer an advantage in this respect. Emulating parts of the behavior of a biological neuron with electrical circuits can be proven to be many orders of magnitude more efficient in terms of power use and also timewise [1]. Event-driven neuromorphic systems are built upon the principle, that spikes (events) govern the activity of the chip. Instead of following a clock, neurons react to external inputs or signals from other neurons, communicated via synapses [2].

Neuromorphic hardware is being developed in both industry and academia. IBM's TrueNorth chip succeeds at a very energy-efficient implementation of learning algorithms but lacks flexibility [3]. Just recently Intel introduced Loihi, a neuromorphic manycore chip that implements a spiking neural network [4]. The BrainScaleS physical model system [5] implements a highly parallel architecture and operates 10^3 to 10^5 times faster than biology. Its analog neuron and synapse circuits are designed to emulate biological spiking behavior. Currently, the system features 20 silicon wafer modules. High configurability allows the emulation and scientific exploration of a variety of models.

Software models of abstract neural networks are translated to actual neurons and synapses on the BrainScaleS system. The goal of this thesis was to develop a visualization of hardware parameters and routing on the wafer. The visualization provides an intuitive overview of the wafer utilization and can be primarily used as a debugging tool for examining the routing. It is important to develop a highly maintainable software. Due to the modular nature of the code, the presented software can be used as a stepping stone for more extensive and dynamic visualizations.

Johann Klähn, Eric Müller and Sebastian Schmitt cooperated to make the hardware configuration data container available for the visualization.

Section 2 presents the neuromorphic hardware and the software stack supporting the system. In section 3, technologies used for the visualization are introduced. Section 4 describes the developed visualization software, starting with an overview of the application features. The structure and implementation is then explained with code samples and a benchmark of the graphics library is presented. Finally, an example of how to extend the software is given.

2. Materials and Methods

2.1. BrainScaleS Hardware System

Beginning with the Spikey chip [6] and continued in the course of the FACETS [7] and BrainScaleS [5] projects and the ongoing Human Brain Project [8], a large-scale mixed-signal neuromorphic hardware has been developed by the Electronic Vision(s) group in cooperation with the Technische Universität Dresden and the Fraunhofer-Institut für Zuverlässigkeit und Mikrointegration IZM Berlin. The hardware dynamics are accelerated with a speed-up factor of around 10^3 to 10^5 compared to biological timescales. The BrainScaleS system (Figure 1) includes 20 wafer modules mounted in industry-standard racks. Those 20 cm silicon wafers were not diced into individual chips, but kept as a whole. A single wafer features 384 High Input Count Analog Neural Network (HICANN) microchips, manufactured in 180 nm complementary metal-oxide-semiconductor (CMOS) technology. The HICANNs are grouped into reticles of 8 chips. In a post-processing step, a metal layer is added on top to connect the reticles. This wafer-scale integration makes super-dense connections for the on-wafer network possible [9][10]. Defects from manufacturing cannot be sorted out, so the routing algorithm that configures the hardware for an emulation needs to find solutions to work around these areas. Figure 2 shows such a wafer with 384 HICANN chips.

2.2. HICANN Microchip

The central building block of the wafer is the 5 mm $\times$ 10 mm HICANN microchip. Each HICANN has 512 neuron circuits, arranged in two rows of 256 circuits in the center of the chip (Figure 2). In the current version, the neuron circuits implement the Adaptive Exponential Integrate-and-Fire (AdEx) model [11], emulating firing patterns found in biology [12]. They are configured by setting a number of analog and digital parameters prior to the experiment. Spiking events are passed to the neurons via synapse arrays, located at the top and bottom of the chip. Combining

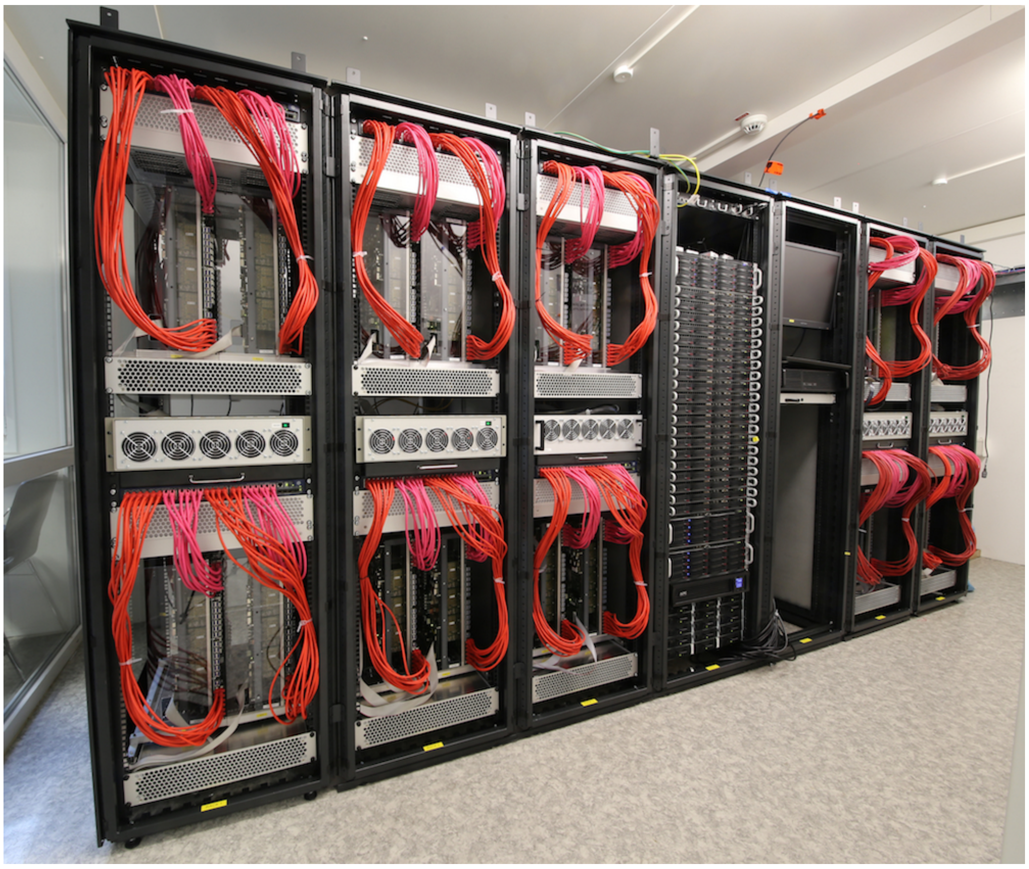


Figure 1: The BrainScaleS neuromorphic hardware including 20 wafer modules mounted in industry-standard racks. The red ethernet cables connect the modules to the host computer.

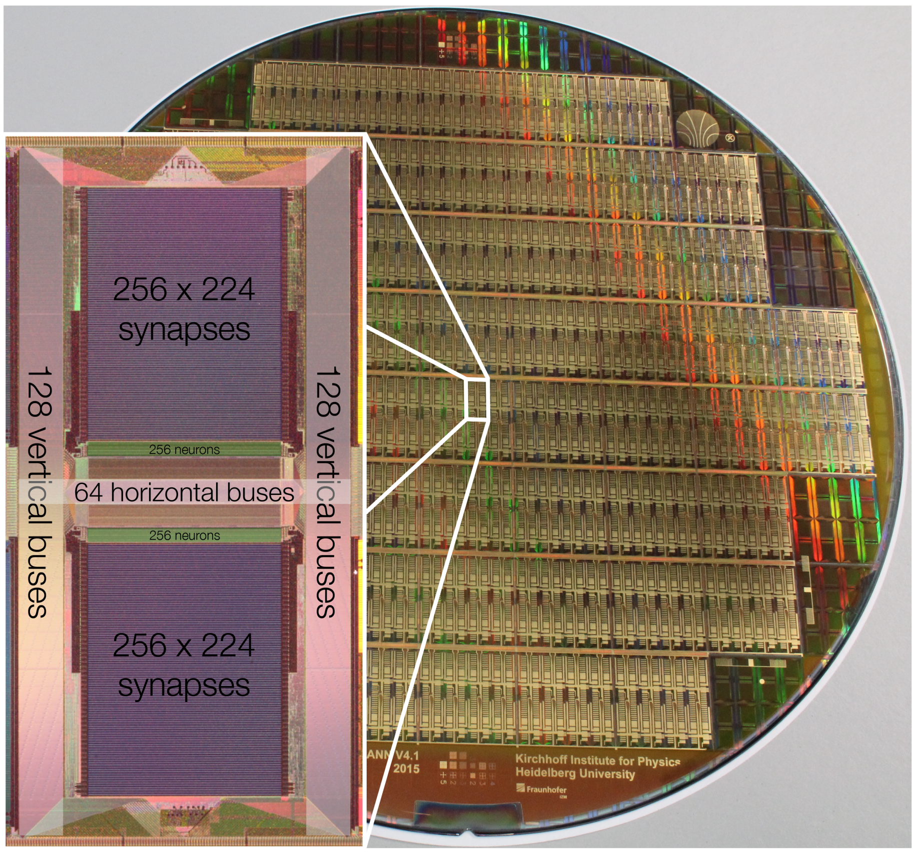


Figure 2: Photograph of a full wafer with 384 HICANNs neuromorphic chips and the metal communication layer added during post-processing. The enlarged image shows a HICANN without the metal layer. The vertical Layer 1 (L1) bus lanes on the left and right side of the chip as well as the horizontal L1 buses in the center of the chip are clearly visible. The two large rectangles are the arrays of 256 by 220 synapse circuits. Between the synapse arrays and the horizontal bus are two arrays of 256 neuron circuits each.

up to 64 neurons (32 from each neuron row) allows up to 14 080 synaptic inputs. Grouping multiple hardware neurons together to form one biological neuron reduces the total number of neurons.

2.3. Communication Networks

Post-processing of the silicon wafer adds a wafer-wide high-bandwidth network between all the HICANNs. L1 routes on each HICANN include 64 horizontal and 2×128 vertical buses, forming together an H-shape as can be seen in Figure 2.

Connectivity between the chips is established via repeaters between the L1 buses of neighboring HICANNs. Crossbar switches connect the vertical and horizontal bus segments and route the signal to the target chip. Additional switches on the vertical buses navigate the signal towards the synapse arrays where they are received and processed by synapse drivers. The two-dimensional synapse arrays of 256×220 synapse circuits decode the event signals together with the synapse drivers and input the signal into the target neuron circuits. Spiking events from the neurons are fed into the merger tree and processed together with signals from the on-chip background generators as well as external L1 signals. The merged signals are injected into the horizontal L1 buses in the center of the chip to target the next neuron. Merging the signals allows to use fewer connections of the L1 bus network.

Apart from L1 routing, the wafers need to communicate with outside systems. During the emulation, external spike signals are inserted into the network and activity is recorded. Connectivity with the host computer is established using field-programmable gate arrays (FPGAs) via Gbit-Ethernet, integration of EXTOLL technology is currently being evaluated [13]. The speed-up factor of the hardware system created the need for high-bandwidth connections and data buffering in the FPGAs.

2.4. Software

A number of software packages developed for the BrainScaleS system provide emulation setup and interaction with the hardware. The Python-based application programming interface (API) PyNN [14] is a language designed for modeling of neural networks and experiment protocol. The library includes a number of standard models for neurons, synapses and plasticity and supports different software simulators as well as the SpiNNaker [15] and BrainScaleS [5] neuromorphic hardware. PyNN allows high-level modeling of networks (using e.g. populations of neurons) but also provides access to details of single neurons and synapses.

After describing the network in PyNN, the "biological" network needs to be translated into the configuration of the neuromorphic hardware. This "mapping" is achieved by the C++ software Marocco [16][17]. Marocco is designed in a feed-forward approach, dividing the mapping task into small natural steps, that are solved consecutively. In the first step, the neuron placement, one or more hardware neuron circuits are combined to form a model neuron. In the next step, the merger tree is configured to merge signals from multiple neurons and external sources. The L1 network is configured to route the signals across the wafer and finally synapse drivers and synapse arrays are prepared. Marocco tries to find solutions for a large variety of neuronal networks while still finding configurations that resemble the original network as closely as possible. With increasing network size, this becomes harder. Additionally, hardware defects have to be taken into account all the while. Therefore, besides offering an automatic mapping algorithm, Marocco allows the user to manually add constraints helping to find the best configuration result.

3. Software Framework

Developing a web-based visualization brings a few advantages. New technologies are quickly adopted and supported on multiple operation systems and devices. Cross platform development even for mobile devices is possible. Due to the widespread use of web technologies, long-term support is guaranteed and a variety of well maintained libraries are available. The standard web development technologies HTML, CSS, and JavaScript [18] are comparatively easy to implement but still yield good performance.

3.1. Visualization Library

Scalable Vector Graphics (SVG)) is an XML based vector description of graphic elements. SVGs can be easily integrated into web applications and have the advantage of being easily manipulable after creation due to their representation as XML objects. In 2014, the canvas element (`<canvas></canvas>`) was introduced with HTML 5. It is heavily used in today's web applications for drawing graphics and displaying videos and is supported by all major browsers [19][20]. There are different methods to draw graphics on a canvas element, SVGs can be embedded and WebGL [21] makes use of the canvas environment as well. A large number of JavaScript libraries, built on these HTML 5 technologies, are available, serving different purposes in drawing graphics. Often times it is a trade-off between easy implementation and rendering speed.

Benchmarking The huge number of small elements on the wafer sets clear limitations as to which library can be used and also makes it necessary to think about how detailed the visualization should be. Table 1 gives an overview over the number of elements on the wafer. The total number of around 40 million synapses exceeds the number of pixels on a regular screen, even 100 000 L1 buses are probably too much too view at once since they should be represented by multiple pixels each and have space around them. However, to estimate the limitations from the software side, a

element	total number on wafer
neurons	196 608
L1 buses	122 880
synapse drivers	84 480
synapses	43 253 760

Table 1: An overview over the number of elements on a full wafer. There are more synapses on a wafer than pixels on a regular screen.

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 2: Different technologies for visualization were compared in a brief benchmarking. PixiJS allows high performance due to hardware accelerated rendering using the graphics processing unit (GPU).

simple benchmarking is performed.

