Insert here your thesis' task.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Parallel sorting algorithms simulator

Bc. Van Nhan Nguyen

Department of ...(SPECIFY)
Supervisor: Ing. Michal Šoch, Ph.D.

December 30, 2021

Acknowledgements

THANKS (remove entirely in case you do not with to thank anyone)

Declaration

Citation of this thesis

Nguyen, Van Nhan. *Parallel sorting algorithms simulator.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021. Also available from: ⟨`https://github.com/nhanilicious/thesis`⟩.

Abstrakt

V několika větách shrňte obsah a přínos této práce v českém jazyce.

Klíčová slova Replace with comma-separated list of keywords in Czech.

Abstract

The thesis consists of design and implementation of a single page web application, that simulates selected (Odd-Even Sort, Shear Sort) parallel sorting algorithms. The application is developed on top of Vuetify framework, using Three.js library for 3D animation.

Keywords web application, simulation, parallel sorting algorithm, Odd-Even Sort, Shear Sort, Vuetify, Three.js

Contents

List of Figures

Introduction

Motivation

[TODO]

Problem statement

The main purpose of the application designed and implemented in this thesis is to simulate and visualize running of parallel sorting algorithms. More specifically, Odd-even sort and Shear sort algorithms were chosen to be implemented, however the application should be designed so that more algorithms can be added and simulated in the future.

Presenting such algorithms involves understanding some concepts of distributed computing.

Node

A computing node is commonly an autonomous computational entity with its own local memory, which is part of a larger computer network and is able to communicate with other nodes if connected.

Grid

A grid can be understood as a network of computing nodes. The application will simulate parallel algorithms that will be working with regular 2-dimensional (rectangular) grids. Nodes are arranged in rows and columns, have up to four neighbors and the grid can be represented with a graph where of vertices are computer nodes and edges are connections.

Parallel sorting algorithm

In a parallel algorithm computations are executed concurrently on all nodes in the grid that synchronize and pass information via communication between connected nodes. A parallel sorting algorithm reorders elements, which are evenly spread out across the nodes. Elements are passed and exchanged, resulting in an element placement specific to each algorithm, such that the elements can be easily retrieved in sorted order.

Simulation step

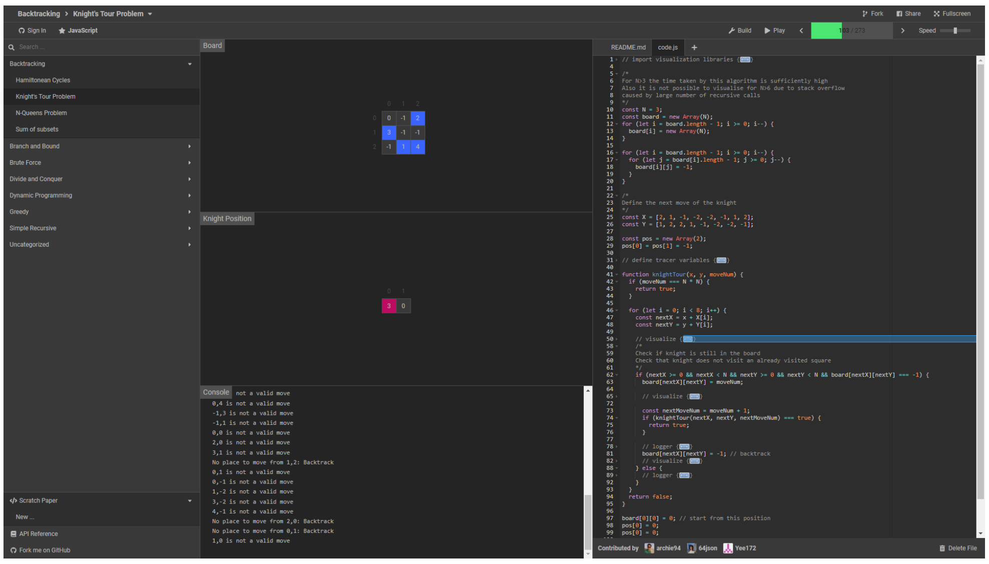
A simulation of an algorithm is an imitation of running the algorithm on some input data. One simulation step can be specific to each algorithm. For parallel sorting, it is a small set of operations ran concurrently on all nodes. One step difference is a difference between states of the grid before the step and after the step, i.e., once the operations finished execution on all nodes and are synchronized.