For plain HTML5 SVG, HTML5 Canvas as well as the JavaScript libraries PixiJS [22] and ThreeJS [23], a short test code was written, drawing a number of lines on the screen and implementing mouseover as well as pan & zoom effects. Performance depends on the utilized hardware as well as browser version and operating system, but the results can still be used for comparison.[1]

The number of lines was varied and the maximum number that still yields smooth rendering (Table 2) was manually evaluated. Even though SVGs would be a very convenient choice, the low maximum number of elements would limit the visualization too much. Drawing directly on the HTML5 Canvas allows up to almost 1 million elements, but is more difficult to implement. Both PixiJS and ThreeJS are JavaScript libraries that use WebGL, allowing GPU accelerated high-performance

[1]The tests were performed using Chromium Version 57.0.2987, running on Debian 8.10. Hardware specifications: 16 GB RAM, 256 MB VRAM. The test code is available under `https://brainscales-r.kip.uni-heidelberg.de/projects/marocco/wiki/javascript-visu`.

graphics rendering. PixiJS can easily handle over 1 million elements but is limited to two-dimensional applications. ThreeJS is a great choice for three-dimensional visualizations but is therefore limited to less than 100 thousand elements. In conclusion, PixiJS seems to be the best library for the purpose of a two-dimensional visualization with a high number of elements but no complicated shapes or the need for exceptional visual effects. A more in-depth benchmarking of the PixiJS library is presented in section 4.3

The PixiJS Library PixiJS supports WebGL rendering but is not limited to it. Some browsers still do not support WebGL, in which case PixiJS uses the slower canvas renderer as a fallback option. After initialization, objects can be created and stored in `PIXI.Container()` objects. Containers can be nested to build something comparable to a folder structure in a filesystem. However, creating too many containers slows down the rendering process. Each container has a `children` object that holds the substructure. Containers have a member `transform` with position and scale properties that are used to effectively move all the children in that container and create the zooming interaction. To actually see something on the screen, the `PIXI.(WebGLRenderer | CanvasRenderer).render(PIXI.Container())` method is called on the container to be drawn. The `PIXI.Graphics` class (will be referred to as graphics object) is used to draw primitive shapes such as rectangles or circles with specified fill- and border-style. Storing many shapes as part of a single graphics object greatly enhances performance as opposed to creating a new instance of the `PIXI.Graphics` class for each shape. It has to be noted though, that active areas for mouse effects can only be defined on the whole graphics object. Another way to render graphic elements is the `generateCanvasTexture()` method that transforms graphics objects into textures of a specified resolution. These textures can then be used to create a sprite that is rendered on the screen like a normal image. The limited resolution leads to pixelated graphics when viewed in close detail.

3.2. Data Input

When emulating a network, the hardware configuration determined by Marocco can be archived in a file using the `boost::serialization` library [24]. The `marocco::results` container is traversed recursively and specified properties are saved for example as binary data, text data or XML. In the following this hardware configuration file will be referred to as `marocco::results` file. There are multiple ways to load the data from this file into the JavaScript application. An obvious solution would be to write a script that outputs the data in a specific format and saves it for example as a comma-separated list. This file could then be further processed in the JavaScript software, but the intermediate step makes the solution prone to errors (cf. previous visualization tools). In fact there is already an API implemented in Marocco to process the data from the `marocco::results` file via `boost::serialization`, so it seems to be unnecessary to write custom code for writing and reading the configuration data again. The question is, how can the C++ API be accessed with the JavaScript application. Possible options would be to either add a python wrapping to the C++ code or use Node.js requests with C++ add-ons, but there is a faster and more elegant way to use the already existing mechanisms.

Emscripten [25] is a compiler that takes C/C++ source code or LLVM [26] bitcode and outputs JavaScript. The Clang converter is used to convert C/C++ files to LLVM bitcode, which is then compiled by emscripten's LLVM backend to highly optimized JavaScript code. Comparable to `boost::python` [27] for Python, Embind is used to create a register that exposes the C++ functions and classes to the JavaScript code by defining their names. The example below makes the Marocco class of the `marocco::results` container available, the constructor as well as the functions "load", "save" and "properties" are now accessible via JavaScript.

```
EMSCRIPTEN_BINDINGS(marocco_results)
{
  emscripten::class_<marocco::results::Marocco>("Marocco")
    .constructor<>()
    .function("load", &marocco::results::Marocco::load)
```

```
        .function("save", &marocco::results::Marocco::save)
        .function("properties", &marocco::results::Marocco::
            properties)
      ;
      ...
  }
```

The advantage of using emscripten is that, apart from the bindings, no new functionality is needed. Marocco can be transpiled to JavaScript automatically with every build of the software stack and used directly in the visualization software. Marocco can be extended to make the desired data accessible via an API, and as long as this API stays the same, changes will not effect the visualization software.

3.3. Tools

TypeScript Javascript is an untyped language which means, that the data type (e.g. string, number) is not explicitly defined, but figured out by the JavaScript engine at runtime. Browsers use just-in-time compilers that include both interpreter and compiler. When a part of the code is executed multiple times (e.g. in loops), this part is compiled and eventually optimized to increase performance. Omitting type definitions leads to very clear looking code, but can quickly cause unexpected errors, especially in larger projects.

TypeScript [28] is a superset of JavaScript maintained by Microsoft, that adds first and foremost static typing to JavaScript. This means, that JavaScript code can be used in a TypeScript file (.ts) and works just fine, but optionally the type can be specified in which case the TypeScript compiler will throw an error as soon as wrong data types are assigned.

```
// declare variable of type number
let myVar: number;

// assign string to that variable
myVar = "I am a string"; // error
```

Another benefit of using the TypeScript compiler is, that multiple .ts files can be compiled into a single .js file. Large projects can be separated into different parts, but in the compiled end version of the software, only one file needs to be

```
|-doc
|-src
: |-main.ts
: |-modules
: : |-wafer.ts
: : |-...
|-build
: |-main.html
: |-main.js
: |-main.css
: |-img
: |-libraries
: : |-pixi.min.js
: : |-...
: |-Marocco.js
: |-routes.json
: |-...
```

Figure 3: Typescript allows compiling all the TypeScript files in the src folder into a single JavaScript file (main.js). The complete file structure can be found in the appendix.

included. Figure 3 gives an overview of the resulting file structure. The TypeScript compiler also allows setting a target version for the JavaScript code, so that the most recent features can be used without limiting the software to work only on the latest browser versions. Using TypeScript namespaces, the code can be organized to avoid problems caused by globally declared variables.

jQuery jQuery [29] is a library that brings additional features mostly for DOM interaction and animations. HTML elements are selected via `$("selector")` and can be easily manipulated. Below is an example for implementing a mouseover effect on the HTML tag `<div id="myDiv"></div>`.

```
$("#myDiv").mouseover( () => {
  console.log("mouse is over div");
});
```

The extension jQuery UI [30] is a set of user interface (UI) interactions that makes it easy to implement the most common UI items such as draggable and resizable windows. Both libraries perform tasks that could also be implemented in plain JavaScript and CSS, but make it a lot easier. jQuery was used in the visualization to dynamically build and manipulate the user interface.

4. Results

When a neural network is built with PyNN, Marocco tries to find a hardware configuration that resembles the network as closely as possible. Going through the configuration parameters in a text file can be very cumbersome, for large networks it becomes rather impossible. A visualization of the hardware and the configuration could save time and provide a better overview of the utilization of the wafer. Previous visualizations developed within the group ([16], [31], [32], [33], [34], [35]) were designed to solve specific tasks. They were located at a different code path than the continuously changing software stack and were thus not compiled with the underlying data structures. Errors occurred, the visualizations were not maintained and are mostly not usable any longer.

Building on the work of my previous internship, a browser-based visualization was developed in the course of this thesis. The goal is to lay a solid groundwork for a maintainable and easily extendible software that could primarily be used as a static debugging tool. The hardware is to be represented in the necessary detail and a few configuration parameters visualized. As a crucial feature for debugging, visualization of the L1 routes is implemented. By separating the code in modules with discrete responsibilities it will be easy to make changes to the software at a later point.

4.1. Visualization Features

Before the actual visualization[2] can be started, the wafer configuration has to be loaded into the program. A start screen (Figure 4) lets the user browse through local files, or choose a file via drag and drop. The software checks if any file was selected at all, but has no functionality to check the type of the file, so it is the user's responsibility to upload a correct `marocco::results` file that can be processed by Marocco. On hitting the "upload" button, the file is uploaded, and the visualization

[2]A working example of the visualization is available from the Electronic Vision(s) GitHub account: `https://github.com/electronicvisions/wafer-visu.git`.

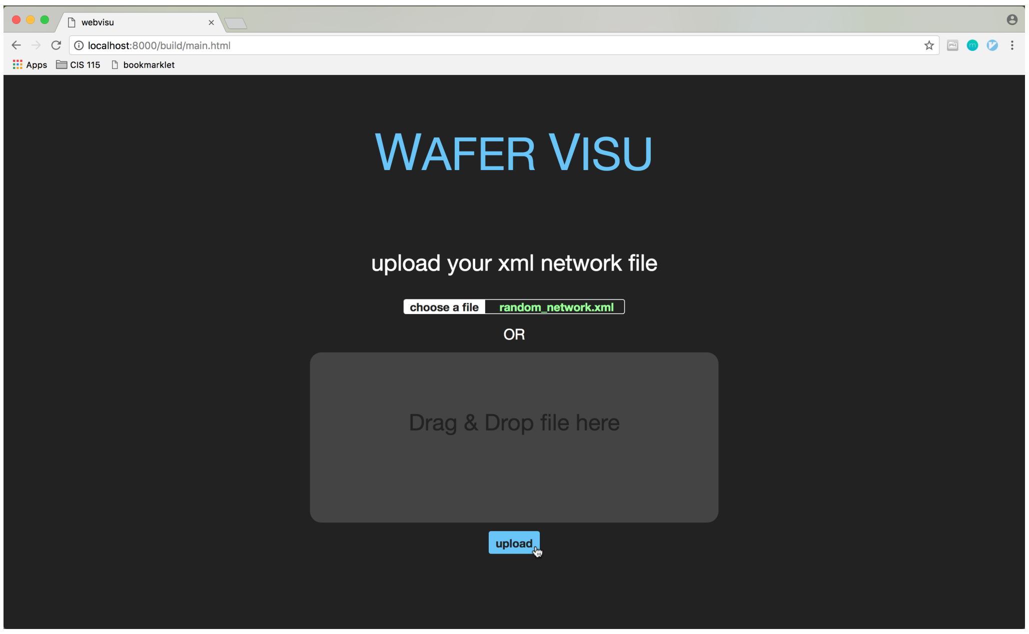


Figure 4: Starting the software by opening `main.html` opens up a start screen. The network configuration in the `marocco::results` file can be selected via the file browser or drag and drop. Hitting the upload button starts the visualization for that network configuration.

initialized and started.

The user will first see the full wafer in what is called the "overview" (Figure 5). The HICANNs are drawn as rectangles with a two-to-one ratio, resembling the shape of the physical wafer. As explained in section 4.2.2, the wafer can and has to be drawn at different levels of detail. The overview does not show all the elements of a HICANN in full detail, but instead provides some cumulative information about the utilization of different parts of the wafer. L1 Bus segments, for instance, are drawn as rectangles in a color representing the number of routes running over all the buses together on that segment. Further, the user can choose to display photographs of the HICANNs to get a physically more accurate visualization. In what is called the "detailview", all single buses are drawn on the HICANN as lines, so the user can check for example the exact path of a L1 route from the source to the target HICANN.

There are two fundamentally different modes to coordinate the visualization. The

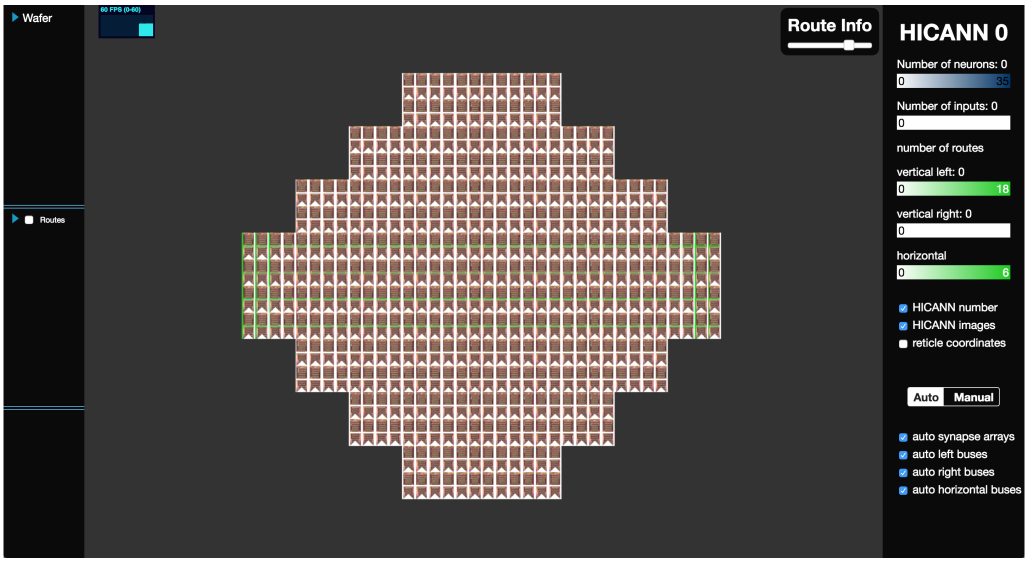


Figure 5: After uploading the network file, the visualization starts with an overview of the full wafer. The UI was designed to give much space to the visualization window and has two panels to access parameters and control the visualization.

automatic mode switches automatically between overview and detailview when a certain zooming threshold is passed. This way the user does not need to worry about manually selecting the supposedly appropriate level of detail and also no performance issues should occur. However, the automatic switching yields short loading breaks during panning and zooming, also the user might want to display different details than the ones proposed by the automatic mode. Hence, a manual mode was introduced that gives the user complete control over what elements to show for each HICANN. Elements can be selected and deselected via checkboxes.

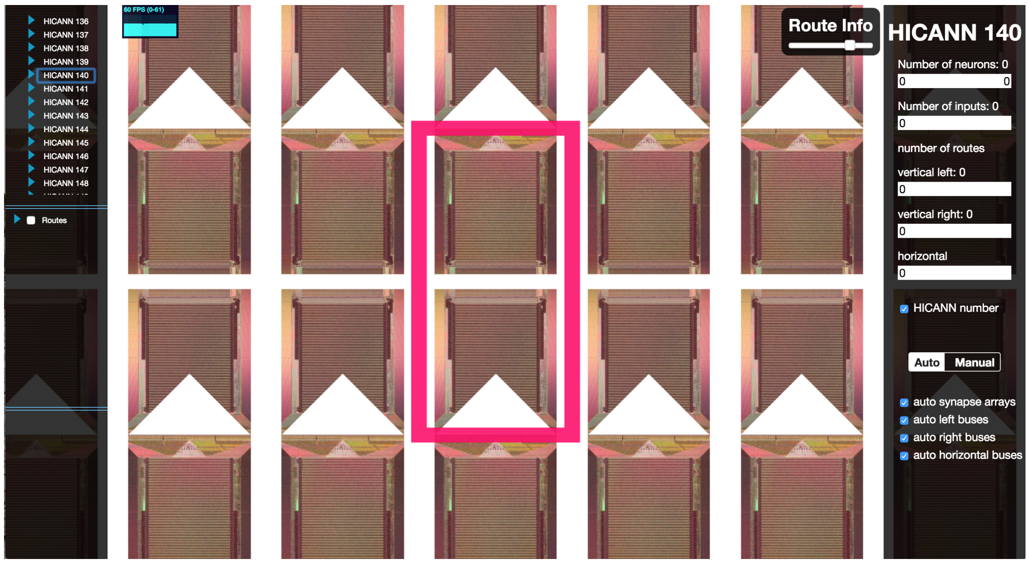
User Interface The UI is designed to leave the main space to the visualization. Pinned to the right side of the window is an information panel that provides information about a selected HICANN and allows some visualization settings. The number of neurons and the number of inputs on a chip, as well as the number of L1 routes running over the three segments "vertical left", "vertical right", and "horizontal", are displayed together with a color gradient for each property. The color gradient's far left and far right colors correspond to the minimum and the maximum of a property throughout the whole wafer. Below the properties section, the automatic mode "Auto" or the manual mode "Manual" can be selected. Depending on the selected mode, different options for customization are listed.

The panel on the left side of the window contains two "tree" dropdown lists. The first one "Wafer" contains a list of all HICANNs and their elements for both overview and detailview. The second list "Routes" holds all the L1 routes. The infobox attached to the right info panel shows information about selected routes. When routes are created, a unique number "ID" is assigned. If multiple routes are selected, their IDs are listed. If only one route is selected, the source and target HICANN is displayed and by clicking on "details", all the traversed L1 bus segments are listed.

User Interaction Moving the mouse while holding down the left mouse button moves the whole wafer around (panning). Zooming is achieved by scrolling the mouse wheel. The reaction of the visualization though differs between the automatic and the manual mode. Additionally, the user can customize the modes with checkboxes in the right info panel. In auto mode, the user can choose what details (e.g. synapse arrays, left buses) to load when the threshold is passed. In manual mode, elements for all the HICANNs on the wafer can be selected to be visible or hidden.

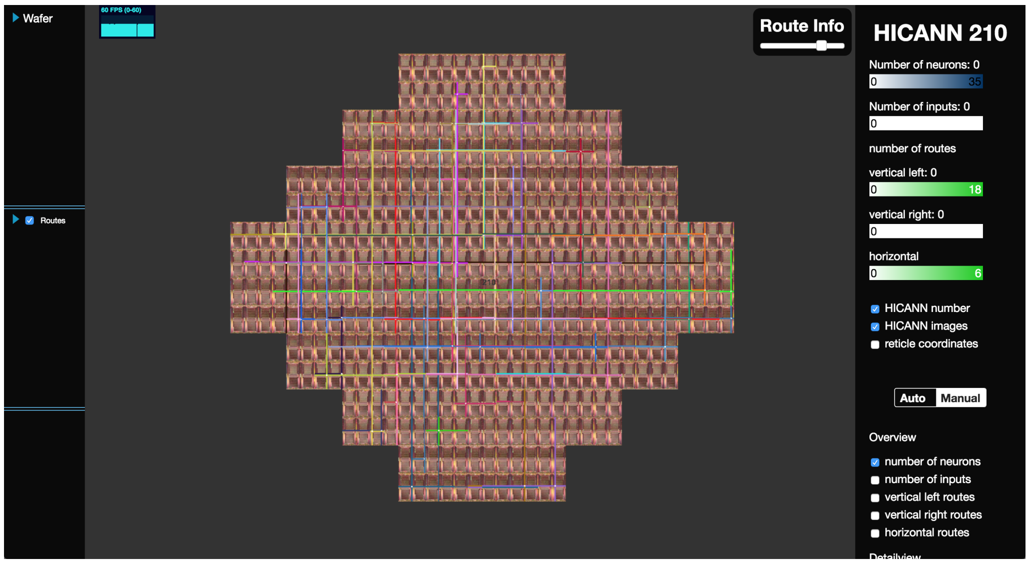
To allow selecting elements for every HICANN separately, the "Wafer" list in the left info panel has checkboxes for each element. This way, the user can choose to display, for example, the detailed L1 buses for all those HICANNs, a specific L1 route is traversing. Clicking on a HICANN in the list (not on the checkbox, but the label) animates the wafer to center the selected HICANN. The properties are displayed in the right info panel (Figure 6). The same can be achieved by clicking on a HICANN in the visualization.

All the L1 routes listed in the "Routes" list have checkboxes as well, making selected routes visible in the visualization. The routes are drawn in random colors but keep the color when they are hidden and displayed again. Clicking on the



Number of neurons: 0
0 0
Number of inputs: 0
0
number of routes
vertical left: 0
0
vertical right: 0
0
horizontal
0
☑ HICANN number
☑ auto synapse arrays
☑ auto left buses
☑ auto right buses
☑ auto horizontal buses

Figure 6: Clicking on a HICANN in the left info panel triggers an animation to center the selected HICANN and displays its properties in the right info panel.



Number of neurons: 0
0 35
Number of inputs: 0
0
number of routes
vertical left: 0
0 18
vertical right: 0
0
horizontal
0 6
☑ HICANN number
☑ HICANN images
☐ reticle coordinates
Overview
☑ number of neurons
☐ number of inputs
☐ vertical left routes
☐ vertical right routes
☐ horizontal routes

Figure 7: All L1 routes are drawn as colored lines running over the L1 bus segments. After clicking and highlighting a selection of routes, all routes can be showed in color again by double-clicking on any route.

label of a route in the list has the same effect as clicking on a route in the visualization: The clicked routes are drawn on top and all other routes are greyed out (Figure 14). If multiple routes are on top of each other and selected together, they are all highlighted. Additionally, the route infobox displays information about the selected routes. Double-clicking on a route or alternatively clicking on the "Routes" label draws all routes in color again and resets the infobox (Figure 7).

During development, an emphasis was put on a user-friendly UI with few settings and a maximized visualization window, but still giving access to the necessary customizations.

4.2. Code Structure and Implementation

One of the major goals and challenges throughout this thesis was to write maintainable code. With growing size of the project, it gets increasingly difficult to maintain and debug code if all is defined in global space. Using the TypeScript compiler option to combine multiple .ts files into a single .js out-file and exploiting namespaces and classes made it possible to separate chunks of code into parts with clearly defined dependencies on each other. Namespacing, i.e. defining modules with their own variable scopes, is per se not yet available in JavaScript ES5[3], but namespaces (formerly internal modules) can be defined in TypeScript and are compiled into function variable scopes.

There are three such namespaces: `tools`, `pixiBackend`, and `internalModule`. `tools` is just a collection of useful general functions, that do not have a specific connection to the visualization software. Examples are `randomHexColor()` that returns a random color in the hexadecimal form or `numberInString(string)` that filters out the numbers in a string.

`pixiBackend` contains all the functionality that directly uses the PixiJS library. The goal is to make it easy to switch to a different visualization library without having to rewrite the whole software. Thus, `pixiBackend` contains, besides all the

[3]ECMAScript 5.1 adopted June 2011, language specification

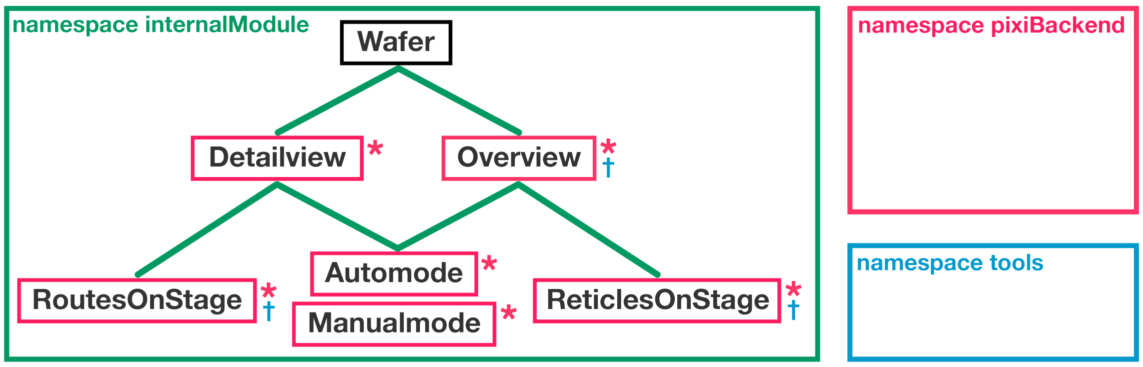


Figure 8: The tree structure shows the dependencies of the namespaces and classes. The `Wafer` class at the top of the tree is completely independent. Subordinate classes need instances of the higher classes for initialization. Dependencies on the namespaces `pixiBackend` and `tools` are indicated by the red star and the blue cross respectively.

containers to store the graphic elements, mostly methods to draw primitive shapes. There are different functions to draw, for example, a rectangle with or without mouse interaction, as a graphics object or as a sprite, multiple or just one rectangle as one graphics object. The functions `zoomIn`, `zoomOut`, and `moveStage` handle the pan & zoom interaction. The following example function `drawCircle` implements creating a new graphics object, adding a primitive filled circle and storing the graphics object in a specified PixiJS container.

```
function drawCircle(container: PIXI.Container, x: number, y:
        number, radius: number, color) {
  const circle = new PIXI.Graphics();
  circle.beginFill(color);
  circle.drawCircle(x, y, radius);
  circle.endFill();
  container.addChild(circle);
}
```

The namespace `internalModule` contains a number of classes that each have their separate purposes but have dependencies on each other. They are written into separate files to keep a clear structure. TypeScript manages to combine classes from multiple files into the same namespace. Figure 8 shows graphically how the namespaces and classes depend on each other. The classes `Wafer`, `Detailview`, `Overview`, `RoutesOnStage`, `Automode`, `Manualmode`, and `ReticlesOnStage` are all defined in the namespace internalModule and do not inherit from each other as the graph may mistakenly suggest. The tree structure shows the dependencies, beginning at the top with the completely independent `Wafer` class. New instances of `Detailview` and `Overview` need an instance of `Wafer` to be created. Similarly, `RoutesOnStage` is dependent on `Detailview` (and therefore also on `Wafer`), `ReticlesOnStage` is dependent on `Overview`, and `Automode` and `Manualmode` are dependent on both `Detailview` and `Overview`. As indicated by the red borders and stars, all classes in `internalModule` except for `Wafer` also have dependencies on the `pixiBackend` namespace. All those classes need to draw graphic elements at some point and need the `pixiBackend` for that. The appendage "onStage" for the routes and reticles classes comes from the top level PixiJS container, typically named "stage". `Overview`, `RoutesOnStage`, and `ReticlesOnStage` also need functions from the `tools` namespace, as indicated by the blue crosses.

In the main.ts file everything comes together and the class instances are connected. The dependencies are created by passing the class instances to the respective constructors. Following up is a more detailed description of the classes in the `internalModule` namespace.[4]

4.2.1. Wafer

The `Wafer` class is located in the file wafer.ts and contains the core hardware elements and properties. Besides the minimum and maximum values of the HICANN coordinates, both enumerated and cartesian, a `hicanns` array contains information about each individual HICANN. For each HICANN a new instance of the `HICANN` class is instantiated with its coordinates and a number of properties that are currently available via the Marocco API. Those properties are: `hasInputs`, `hasNeurons`, `isAvailable`, `numBusesHorizontal`, `numBusesLeft`, `numBusesRight`, `numBusesVertical`, `numInputs`, and `numNeurons`. They are used to draw the color map, as explained in the section Overview. The HICANN coordinates and

[4]Using TypeDoc (`http://typedoc.org/api/`), a complete documentation of the software was generated. The documentation is available as an HTML document in the doc folder of the webvisu project.

properties are not hardcoded but dynamically loaded into the software using the `marocco:results` API inside the method `Wafer.loadOverviewData`. Since the number of available and implemented properties is fairly small, all data can be loaded at once. With a larger number of features and therefore more data to load in future versions, the data should be loaded asynchronously in batches only when it is needed.

```
loadOverviewData(networkFilePath?: string) {
  // load the marocco::results file into Marocco
  const marocco = networkFilePath ? new Module.Marocco(
        networkFilePath) : new Module.Marocco();

  // loop through all HICANNs
  for (let i=this.enumMin; i<=this.enumMax; i++) {
    // Build Maroccos HICANN and property objects
    const enumRanged = new Module.HICANNOnWafer_EnumRanged_type(
        i)
    const hicann = new Module.HICANNOnWafer(enumRanged);
    const properties = marocco.properties(hicann);

    // create new instance of local HICANN class
    // store coordinates and properties
    this.hicanns.push(new HICANN(
      i,
      hicann.x().value(),
      hicann.y().value(),
      properties.has_inputs(),
      properties.has_neurons(),
      properties.is_available(),
      properties.num_buses_horizontal(),
      properties.num_buses_left(),
      properties.num_buses_right(),
      properties.num_buses_vertical(),
      properties.num_inputs(),
      properties.num_neurons(),
    ));
  }

  // further process properties
  this.maxPropertyValues();
}
```

Lastly, the functions `northernHicann`, `easternHicann`, `southernHicann` and `westernHicann` return the enum coordinate of the northern, eastern, southern and western HICANN respectively, if it exists.