Chapter 1

Analysis

1.1 Existing solutions analysis

I haven't found any existing parallel algorithm simulators, however there are many other existing algorithm simulators and visualizations online. Since they are generally similar and share many features, I have selected the ones that stand out for further analysis.



```
not a valid move
0,4 is not a valid move
-1,3 is not a valid move
-1,1 is not a valid move
0,0 is not a valid move
2,0 is not a valid move
3,1 is not a valid move
No place to move from 1,2: Backtrack
0,1 is not a valid move
0,-1 is not a valid move
1,-2 is not a valid move
3,-2 is not a valid move
4,-1 is not a valid move
No place to move from 2,0: Backtrack
No place to move from 0,1: Backtrack
1,0 is not a valid move
```

```
// import visualization libraries {}

/*
For N>3 the time taken by this algorithm is sufficiently high
Also it is not possible to visualise for N>6 due to stack overflow
caused by large number of recursive calls
*/
const N = 3;
const board = new Array(N);
for (let i = board.length - 1; i >= 0; i--) {
  board[i] = new Array(N);
}

for (let i = board.length - 1; i >= 0; i--) {
  for (let j = board[i].length - 1; j >= 0; j--) {
    board[i][j] = -1;
  }
}

/*
Define the next move of the knight
*/
const X = [2, 1, -1, -2, -2, -1, 1, 2];
const Y = [1, 2, 2, 1, -1, -2, -2, -1];

const pos = new Array(2);
pos[0] = pos[1] = -1;

// define tracer variables {}

function knightTour(x, y, moveNum) {
  if (moveNum === N * N) {
    return true;
  }

  for (let i = 0; i < 8; i++) {
    const nextX = x + X[i];
    const nextY = y + Y[i];

    // visualize {}
    /*
    Check if knight is still in the board
    Check that knight does not visit an already visited square
    */
    if (nextX >= 0 && nextX < N && nextY >= 0 && nextY < N && board[nextX][nextY] === -1) {
      board[nextX][nextY] = moveNum;

      // visualize {}

      const nextMoveNum = moveNum + 1;
      if (knightTour(nextX, nextY, nextMoveNum) === true) {
        return true;
      }

      // logger {}
      board[nextX][nextY] = -1; // backtrack
      // visualize {}
    } else {
      // logger {}
    }
  }
  return false;
}

board[0][0] = 0; // start from this position
pos[0] = 0;
pos[0] = 0;
```

Figure 1.1: algorithm-visualizer.org

1.1.1 algorithm-visualizer.org

algorithm-visualizer.org[1] hosts a complex application with numerous features. The most interesting one is the ability to view and edit the simulated JavaScript code. The application could be perceived as a graphically enhanced debugger. Apart from browsing through and modifying algorithms listed in the menu, one can also create a new algorithm from scratch with own graphical representation, using deployed visualization libraries.

The user interface is split in 3 major parts: a menu selection of algorithms, output panel(s) and a code editor. The output panel(s) are generally a combination of one or more graphical contents and a textual, debug or console output. The menu can be hidden away, allowing for more workspace on the screen. The application also provides a search field to look up an algorithm from the large selection.

The simulated scenarios are fully in the hands of a user. On the other hand, there is a lack of a simple scenario setup and any kind of modification requires a certain level of qualified knowledge in order to update the relevant variables in the code.

Along with the obvious components covering most of the page, there are also other elements relevant to a simulation application. In the top right corner is a typical player interface with play/pause input, step selection and speed manipulation.

Overall, algorithm-visualizer.org is an amazingly complex application with numerous features. Most of the are however well above the scope of ambitions of the aim of this thesis.

1.1.2 SORTING

SORTING is an application hosted at sorting.at[2] and offers simulation of sequential sorting algorithms. What I found unique about this application was the option to simulate multiple sorting algorithm running alongside each other on the same input data to observe and compare the differences.

Adding another (simulated) algorithm navigates a user to a section where a specific algorithm can be selected, and its visualization can be configured. The current selection contains 17 both commonly known and less known sorting algorithms. The page also allows to setup element color shade, size and even the initial condition of the data (random, nearly sorted, reversed, few unique).