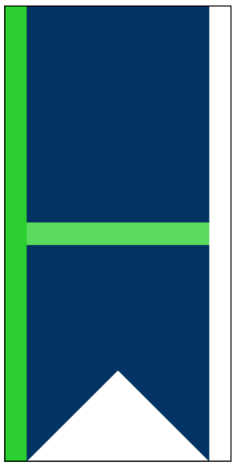


Figure 9: HICANN in the overview representation. A color scheme provides information about the utilization of the part of the chip. The route segments are colored in shades of green, indicating how many routes run over each segment. The blue background color of the chip represents the number of neurons on that HICANN and the triangle at the bottom of the chip is colored depending on the number of inputs.

4.2.2. Overview and Detailview

As a first step, a very rough visualization of the wafer is drawn. Later more detailed objects are added, therefore the visualization is divided into the `Overview` and the `Detailview` class.

Overview Before drawing all the L1 buses and synapses in detail, a color map was developed that had few enough elements to be rendered for the full wafer and still provides meaningful information. All buses on one segment were combined into a single rectangle, with the background color according to the gradient, representing the total number of L1 routes running over that segment (Figure 9). If for example eight routes use the left vertical buses on HICANN 194, and the maximum number of routes using a vertical bus anywhere on the wafer is 18, a rectangle in medium light green color will be drawn on the left side of HICANN 194. The same principle holds for the other route segments as well as the number of neurons and the number of inputs on a chip. The number of neurons is represented by a colored background rectangle, the number of inputs by a triangle located at the bottom of each HICANN.

The full wafer is drawn by calling the `drawWafer` function. Inside that function, the position on the stage is calculated for every HICANN, according to preset `hicannWidth`, `hicannHeight`, and `gap` properties. Subsequently, the color-coded elements are drawn by calling the functions `drawHicannBackground`, `drawInputs`, and `drawBusH`. `drawHicannBackground`, for instance, uses `tools.colorInGradient` to determine the color in the number-of-neurons color gradient and calls `pixiBackend.drawRectangle` to draw the HICANN background in that color.

```
drawHicannBackground(hicannNumber: number, x: number, y: number)
        {
  // calculate color on number of neurons color gradient
  let colorNumNeurons = tools.colorInGradient(...);

  // draw rectangle as hicann representation
  pixiBackend.drawRectangle(pixiBackend.container.backgrounds, x
        , y, this.hicannWidth, this.hicannHeight,
        colorNumNeurons);
}
```

Detailview The detailed visualization of single L1 buses and synapses followed the implementation of `Overview` to pave the path for drawing L1 routes on the buses. To improve performance when drawing the increasingly large number of elements, many elements were grouped together in `PIXI.Graphics` objects and rendered as `PIXI.Sprites`. As an example, all the 128 vertical left buses of one HICANN are drawn together. Rendering sprites instead of graphics objects is computationally more expensive but makes it possible to use antialiasing. Drawing a large number of thin lines as a representation of the L1 buses leads to undesired pixelation effects because the rendering engine has to decide whether the line fits on a row of pixels or not. In video games, antialiasing is very commonly used to smooth edges. Unfortunately, graphics objects cannot be antialiased with the WebGL renderer because it uses the stencil buffer. However, rendering the graphics objects as textures using the canvas renderer allows smoothing edges. These textures are then included as sprites into the visualization. The textures are created with linear scale mode and 10 times the screen resolution. This reduces pixelation effects dramatically (Figure 10). When zoomed in very closely, one starts to see the smoothing around the edges of the buses. To avoid that and get clearly distinguishable shapes at all times, the sprites are removed at a predetermined zoom level and instead the graphics objects drawn (Figure 11).

The method `drawHicann` handles drawing all the graphics objects when entering

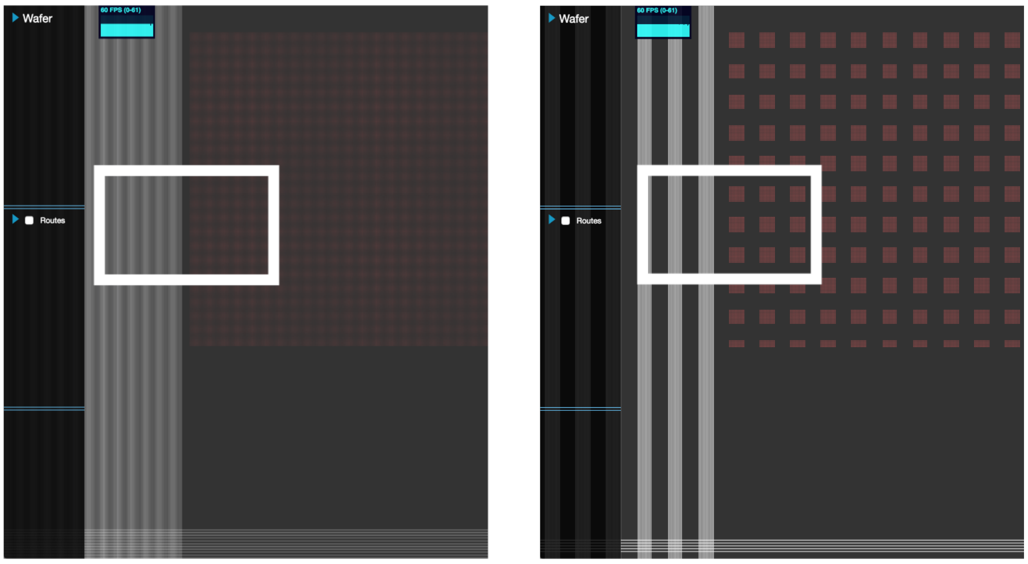


Figure 10: The large number of regularly spaced elements leads to pixelation effects that look like completely missing elements (right image). Using antialiasing, these effects can be drastically reduced (left image).

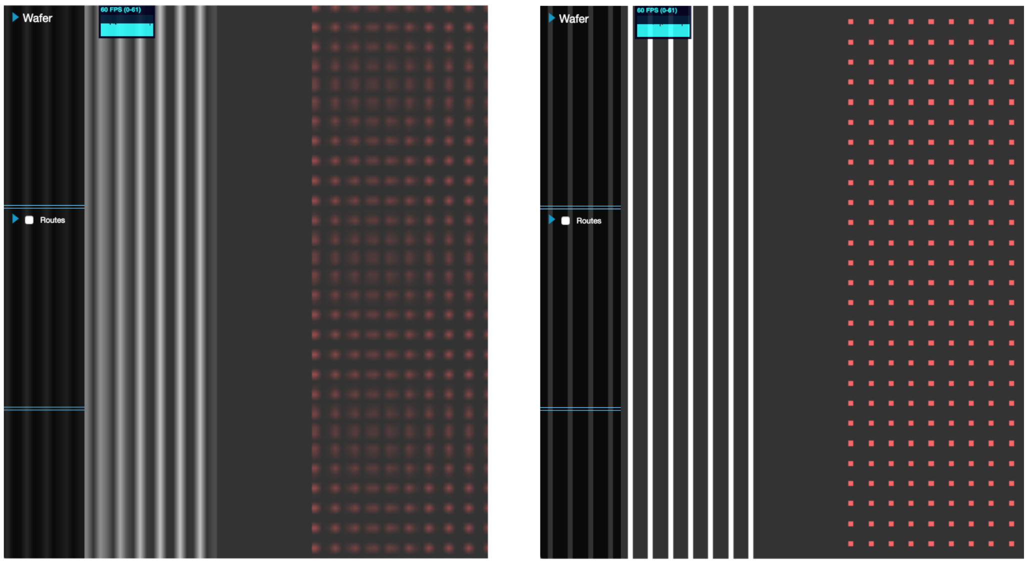


Figure 11: At very close distances, the sprites limited resolution leads to blurry lines (left image). Thus, the sprites are automatically hidden and the graphics objects displayed instead (right image).

the detailview. Similarly, `resetDetailview` can be called when leaving the detailview to remove the graphics objects from the respective `PIXI.Containers`. The methods `hicannCenter`, `hicannClosestToCenter`, `updateSurroundingHicanns`, `distanceFromCanvas`, `determineThreshold`, and `northernHicannCloser`, `easternHicannCloser`, `southernHicannCloser`, `westernHicannCloser` aide determining which HICANN is currently in the center of the display and when to switch to the neighboring HICANN, as explained in the chapter Automode.

4.2.3. Automode and Manual Mode

All the Elements that are drawn with the `Overview` and `Detailview` classes need to be managed. Two different user modes were developed to render objects visible or hide them depending on the user input. The automode determines automatically what details on which HICANN to draw, while the manual mode gives the user complete control. The possibility to switch the mode at every point makes this complicated, on the other hand maintaining a lot of variables that save the state of the visualization can be resource-inefficient and lead to confusing code. The task is made even more difficult by the need to delete graphics data when it is not needed anymore, in order to improve performance. Reactive programming based on so-called asynchronous event streams is a common concept in JavaScript programming. Libraries like react.js [36] help to organize large numbers of asynchronously interacting components, but were not yet considered necessary for this project.

Automode The automode makes zoom level dependent decisions on what details to visualize. The wafer visualization is split into three parts, overview, detailview, and detailview level 2. Zooming past `Detailview.threshold` and `Detailview.threshold2` determines which detail level to load. The Detailview and DetailviewLevelTwo are only loaded for a limited number of HICANNs. Specifically the one, the mouse is over during zooming, or the one closest to the center of the screen as well as the eight surrounding HICANNs. Drawing all details of all

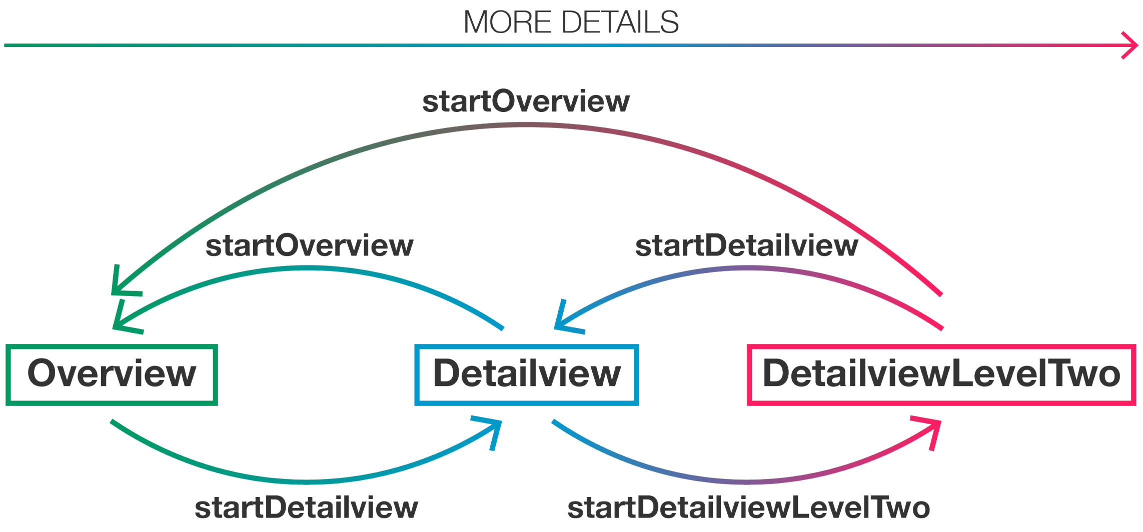


Figure 12: The automode switches automatically between the three levels `Overview`, `Detailview` and `DetailviewLevelTwo`. The basic structure are start functions for each level. `Detailview` can be reached from either `Overview` or `DetailviewLevelTwo` and `startDetailview` takes that into account. `Overview` can be accessed directly from `DetailviewLevelTwo` when either the HICANN is moved out of scope, or the manual mode is started.

HICANNs would yield painfully slow performance, if not cause the software to crash on most browsers and devices[5].

Changing between detail levels is done by the methods `startOverview`, `startDetailview`, and `startDetailviewLevelTwo` as shown in Figure 12. Each of these methods needs to take care of three things: drawing graphic elements if they are not drawn yet, setting the `visible` property of already drawn graphic elements and updating the `Detailview` parameters `enabled` and `levelTwoEnabled` that describe the current state of the level of detail. The detailview can be reached either by zooming in from the overview or zooming out of detailview level 2. The sprite representation of the L1 bus segments is already drawn as soon as the visualization is loaded, but hidden to make it available for the Manual Mode. The graphics objects for the bus segments as well as sprites and graphics objects for the

[5]Tested on **A**: Chromium Version 57.0.2987, running on Debian 8.10 with 16 GB RAM, 256 MB VRAM and **B**: Google Chrome Version 65.0.3325, running on macOS HighSierra with 16GB RAM and 2GB VRAM.

synapse arrays are not drawn yet. Hence if the detailview is started coming from the overview, those parts need to be drawn by calling `Detailview.drawHicann`. Prior to that, the indices of the surrounding HICANNs are determined. All elements are set visible or hidden with the methods `setOverview`, `setDetailview`, and `setDetailviewLevelTwo`.

```
startDetailview(hicannIndex: number) {
  // check if coming from detailview level two
  if (!this.detailview.enabled) {
    this.getDetailedHicanns(hicannIndex);
    // draw detail objects i.e.
    //   synapse array level one and level two
    //   buses level two
    for (const hicannIndex of this.detailedHicanns) {
      this.detailview.drawHicann(hicannIndex);
    };
  }
  // display level one detailview
  // hide overview containers
  for (const hicannIndex of this.detailedHicanns) {
    this.setDetailview(hicannIndex, true);
    this.setOverview(hicannIndex, false);
  }
  // hide level two details
  this.setDetailviewLevelTwo(false);
  // set parameters in detailview
  this.detailview.enabled = true;
  this.detailview.levelTwoEnabled = false;
  this.detailview.currentHicann = hicannIndex;
  this.detailview.updateSurroundingHicanns();
}
```

`startDetailviewLevelTwo` and `startOverview` work in principle exactly the same. Starting level two, nothing needs to be drawn but only the `visible` property set correctly for all elements. `startOverview` serves as a reset function for all the detailview elements, both graphics objects and sprites. The synapse arrays have to be removed again in order to take the load off the renderer.

This set of functions allows now to easily handle not only zooming in and out but also switching to automode from every zoom level or panning the currently centered HICANN out of the scope of the screen. `startNorthernHicann`, `startEasternHicann`, etc. first call `startOverview` to reset the `Detailview` and then `startDetailview` and `startDetailviewLevelTwo` on the neighboring HI-

CANN, depending on the `Detailview.enabled` and `Detailview.levelTwoEnabled` properties.

Manual Mode The manual mode is in principle very simple but is complicated by the large number of DOM elements, the mode is interacting with. The idea is, that the user has full control over which elements to show in both the overview and the detailview representation. In the left panel of the window, a complete tree list of all HICANNs allows setting checkboxes for the full overview, full detailview as well as the individual elements such as detailed left buses. The right info panel holds further checkboxes to select all elements of one type at once. When switching to automode and back to manual mode, the state of the visualization should be restored, all elements set to visible or invisible, just like before leaving the manual mode. For that reason, a `selectedElements` object was introduced to always hold the information about which elements are selected by the user. Here again, a reactive programming scheme is used to synchronize the following properties: The DOM checkboxes changed by the user, the `selectedElements` object in manual mode, and the `visible` property of the PixiJS elements. The flowchart Figure 13 shows this connection. Even though the manual mode is designed to let the user control which elements are visualized, the switch between sprites and graphics objects is done automatically, because there is no benefit from showing blurry sprites instead of sharp graphics objects at a large zoom scale.

Since the user chooses which elements from the detailview to show, there is in principle no `startDetailview` function as in `Automode` needed. However, the text showing the HICANN enum coordinate when hovering the mouse over a HICANN is not useful when only one HICANN takes up most of the screen. Therefore, the HICANN number is hidden when the zoom level is greater than the threshold for the detailview. Detailview level two (i.e. graphics objects instead of sprites) is entered automatically and not managed by the user, as previously explained. The methods `startDetailviewLevelTwo` and `leaveDetailviewLevelTwo` take care of

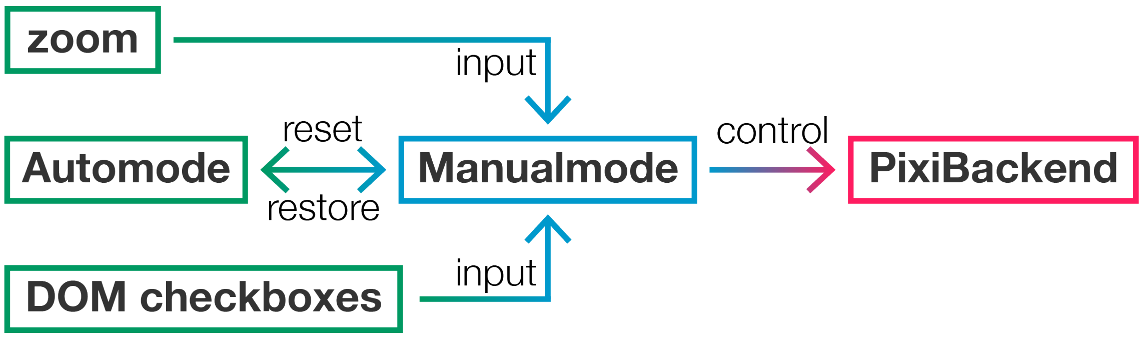


Figure 13: User interaction in the form of zooming and clicking checkboxes is used as input for the manual mode to control the visualization via the pixiBackend. Switching from automode to the manual mode requires resetting the details from the automode and restoring the previous selections of the manual mode

that for the manual mode. Essentially the same three things need to be done as in the corresponding methods of the `Automode`. Variables, describing the state of the visualization need to be set, missing graphic elements are drawn (or removed when leaving the detail level), and `visible` properties set for the PixiJS containers. To maximize performance, only the graphics objects for those buses, that are selected by the user via the checkboxes, are drawn. Consequently, the PixiJS container for these graphics objects has in general not 384 children, but only as many as elements were selected, and not even in numerical order since elements can be selected and deselected in any order at any time. Hence, an additional object `containerIndices` keeps track of the graphics objects' index in the respective PixiJS container. If single detailview elements are selected via the checkboxes, the methods `busesLeft`, `busesRight`, and `busesHorizontal` handle setting properties and possibly drawing new graphics objects just for that one selected element.

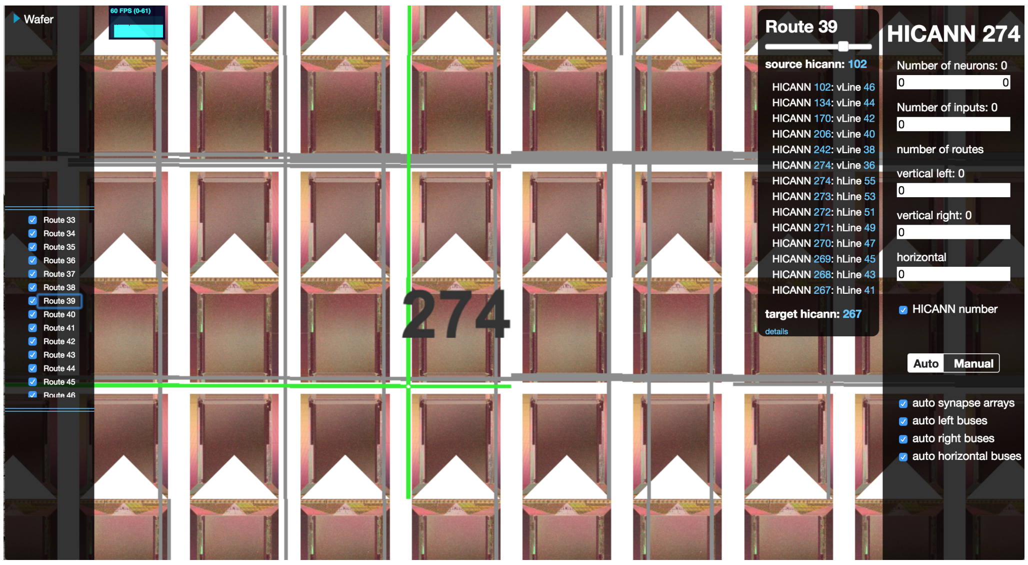
Finally, the checkboxes themselves should be consistent, meaning that if for example all detailview elements for one HICANN are checked, the detailview checkbox itself for that HICANN should be checked as well. The `checkAllCheckboxes` and `setAllCheckboxes` methods handle that.

4.2.4. Routes

Drawing detailed L1 routes between HICANNs was one of the main goals right from the start and is a useful feature for debugging purposes. Instead of seeing only how many routes run over which L1 bus segment, the exact bus should now be highlighted. The data is at the time of writing not yet available via the `marocco::results` API, so L1 routes were created manually using the stand-alone library for L1 routing [17] and stored in JavaScript Object Notation (JSON) for the meantime. The function `loadRouteData` in routes.ts handles the JSON and stores the routes in an internal format. An instance of `Route` holds an array of `RouteElement`s of type ”vLine” or ”hLine” for vertical and horizontal L1 bus segments respectively. The class `RoutesOnStage` holds all the `Route` instances and has the methods needed for drawing the Routes and interacting with them.

L1 Routes are drawn as a number of randomly colored basic rectangles for each L1 route segment. At the intersection of horizontal and vertical route segments of the same route, switches are represented by white circles. Similar to the HICANN list for the manual mode, a routes list in the left info panel on the screen lets the user select which routes to show. Since the routes should be clearly visible at every zoom scale, the route width is adjusted automatically in five steps during zooming. When the zoom event occurs, the method `currentZoomLevel` determines which of the five levels should be started and compares that to `zoomLevels.current`. If they do not coincide, all routes are removed from the PixiJS containers, and new routes drawn with the adjusted width. All routes have a `visible` property, redrawing routes updates only the position and width and height values of the old route objects.

When L1 routes are clicked, all other routes are greyed out and properties about the selected route are shown in the route infobox (Figure 14). Source and Target HICANN can be extended to show a list of all the L1 routes segments. Double-clicking on a route resets the route infobox and shows all routes in color again. The same interaction is possible by clicking the ”Routes” label in the route list.



Route 39

source hicann: 102

HICANN 102: vLine 46
HICANN 134: vLine 44
HICANN 170: vLine 42
HICANN 206: vLine 40
HICANN 242: vLine 38
HICANN 274: vLine 36
HICANN 274: hLine 55
HICANN 273: hLine 53
HICANN 272: hLine 51
HICANN 271: hLine 49
HICANN 270: hLine 47
HICANN 269: hLine 45
HICANN 268: hLine 43
HICANN 267: hLine 41

target hicann: 267

details

HICANN 274

Number of neurons: 0
0 0

Number of inputs: 0
0

number of routes

vertical left: 0
0

vertical right: 0
0

horizontal
0

✓ HICANN number

Auto Manual

✓ auto synapse arrays
✓ auto left buses
✓ auto right buses
✓ auto horizontal buses

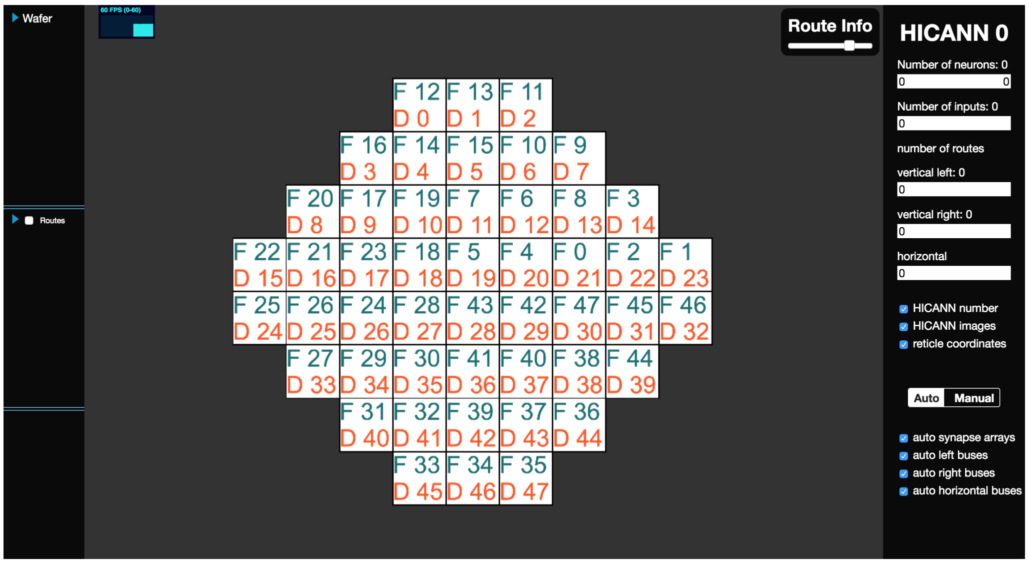
Figure 14: Selecting one or more L1 routes by clicking on them in the visualization or in the routes list in the left info panel highlights those routes by greying out all other routes. If only one route was clicked, the route infobox in the top right corner of the visualization displays detailed information about the route.

4.2.5. Reticle Coordinates

As a last feature, a lookup plot for the reticle coordinates (i.e. DNC coordinates) and FPGA coordinates was created (Figure 15). When zooming all the way out (wafer is fully displayed), an overlay with reticle and FPGA coordinates is automatically shown. Additionally, the overlay is accessible at every zoom level via a checkbox in the right info panel. The drawing and hiding of the lookup plot works with the same principles as already discussed for other parts of the visualization, the interesting part was to calculate the position of the reticles on the wafer. Since the positions for all HICANNs were known, the top left HICANN of a reticle was determined from the reticle coordinate.

4.2.6. Global Namespace

In the global variable scope of the main.ts file, the visualization software is set up and all the modules built together. After the DOM finished loading, `setupScreen` is started. A setup screen allows the user to upload the `marocco::results` file either


Wafer
60 FPS (0-60)
Routes
Route Info
F 12 F 13 F 11
D 0 D 1 D 2
F 16 F 14 F 15 F 10 F 9
D 3 D 4 D 5 D 6 D 7
F 20 F 17 F 19 F 7 F 6 F 8 F 3
D 8 D 9 D 10 D 11 D 12 D 13 D 14
F 22 F 21 F 23 F 18 F 5 F 4 F 0 F 2 F 1
D 15 D 16 D 17 D 18 D 19 D 20 D 21 D 22 D 23
F 25 F 26 F 24 F 28 F 43 F 42 F 47 F 45 F 46
D 24 D 25 D 26 D 27 D 28 D 29 D 30 D 31 D 32
F 27 F 29 F 30 F 41 F 40 F 38 F 44
D 33 D 34 D 35 D 36 D 37 D 38 D 39
F 31 F 32 F 39 F 37 F 36
D 40 D 41 D 42 D 43 D 44
F 33 F 34 F 35
D 45 D 46 D 47
Auto
Manual

HICANN 0

Number of neurons: 0

0 0

Number of inputs: 0

0

number of routes

vertical left: 0

0

vertical right: 0

0

horizontal

0

✓ HICANN number
✓ HICANN images
✓ reticle coordinates

✓ auto synapse arrays
✓ auto left buses
✓ auto right buses
✓ auto horizontal buses

Figure 15: The lookup plot for reticle and FPGA coordinates can be accessed either by zooming all the way out or by clicking the "reticle coordinates" checkbox in the settings section of the right info panel.

using the file browser or via drag and drop. Clicking the upload button, the file is read using a JavaScript `FileReader` object and then written into emscripten's virtual file system to make it accessible for marocco. The loading screen is started and the function `main` called, which starts the core visualization program.

Initially, everything was declared in global space, later parts were separated and packed into namespaces and classes. Still a number of global variables remain, serving mainly as configuration properties for the visualization, such as HICANN width and also the colors for property gradients in the right info panel.

At the top of the document, all module files are referenced to make them available in main.ts. Inside `main`, new instances of those classes are created and built together. It is important to do that in the right order respecting the dependencies. First, pixiBackend is set up, creating the PixiJS container "folder" structure. Following the dependencies tree in Figure 8, a new instance of `Wafer` is created and the data from the `marocco::results` file loaded with `wafer.loadOverviewData`. Next, `Overview` is set up using the global visualization property variables. At this point, the overview of the wafer can already be drawn with `Overview.drawWafer`.

Subsequently, `Detailview`, `RoutesOnStage`, `ReticlesOnStage` and the two modes `Automode` and `Manualmode` are initialized. The visualization is set to start in automatic mode.