The user interface is not overwhelming and intuitive. The simulations are controlled with buttons at the bottom of the initial screen. A user can play or pause, step forwards, step backwards or go all the way to the initial (unsorted) state or the final (sorted) state. Top right corner contains buttons that allow to change the layout of the simulations.

[1] `https://algorithm-visualizer.org/`

[2] `https://sorting.at/`

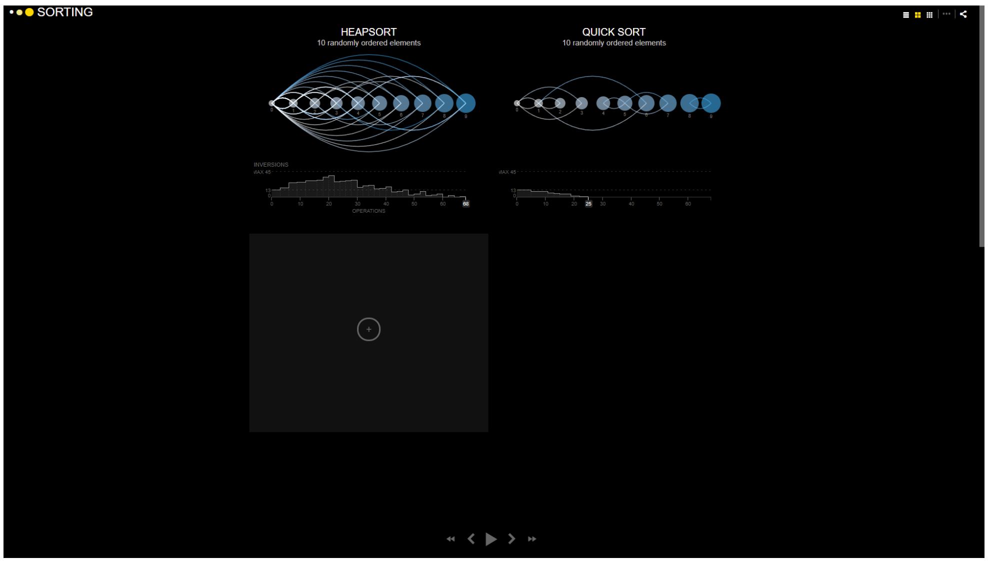
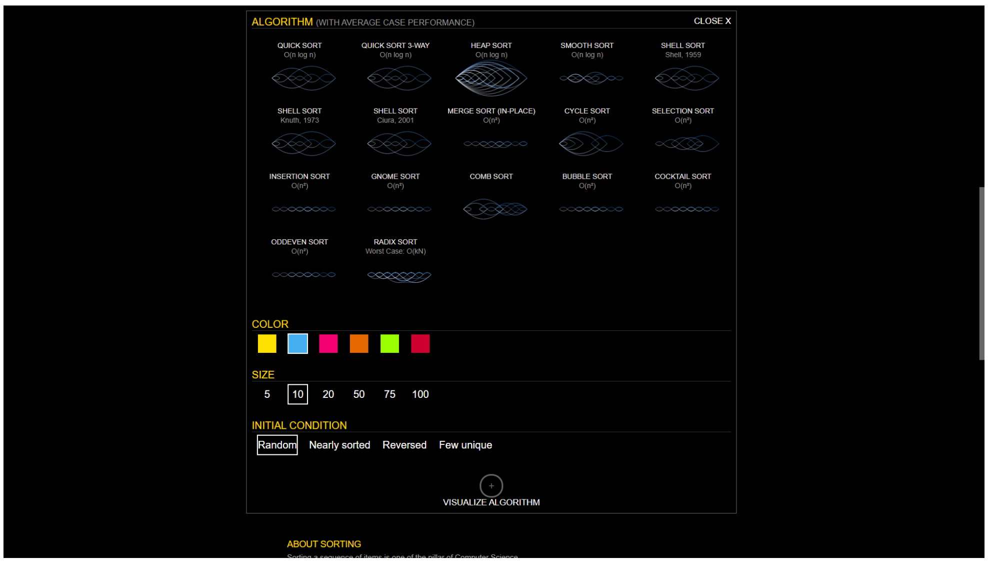