Even though most of the HTML structure is coded statically or controlled with CSS, some JavaScript is necessary to set up the UI. The HICANN list and the routes list in the left info panel are created dynamically from the wafer data in the `marocco::results` file and the routing data, currently accessed via the routes.json file.

Lastly, all the event handlers for mouse and keyboard interaction, as well as checkboxes are defined in the `main` function. `handleWheel` does not only zoom in and out according to mouse wheel movement but also controls manual or automatic Mode and triggers the route width adjustments. Basically, the PixiJS scaling value is compared to the predetermined thresholds to then call the respective functions in manual or automatic Mode. An emphasis was put on performing as few operations as possible to try and allow smooth zooming, but also keep a clear structure that can easily be modified for new features.

4.3. Benchmarking

Even though the PixiJS library allows high-performance visualizations, it is easy to run into performance issues by using the wrong mechanisms. The goal of the first part of this section is to explore the general behavior of the PixiJS library, to find out about limitations and draw conclusions about best practices. In the second part, the visualization software's capability of drawing L1 routes is tested.

4.3.1. PixiJS Benchmarking

The presented benchmarking was executed in a testing environment, using the namespace `pixiBackend` that was developed for the wafer visualization. A simple canvas of the size 952 pts $\times$ 781 pts[6] was set up and WebGL used for rendering. High-

[6] 952 pts $\times$ 781 pts was the effective size of the browser window and has no further reason.

resolution time measurements are possible in browsers with the `performance.now()` method. To prevent timing attacks, the time stamps have been rounded to the nearest 5 microseconds in most browsers [37]. As a reaction to Spectre and Meltdown the time resolution was then even further reduced [38]. For the purposes of this benchmarking however, the time resolution of `performance.now()` is still precise enough and was used to time different processes such as creating ("drawing") graphics objects in PixiJS. To calculate the frame rate in frames per second (fps), the stat.js tool counts each time, a new frame is drawn with `requestAnimationFrame()` over the span of 1000 milliseconds. The tool was extended to write out 10 frame rate measurements by pressing an action key and calculating mean value and standard deviation. The distribution of the frame rate measurements varied a lot but resembled in most cases a gaussian distribution. More data points would be necessary to perform a fit. The GPU memory and RAM usage as well as CPU usage was retrieved from the Google Chrome Task Manager [39].

All tests were performed with Google Chrome Version 65.0.3325 on macOS High-Sierra with 16GB RAM and 2GB VRAM.

Comparison of Methods Since WebGL is using the GPU for hardware accelerated computation, the GPU memory presents a limitation. However, depending on the "method" used for drawing in PixiJS, other factors will lead to limitations beforehand. Table 3 shows a comparison between storing every shape in its own graphics object (GO), storing all shapes as a single graphics object, and creating a sprite with twice the resolution of the canvas. In each case, an array of 50×50 squares with a length of 100 px and a 10 px gap between the squares was drawn. The frame rate and CPU usage were measured during panning interaction.

Using a new graphics object for every new square yields with 9 fps the worst performance. This is certainly not only due to the GPU memory usage of 324 MB, since drawing a sprite takes up 922 MB of GPU memory but still results in the maximum frame rate of 60 fps. The high RAM consumption of around 2 GB probably con-

parameter	multiple GO	single GO	single sprite (2x resolution)
GPU memory [MB]	324.0±0.5	3.00±0.05	922.0±0.5
RAM [MB]	2000±50	87.40±0.05	54.90±0.05
CPU [%]	23.25±0.79	21.19±0.61	21.02±0.42
upload time [ms]	7730.300±0.005	11.800±0.005	5.600±0.005
frame rate [fps]	9.01±0.39	59.99±0.01	60.0±0.31

Table 3: Comparison of different methods to draw graphics in PixiJS. Drawing 50x50 squares each stored in a separate graphics object (GO) yields by far the worst performance. sprites require a significantly higher amount of GPU memory than drawing all rectangles as one graphics object but a lower amount of RAM.

tributes to the low frame rate for the multiple GOs. Interestingly the CPU usage during panning is almost the same for all three drawing methods, but when leaving the browser tab in idle state with multiple graphics objects drawn, a CPU usage of around 99 % was noted. The reason for this is not clear as there were no instructions coming from the test software at all. Storing all squares in a single graphics object takes up only 3 MB of GPU memory and is outweighed by RAM usage as will be explored in detail later. The upload time should theoretically be the timespan needed for uploading the data into the GPU, which is necessary before rendering the first time. However, there are other processes that cannot be measured well, which are often times much longer. Thus, the upload time turns out to be not a good measure of the latency.

In the following, the methods are explored in detail.

Sprites As previously shown, the GPU memory is the primary limiting factor of performance, when drawing large sprites. The GPU memory needed for a sprite depends on its width and height as well as its resolution. PixiJS provides an option to specify a resolution relative to the canvas resolution. In the following, resolution refers to the ratio $\frac{\text{sprite resolution}}{\text{canvas resolution}}$. Canvas resolution is not necessarily the devices screen resolution. When HiDPI (High Dots Per Inch) images became common, device independent sizes had to be introduced to display objects in approximately the same size on different devices, independent of its resolution. The retina screen of a MacBook, for example, has approximately twice the resolution of a traditional screen. Web browsers deal in different ways with this issue. A canvas in Safari, for example, takes care of the higher resolution, while Chrome scales the canvas in a way to always have the standard resolution at the same canvas size. When comparing memory allocation from drawing on a canvas in a browser other than Chrome, the device's resolution has to be taken into consideration.

In Figure 16 the allocated GPU memory for a sprite is plotted against its size and resolution. For the left graph, a varying number of squares with a length of 100px and gap between the squares of 10px was drawn into a single sprite with resolution 2. The relation can be well approximated by a linear function. For the second graph, an array of 30x30 squares with the same size as before was drawn at different resolutions. The quadratic relation comes from the two dimensions of the sprite. The GPU memory allocation is limited by the 2.0GB VRAM of the Graphics Card. Using the gradient fit parameter from the first graph, and taking the quadratic dependency on the resolution into account, a formula for calculating the required memory of a sprite can be put up:

$$\text{size[Bytes]} = 7.48\frac{\text{Bytes}}{\text{px}} \cdot \text{numPixels} \cdot \text{resolution}^2$$

where numPixels is the number of pixels the sprite would have at resolution one.

Even though the maximum size of a sprite is limited by the VRAM, multiple sprites each of maximum size can be drawn after each other. In that case, the sprite data is distributed to both RAM and GPU memory while the GPU memory rises above the physical amount of VRAM as shown in the bar chart Figure 17. This is possible because the GPU memory is a virtual amount, including swap memory. RAM and GPU memory together equal the amount of memory needed for one sprite multiplied by the number of sprites, as expected. It has to be noted though, that the sprites need to be drawn on top of each other. Drawing them with a relative shift results in a too large size of the effective sprite.

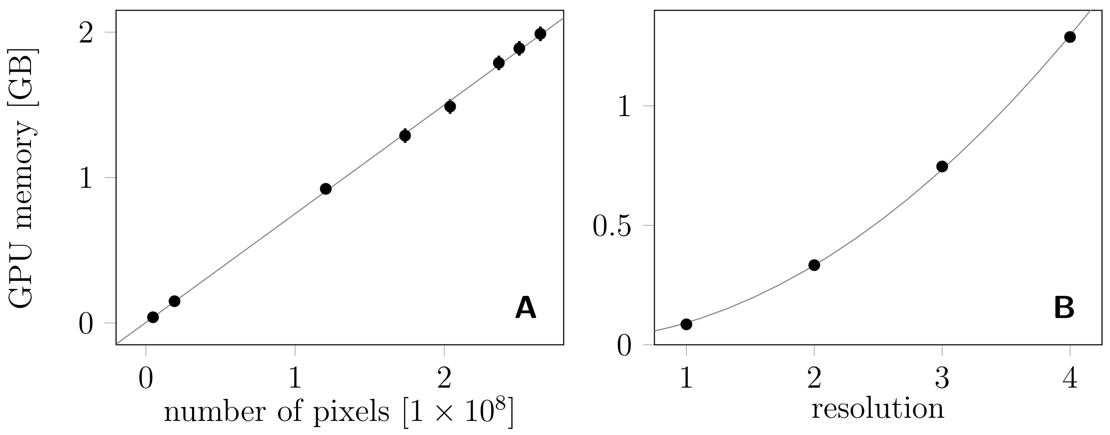


Figure 16: Memory allocation for sprites, depending on their size and resolution. **A**: a sprite of varying size with constant resolution was drawn and the allocated GPU memory measured. The linear fit has a gradient of $7.48\frac{\text{Bytes}}{\text{px}}$. **B**: The sprite size was kept constant and the resolution relative to the canvas resolution increased. In effect, the sprite size increases quadratically as expected due to the two-dimensionality.

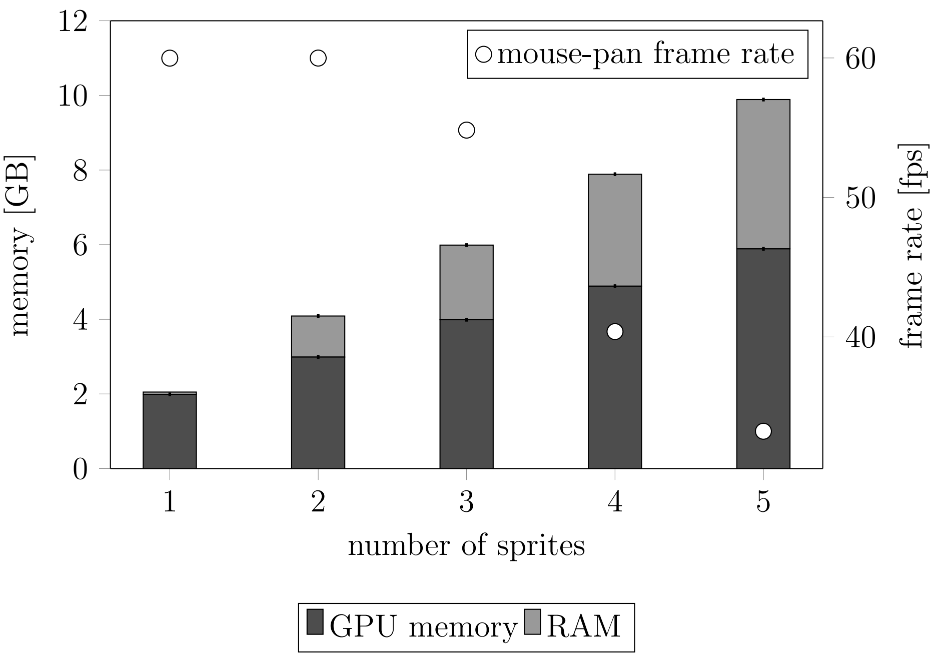


Figure 17: Memory allocation for drawing multiple sprites, each of maximum size. The sprites need to be drawn at the same position on the canvas to keep the effective size of the resulting sprite the same. This way, the GPU memory can exceed the physically available VRAM. In addition, RAM is allocated. RAM and GPU memory add up to a multiple of the 2 GB needed for one sprite. the Memory has to be swapped around during rendering processes, resulting in a lower framerate (white dots).

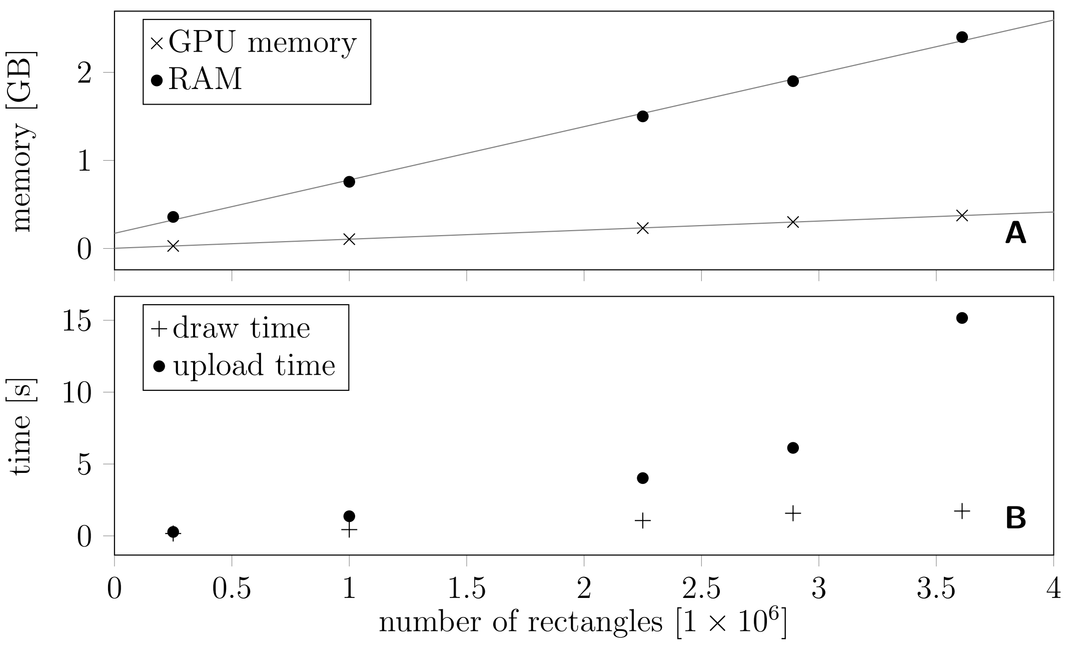


Figure 18: A varying number of rectangles was drawn into a single graphics object until crashing at around 3.6 million elements. **A**: RAM and GPU memory allocation depend linearly on the number of rectangles and RAM is the limiting factor in this case. **B**: Especially the time to upload graphics data to the GPU increases rapidly up to 15 seconds for the maximum number of rectangles.

Interestingly, the frame rate remains constantly at 60 fps until the GPU memory is full. This means that if no data has to be loaded into the GPU it is processed really fast. When the physical size of the VRAM is exceeded, the frame rate drops significantly, as shown by the white dots in Figure 17.

Graphics Objects To explore the limits of drawing graphics in the most performant way possible, a varying number of squares was created as a single graphics object (Figure 18). The relationship between both RAM and GPU memory and the number of rectangles is clearly linear, up to the maximum number of around 3.6 million rectangles at which the whole browser tab crashed. For graphics objects, the limiting factor is RAM and not GPU memory, as the amount of RAM that can be used in a single tab is limited by the browser. In Chrome, the available memory for a tab can be increased by setting the `--max_old_space_size` flag.

The time to draw the rectangles and prepare for rendering by uploading the graphics to the GPU also increases significantly. The frame rate decreases at approximately 2 million rectangles but is with 37 fps at 3.6 million rectangles still high enough to yield smooth usage.

General Behavior Two things have to be kept in mind when deciding whether to use graphics objects or sprites. Firstly, the performance of graphics objects does not depend on their size. Drawing a square of length 1 px or 1000 px does not make a difference. The performance of sprites, on the contrary, depends significantly on their size as described before, but the complexity of the texture does not make a difference. However, a very detailed sprite with tiny elements has to be rendered at high resolution to show these details. In general, it is best to use plain graphics objects and avoid sprites. The primary benefit of sprites is, that antialiasing can be used to minimize pixel effects for large amounts of small elements.

To test, how graphics outside of the canvas boundaries influence performance, an array of 50×50 rectangles, each stored as an individual graphics object was created. Panning the graphics outside of the canvas boundaries increased the frame rate from 10 to 43 fps. In a second experiment, five large sprites with a total GPU memory of 5.9 GB were created. For the sprites, the frame rate increased from 37 to the maximum of 60 fps.

Setting the visibility property of graphic elements happens in split seconds and elements that are set invisible increase performance significantly. Therefore a good way to deal with regularly changing elements such as the width adjusting routes is to preload them all into the GPU and then simply set the `visible` properties. PixiJS allows uploading data to the GPU with the `PIXI.prepare` namespace and remove elements again by calling the `destroy` method. By default, the PixiJS garbage collector removes unused objects from the GPU memory after two minutes, but all the processes can also be managed manually. However, doing this for the whole visualization requires a lot of extra effort and has to be thoroughly planned, since errors could quickly lead to worse performance. Additionally, the software would become harder to maintain.

4.3.2. Visualization Benchmarking

To get an estimate of the limits in visualizing L1 routes, a varying number of routes between 100 and 10 000 was drawn. The total time to draw all lines as well as the frame rate during panning and hovering the mouse over the wafer were measured. The number of routes was increased in steps of 100 between 100 and 1000 routes, then in steps of 1000 up to 5000 routes and lastly, 10 000 routes were drawn. The time, the software takes to draw all elements was measured simply using a stopwatch since the underlying processes could not be timed using JavaScript. 10 measurements were taken for each configuration and an estimated reaction time of $t_{\text{react}} = (0.15 \pm 0.05)$s for the manual handling of the stopwatch was subtracted. As can be clearly seen in Figure 19, the time to draw all routes increases in a linear fashion with the number of routes. Since in the current implementation of the software, all routes have to be redrawn every time the route width is adjusted, long drawing times have a direct negative impact on the usability of the software. While for 100 routes drawing happens more or less instantaneously, zooming over a route width threshold is already noticeably laggy for 500 routes. For 700 routes the drawing time is larger than one second and zooming over thresholds becomes annoying. For 3000 routes, drawing times are almost prohibitively high and at 10 000 routes, the visualization cannot be used in a proper way anymore.

As a second benchmark parameter, the frame rate for panning and hovering the mouse is assessed. The frame rate is limited to 60 fps by the display, theoretically possible changes in the frame rate above 60 fps cannot be detected, thus the "plateau" in Figure 19. Drawing more routes than 2000 causes the frame rate to drop quickly during both hovering the mouse and panning the view. A low of 15 fps within the measured area is reached for 10 000 routes.

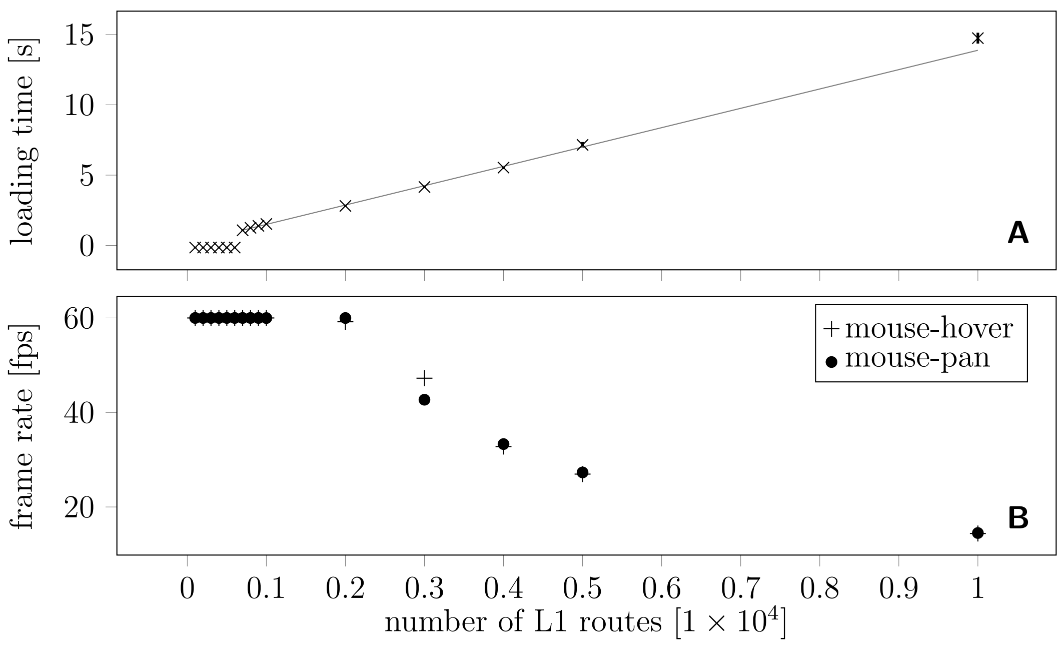


Figure 19: The number of visualized L1 routes was increased from 100 routes up to 10 000 routes. **A**: The loading time (i.e. time between clicking the routes checkbox and the first rendering of the routes) increases linearly. Loading times below one second could not be well measured by hand and were thus set to zero. **A**: The frame rate for hovering the mouse and panning drops quickly at around 2000 routes.

4.3.3. Conclusion

For the general use of PixiJS, graphics objects should be the first choice if large numbers of primitive shapes have to be drawn. The benchmarking showed, that performance is significantly higher for graphics objects than for sprites because latter require vastly more GPU memory. Further, shapes should be stored in as few graphics objects as possible to improve performance.

By implementing the results of the general PixiJS benchmarking, the performance of L1 route visualization could be even further improved. Drawing 1000 routes both in color and greyed out, for all five currently implemented width levels yields a total of approximately 300 000 graphics shapes. Drawing all the elements as one graphics object would certainly cause no memory issues (Figure 18). The routes have to be separated into different graphics objects for colored and greyed out routes as well as the different widths, but 10 graphics objects can be easily handled by PixiJS as well. On clicking the ”Routes” checkbox, all routes in all widths in color as well as greyed out could be created and uploaded to the GPU. Then, depending on the zoom level, the correct graphics `visible` property would be set, which happens in a split second. Having a faster response for a higher number of routes comes at the cost of having to draw no routes or all routes at once since they are bound together in one graphics object. However, upon clicking on routes in the visualization, all routes could be greyed out and the selected ones drawn additionally at that moment since this will require only a small number of new graphics objects.

4.4. Extending the Visualization

Adding features to the visualization is more or less complicated, depending on the type of feature. However, an effort was put into keeping a clear code structure through modularity and thus making the process of extending the software as easy as possible. This should be demonstrated with the following example of adding the neuron arrays represented by rectangles to the visualization. The complete code necessary for the extension can be found in the appendix.

In a first step, a new PixiJS container needs to be added to later hold the graphics objects. Two lines have to be added to `pixiBackend` inside the `container` property:

```
loadOverviewData(networkFilePath: string) {
  // create new instance of a container
  neurons: new PIXI.Container(),

  // place the container in the main container "stage" inside
       setup()
  this.stage.addChild(this.neurons);
};
```

Next, a draw function has to be added to the `Detailview` class in the detailview.ts file. The draw function needs to perform two tasks: calculate the positions of the neurons using the `hicannPosition` as well as `Detailview`’s properties controlling the position of the already existing buses and synapses. And secondly, the rectangles representing the neurons are drawn by calling `pixiBackend.drawRectangles` and passing the neuron positions as well as specified rectangle widths and heights and a chosen fill color. To draw the neurons together with the L1 routes and synapse arrays, the draw function simply has to be included into `Detailview`'s `drawHicann` method.

To improve performance and make use of antialiasing, the rectangles should additionally be drawn as sprites. Sprites are drawn by simply calling `pixiBackend`'s `drawRectanglesSprite` method instead of `drawRectangles`.

At this point all the functionality is implemented to draw rectangles representing the neurons on the HICANNs. To include the neurons into the flow of the rest of the detailview, `Automode` needs to be adjusted. `Automode`'s start functions, as well as the set functions that handle switching between the sprite representation and the graphics objects, have to be extended, but mostly just by one or two lines of code. A checkbox to set whether the neurons should be displayed when entering the detailview in automode has to be added to the UI.

5. Discussion and Outlook

The developed software presents a solid foundation for a maintainable and easily extendible visualization of the HICANN wafer. At the current state, the application is a useful static debugging tool and allows an intuitive exploration of the wafer.

An intermediate layer that uses wafer configuration data and provides an environment for visualizing hardware properties was constructed. The configuration data is accessed with the `marocco::results` API directly in the JavaScript application, eliminating possibly error-prone conversion steps. Integration of emscripten into the existing CI (continuous integration) flow and testing of the JavaScript API to detect errors right away is planned. The JavaScript graphics library PixiJS was extended by an API specifically for the visualization of the wafer. Hardware-accelerated drawing of graphic elements on an HTML canvas facilitates smooth user interaction with a large number of elements. Still, drawing every L1 bus segment and synapse for the whole wafer at once was not feasible and hence two separate modes were developed to control the visualization. The automatic and manual mode can be used together for an intuitive exploration of the wafer but also specific examination and debugging of certain parts. L1 Routes are visualized as colored lines running over the L1 bus segments and can be easily tracked from the source to the target HICANN. The visualization also provides an immediate feedback about the utilization of different parts of the wafer. A color map intuitively shows concentrations of neuron placement and their routing across the wafer.

The application presented in this thesis is documented and an effort was made to keep the code maintainable. The code was separated into functional parts with well-defined dependencies. Writing in TypeScript instead of JavaScript enables static typing and allows defining namespaces and classes. All the TypeScript source files are compiled into a single JavaScript file to achieve both, comprehensibility for the developer and simplicity for the user.

A detailed benchmarking of the graphics library PixiJS was performed. The library includes different methods for drawing graphics and managing memory. It was found that graphics objects allocate little GPU memory. Therefore, the maximum number of graphics objects that can be drawn is more likely to be limited by other factors such as available RAM. Sprites on the other hand require a large amount of GPU memory to resolve detailed graphic elements like single synapses on an array. GPU memory can exceed the physically available VRAM by making use of swap memory which comes at the cost of reduced performance. In general, it was found, that performance can be dramatically improved by storing multiple graphic elements as one graphics object. The benchmarking holds implications for implementing the visualization software. Based on a blog post by one of the PixiJS developers [40], sprites were used in order to allow antialiasing small graphics elements. As it turns out, antialiasing is by now possible with the WebGL renderer as well. An assessment on the performance losses when using antialiasing on graphics objects could potentially eliminate the need for sprites altogether. This would make the software not only faster, but also easier to maintain. Since the loading times before the first rendering lead to a temporarily frozen visualization, preloading graphic elements into the GPU and setting their visibility when needed can lead to a better user experience. However, with a growing number of features and elements, a smart manual memory management will become necessary. Further work can be done on assessing ways to improve performance. For example, it could be investigated how memory management differs in different software and hardware environments, to support a wide range of devices.

Even though the visualization is developed as a static debugging tool, the groundwork is laid for the implementation of further features. The software can be easily extended to show more details, such as neurons, and visualize additional parameters like synapse loss or blacklisting. Once data can be retrieved from running experiments, the visualization can be used to show live snapshots of the wafer on a monitoring screen. A large but desirable new feature would be a dynamic visualization in slow-motion. As an example, neurons could be highlighted when they fire to show the global firing pattern on the wafer. For reasons of performance, all needed graphic elements should be drawn upfront and then simply turned visible or invisible, potentially animating transparency, when a neuron spikes. Another useful feature would be to mark neuron populations and highlight connections between them. In a major next step, a graph for the PyNN neural network could be drawn and connections with the hardware configuration visualized.

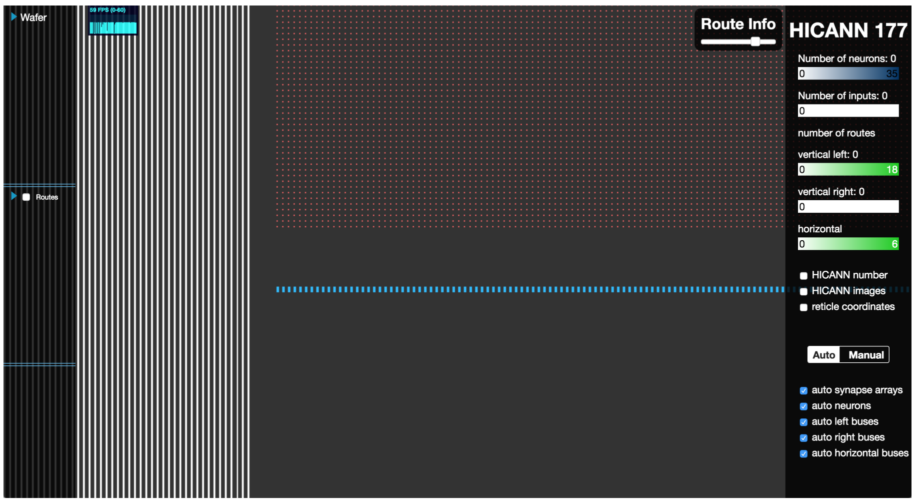
On a large system, visualization is essential not only for understanding experiments, but also the hardware itself, its behavior, limitations, and chances.