Figure 1.2: SORTING - simulation



ALGORITHM (WITH AVERAGE CASE PERFORMANCE) CLOSE X

QUICK SORT O(n log n)
QUICK SORT 3-WAY O(n log n)
HEAP SORT O(n log n)
SMOOTH SORT O(n log n)
SHELL SORT Shell, 1959
SHELL SORT Knuth, 1973
SHELL SORT Ciura, 2001
MERGE SORT (IN-PLACE) O(n^2)
CYCLE SORT O(n^2)
SELECTION SORT O(n^2)
INSERTION SORT O(n^2)
GNOME SORT O(n^2)
COMB SORT
BUBBLE SORT O(n^2)
COCKTAIL SORT O(n^2)
ODDEVEN SORT O(n^2)
RADIX SORT Worst Case: O(kN)

COLOR

🟨 [✓ 🟦] 🟥 🟧 🟩 🟥

SIZE

5 [✓ 10] 20 50 75 100

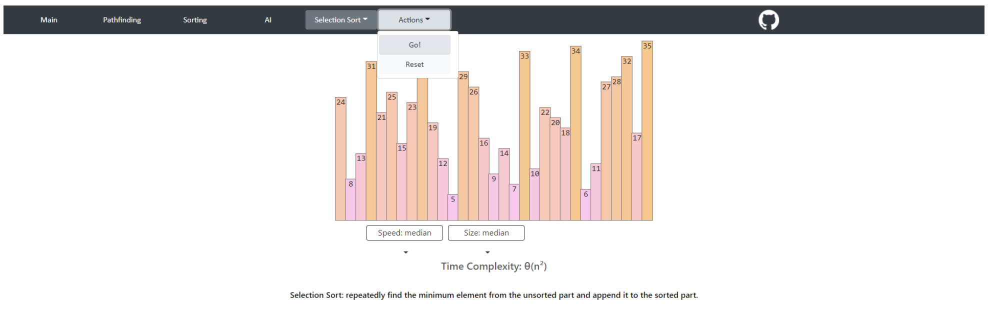
INITIAL CONDITION

[✓ Random] Nearly sorted Reversed Few unique

VISUALIZE ALGORITHM

ABOUT SORTING

Figure 1.3: SORTING - configuration



Time Complexity: $\Theta(n^2)$

Selection Sort: repeatedly find the minimum element from the unsorted part and append it to the sorted part.

Figure 1.4: Sorting Visualizer

The design is esthetically pleasing. A simulation consists of two graphical components. The state of the elements (and their current order) and number of operations completed.

The elements, which are being sorted, are represented as colored circles and their value is represented in numbers. Sorting elements means sorting the numbers from smallest to largest. The elements are further differentiated by color gradient and circle size, which also hints at the correct ordering of the values.

Furthermore, during a simulation, the application visually highlights which elements are being compared and animates their transition when their positions are being swapped.

Visually, sorted.at manages to display a lot of information to a user and also emphasize what is being compared and exchanged, without overwhelming the user with unnecessary text.

1.1.3 Sorting Visualizer

Visualizer[3] is one of many algorithm simulators hosted on github. Compared to the previous two web applications, this one is very minimalistic.

A user has limited options to set up the size of the data that the algorithm is simulated on by choosing from a drop-down menu of three items: small, median and large. The initial state of the elements is randomly generated. The other modifier is speed, again with a set of three options: slow, median

[3] `https://jasonfenggit.github.io/Visualizer/`

and fast. The simulation itself can only be started but not paused nor stopped manually. Once finished, it can be reset to original state.

The data is displayed as a bar chart where individual elements are represented as colored bars with numerical values. Once again, the contrast between two elements is emphasize by not only value, but also the color gradient and the size of its bar. Furthermore, below the simulation is a text with a simple description of how the algorithm works.

The relative simplicity of Visualizer shows that a working algorithm simulator requires only a small number of features and does not need to be a complex application to serve its purpose.

1.1.4 Summary

Analyzing algorithm simulators (or visualizers), one can notice that there are certain common features present in both the selected ones above and other existing simulators not covered in this analysis.