A. Appendix

File Structure

```
|-doc
|-src
: |-main.ts
: |-modules
: : |-automode.ts
: : |-detailview.ts
: : |-lookupPlot.ts
: : |-manualmode.ts
: : |-overview.ts
: : |-routes.ts
: : |-tools.ts
: : |-wafer.ts
|-build
: |-main.html
: |-main.js
: |-main.css
: |-jquery-ui.css
: |-img
: : |-...
: |-libraries
: : |-jquery-3.2.1.min.js
: : |-jquery-ui.min.js
: : |-pixi.min.js
: : |-stats.min.js
: |-Marocco.data
: |-Marocco.html
: |-Marocco.html.mem
: |-Marocco.js
: |-routes.json
```

Figure 20: A complete diagram of the file structure. The doc folder contains a complete documentation of the code as HTML document, generated with TypeDoc. All TypeScript files are collected in the src folder. The code is organized in namespaces and classes and stored in separate files. All TypeScript files are compiled into a single main.js JavaScript file in the build folder. The build folder also contains the markup and style sheets as well as third party libraries and the Marocco code. The file routes.json is used to store layer 1 routes, until they are accessible via the `marocco:results` API.



Number of neurons: 0
0 35
Number of inputs: 0
0
number of routes
vertical left: 0
0 18
vertical right: 0
0
horizontal
0 6

- [ ] HICANN number
- [ ] HICANN images
- [ ] reticle coordinates

Auto Manual

- [x] auto synapse arrays
- [x] auto neurons
- [x] auto left buses
- [x] auto right buses
- [x] auto horizontal buses

Figure 21: A screenshot of the extended detailview that includes blue rectangles as a representation of the neuron circuits. The two rectangle arrays are located below the top synapse array and above the bottom synapse array respectively. A checkbox in the right info panel allows to set the neurons visibility for the detailview in automatic mode.

Extending the Visualization

The following section describes in detail how to add rectangle arrays representing the neuron circuits to the visualization and integrate them into the automode. Figure 21 shows the resulting neuron array below the top synapse array.

pixiBackend.ts First of all, two PixiJS containers have to be created to hold the graphics elements and sprites that represent the neurons. Extend the `container` object of the `pixiBackend` namespace:

```
export const container = {
    ...
  neurons: new PIXI.Container(),
  neuronsSprite: new PIXI.Container(),

  setup: function() {
    ...
    this.detailView.addChild(this.neurons);
    this.detailView.addChild(this.neuronsSprite);
    },
  }
```

detailview.ts A method to draw two arrays of rectangles representing the neurons between the synapse arrays and the horizontal buses is needed. Add a method `drawNeurons` to the `DetailView` class:

```
drawNeurons(hicannPosition: {x: number, y: number}) {
  // calculate the neuron positions
  const synapseArrayHeight = (this.numSynapsesVertical - 1)
      * this.unitDistance + this.unitLength;
  const neurons = {
    arrayOneX: hicannPosition.x
        + (this.numBusesVertical + this.gap)*this.unitDistance,
    arrayOneY: hicannPosition.y
        + 5.5 * this.gap + synapseArrayHeight,
    arrayTwoX: hicannPosition.x +
        (this.numBusesVertical + this.gap)*this.unitDistance,
    arrayTwoY: hicannPosition.y + this.hicannHeight
        - synapseArrayHeight - 5.5 * this.gap - this.unitLength,
    xValues: [],
    yValues: [],
    widthValues: [],
    heightValues: [],
  };
  for (let i=0; i<this.numNeurons; i++) {
    neurons.xValues.push(neurons.arrayOneX + i*this.unitDistance
        );
    neurons.yValues.push(neurons.arrayOneY);
    neurons.widthValues.push(2*this.unitLength);
    neurons.heightValues.push(4*this.unitLength)
    neurons.xValues.push(neurons.arrayTwoX + i*this.unitDistance
        );
    neurons.yValues.push(neurons.arrayTwoY);
    neurons.widthValues.push(2*this.unitLength);
    neurons.heightValues.push(4*this.unitLength);
  };
  // draw the neurons both as graphics objects and sprites
  pixiBackend.drawRectangles(
      pixiBackend.container.neurons,
      neurons.xValues, neurons.yValues,
      neurons.widthValues, neurons.heightValues, "0x26baff");
  pixiBackend.drawRectanglesSprite(
      pixiBackend.container.neuronSprites,
      neurons.xValues, neurons.yValues,
      neurons.widthValues, neurons.heightValues, "0x26baff");
}
```

In order to draw the neurons together with the other elements of the HICANN when the detailview is entered, a call for `drawNeurons` has to be added to `Detailview`'s `drawHicann` method:

```
drawHicann(newHicann: number) {
  ...
  this.drawNeurons(hicannPosition);
}
```

Also extend the `resetDetailview` method to remove the neurons again:

```
resetDetailview() {
  for (let i=0; i<numChildren; i++) {
    ...
    pixiBackend.removeChild(pixiBackend.container.neurons,0);
    pixiBackend.removeChild(pixiBackend.container.neuronsSprite
        ,0);
  }
};
```

main.html Now the neurons need to be integrated into the element handling of the automatic mode. In main.html in the build folder, add a checkbox that can be set to display neurons in automode:

```
<div id="automodeCheckboxes">
  ...
  <div class="elementsCheckbox">
    <input id="autoNeuronsCheckbox" type=checkbox checked=true>
    <label>auto neurons</label>
  </div>
  ...
</div>
```

main.ts An event handler for the checkbox has to be added:

```
$("#autoNeuronsCheckbox").change( () => {
  let checked = (document.querySelector("#autoNeuronsCheckbox")
        as HTMLInputElement).checked
  if (checked) {
    automode.options.neurons = true;
  } else {
    automode.options.neurons = false;
  };
})
```

automode.ts Lastly, the automode itself has to be extended to handle the neuron objects. Add a boolean for neurons in the `options` object of the `Automode` class:

```
constructor(overview: internalModule.Overview, detailview:
        internalModule.Detailview) {
  ...
  this.options = {
    ...
    neurons: true,
  }
}
```

Also in the Automode class, extend the setDetailview and setDetailviewLevelTwo methods to set the visibility of the neuron containers:

```
setDetailview(hicannIndex: number, enabled: boolean) {
  ...
  pixiBackend.container.neuronsSprite.visible = this.options.
        neurons ? enabled : false;
};

setDetailviewLevelTwo(enabled: boolean) {
  ...
  pixiBackend.container.neurons.visible = this.options.neurons ?
         enabled : false;
};
```

Acronyms

API application programming interface 6, 12, 45

CMOS complementary metal-oxide-semiconductor 3

FPGA field-programmable gate array 6, 33

fps frames per second 35

GPU graphics processing unit 10, 35–41, 43, 45, 46

HICANN High Input Count Analog Neural Network 3, 5, 17–19, 22–25, 27–34, 44, 45, 49

JSON JavaScript Object Notation 31

L1 Layer 1 5–7, 9, 10, 15, 17–19, 23–25, 28, 31, 32, 35, 41, 42, 44, 45

SVG Scalable Vector Graphics 9

UI user interface 14, 16, 17, 20, 34, 44

References

[1] C. Mead. “Neuromorphic electronic systems”. In: *Proceedings of the IEEE* 78.10 (1990), pp. 1629–1636. ISSN: 0018-9219. DOI: `10.1109/5.58356`.

[2] C. Bartolozzi et al. “Neuromorphic Systems”. In: *Wiley Encyclopedia of Electrical and Electronics Engineering.* John Wiley & Sons, Inc., 1999. ISBN: 9780471346081. DOI: `10.1002/047134608X.W8328`. URL: `http://dx.doi.org/10.1002/047134608X.W8328`.

[3] Steve Furber. “Large-scale neuromorphic computing systems”. In: *Journal of Neural Engineering* 13.5 (2016), p. 051001. URL: `http://stacks.iop.org/1741-2552/13/i=5/a=051001`.

[4] M. Davies et al. “Loihi: A Neuromorphic Manycore Processor with On-Chip Learning”. In: *IEEE Micro* 38.1 (2018), pp. 82–99. ISSN: 0272-1732. DOI: `10.1109/MM.2018.112130359`.

[5] BrainScaleS. *Research Project.* 2012. URL: `https://brainscales.kip.uni-heidelberg.de/index.html`.

[6] Thomas Pfeil et al. “Six networks on a universal neuromorphic computing substrate”. In: *Frontiers in Neuroscience* 7 (2013), p. 11.

[7] FACETS. *Research Project.* 2010. URL: `http://facets.kip.uni-heidelberg.de`.

[8] Human Brain Project. *Research Project.* 2013. URL: `https://www.humanbrainproject.eu/en/`.

[9] J. Schemmel, J. Fieres, and K. Meier. “Wafer-scale integration of analog neural networks”. In: *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence).* 2008, pp. 431–438. DOI: `10.1109/IJCNN.2008.4633828`.

[10] 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`.

[11] R. Brette and W. Gerstner. “Adaptive exponential integrate-and-fire model as an effective description of neuronal activity”. In: *J. Neurophysiol.* 94.5 (2005), pp. 3637–3642.

[12] Henry Markram et al. “Interneurons of the neocortical inhibitory system”. In: *Nature Reviews Neuroscience* 5 (Oct. 2004), 793 EP –. URL: `http://dx.doi.org/10.1038/nrn1519`.

[13] Tobias Thommes. “Design and Implementation of an EXTOLL Network-Interface for the Communication FPGA in the BrainScaleS Neuromorphic Computing System”. Master. Universität Heidelberg, 2018.

[14] Andrew P Davison et al. "PyNN: A Common Interface for Neuronal Network Simulators". In: *Frontiers in Neuroinformatics* 2 (2008), p. 11. DOI: 10.3389/neuro.11.011.2008. URL: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2634533/.

[15] SpiNNaker. *Research Project.* 2012. URL: http://apt.cs.manchester.ac.uk/projects/SpiNNaker/.

[16] Sebastian Jeltsch. "A Scalable Workflow for a Configurable Neuromorphic Platform". PhD thesis. Universität Heidelberg, 2014.

[17] Johann Klähn. "Training Functional Networks on Large-Scale Neuromorphic Hardware". Master. Universität Heidelberg, 2017.

[18] Ecma International. *ECMAScript Language (JavaScript).* 2011. URL: http://www.ecma-international.org/ecma-262/5.1/Ecma-262.pdf.

[19] Mozilla. *MDN web docs - canvas.* [Online; accessed 5-March-2018]. 2018. URL: https://developer.mozilla.org/de/docs/Web/HTML/Canvas.

[20] Alexis Deveria and community. *caniuse.com - canvas.* [Online; accessed 5-March-2018]. 2018. URL: https://caniuse.com/#search=canvas.

[21] WebGL Public Wiki. *Main Page — WebGL Public Wiki,* [Online; accessed 5-March-2018]. 2017. URL: http://www.khronos.org/webgl/wiki_1_15/index.php?title=Main_Page&oldid=2546.

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

[23] three.js authors. *three.js - JavaScript 3D library.* 2017. URL: https://threejs.org/docs/index.html#manual/introduction/Creating-a-scene.

[24] Robert Ramey. *Boost serialization C++ library.* 2004. URL: http://www.boost.org/doc/libs/1_66_0/libs/serialization/doc/index.html.

[25] Alon Zakai and contributors. *Emscripten: An LLVM-to-JavaScript Compiler.* 2015. URL: http://kripken.github.io/emscripten-site/docs/index.html.

[26] LLVM Project. *LLVM compiler infrastructure.* 2018. URL: https://llvm.org/docs/.

[27] Stefan Seefeld David Abrahams. *Boost python C++ library.* 2015. URL: http://www.boost.org/doc/libs/1_66_0/libs/python/doc/html/index.html.

[28] Microsoft Corporation. *TypeScript.* 2017. URL: https://www.typescriptlang.org/docs/home.html.

[29] jQuery Foundation. *jQuery v3.2.1.* 2017. URL: https://api.jquery.com/.

[30] jQuery Foundation. *jQuery UI v1.12.1.* 2016. URL: https://api.jqueryui.com/.

[31] D. Brüderle et al. "A Comprehensive Workflow for General-Purpose Neural Modeling with Highly Configurable Neuromorphic Hardware Systems". In: *ArXiv e-prints* (Nov. 2010). arXiv: `1011.2861 [q-bio.NC]`.

[32] T. Harion. *3D-Visualisierung einer Abbildung von neuronalen Netzwerkmodellen auf eine neuromorphe Hardware.* Internship Report. 2008.

[33] B. Kindler. Personal communication.

[34] A. Kononov and S. Billaudelle. Personal communication.

[35] J. Bill. Personal communication.

[36] *React, JavaScrip library.* 2019. URL: `https://reactjs.org/docs/hello-world.html`.

[37] Michael Schwarz et al. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript". In: *Financial Cryptography and Data Security.* Ed. by Aggelos Kiayias. Cham: Springer International Publishing, 2017, pp. 247–267. ISBN: 978-3-319-70972-7.

[38] *Google Developer Meltdown/Spectre Update.* 2018. URL: `https://developers.google.com/web/updates/2018/02/meltdown-spectre`.

[39] *Google Chrome consistent memory metrics.* 2017. URL: `https://docs.google.com/document/d/1_WmgE1F5WUrhwkPqJis3dWyOiUmQKvpXp5cd4w86TvA/mobilebasic#`.

[40] *Post by Chad Engler (@rolnaaba) on html5games.com.* 2015. URL: `http://www.html5gamedevs.com/topic/15395-pixi-webgl-antialiasing/`.

Acknowledgments

First of all, I would like to thank Prof. Dr. Karlheinz Meier for accepting me in the Electronic Vision(s) Group. Writing my Bachelor thesis about this fascinating topic, I had the opportunity to become familiar with academic research. Equal thanks to Dr. Johannes Schemmel. The exciting research in your group is inspiring and motivating.

Special thanks to my supervisor Sebastian Schmitt for introducing me to the topic and supporting me all the way to the thesis. I also want to thank Eric Müller and Johann Klähn for thorough explanations and project vision. Many thanks to the people proofreading this thesis, Sebastian Schmitt, Eric Müller, Christian Mauch, Alexander Kugele and Andrea Hegele. Your comments helped bringing structure and fresh ideas into the document.

Finally, I want to thank all the amazing people in the "container" for a motivated and fun working environment and of course my friends, roommates and family for continued support.

Statement of Originality (Erklärung)

I certify that this thesis, and the research to which it refers, are the product of my own work. Any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, April 4, 2018

....
(signature)