Among the most basic functionality is the ability to start and pause a simulation. Some simulators allow to simulate one step at a time, forwards or backwards, and manipulate the speed of the simulation.

Data is generally represented as a set or a list of natural numbers. This theoretically allows for unlimited number of elements and natural ordering. Elements are further distinguished with other visual features and the size of the data is configurable. The initial state of the date is randomly generated.

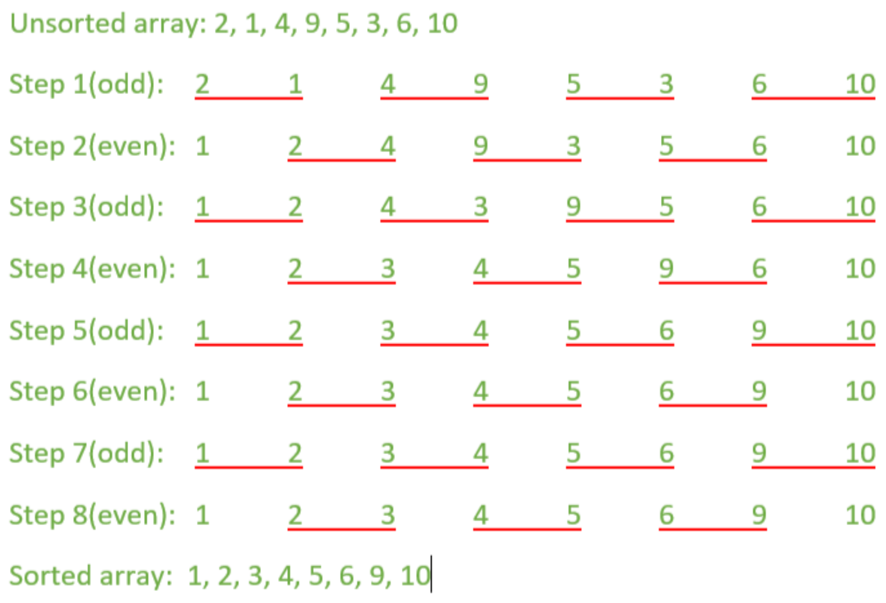
And finally, effort is put into pointing out how a simualted algorithm works using both visually and textually. For example, simulators animate elemenent transition or highlight comparison and exchange operations on element (in case of compare and exchange sorting algorithms).

1.2 Algorithm analysis

As stated before, Odd-even sort and Shear sort algorithms have been selected to be part of the developped simulator applications.To implement them or simulate them, it is important to understand how they work and what kinf od operations they execute on individual nodes.

1.2.1 Odd-even sort

Odd-even sort algorithm sorts a one dimensional array of numbers and works in phases. Every odd phase, all adjacent odd/even indexed pairs of numbers are compared and if they are in a wrong order, they are swapped. Every even phase, the adjacent even/odd indexed elements are compared. The algorithm starts with an odd phase and switches between odd and even every phase and runs until the array is sorted.


Unsorted array: 2, 1, 4, 9, 5, 3, 6, 10

Step 1(odd): 2 1 4 9 5 3 6 10

Step 2(even): 1 2 4 9 3 5 6 10

Step 3(odd): 1 2 4 3 9 5 6 10

Step 4(even): 1 2 3 4 5 9 6 10

Step 5(odd): 1 2 3 4 5 6 9 10

Step 6(even): 1 2 3 4 5 6 9 10

Step 7(odd): 1 2 3 4 5 6 9 10

Step 8(even): 1 2 3 4 5 6 9 10

Sorted array: 1, 2, 3, 4, 5, 6, 9, 10

Figure 1.5: Odd-even sort (GeeksforGeeks.org)

An array can be mapped to a one dimensional grid where each node represents one cell in an array with the same indexing. Every phase, respective adjacent nodes communicate their elemenets and exchange them if neccessary. In this case, instead of checking whether the grid is sorted, the algorithm executes n phases, where n is the number of nodes, after which the correct order is guaranteed.

1.2.2 Shear sort

Shear sort algorithm sorts a two dimensional array using Odd-even and, again, executes operations in phases. First the algorithm sorts the array row-wise by applying Odd-even sort on every row. Every even row is sorted in reverse order. After the rows are sorted, the algorithm switches to sorting the array column wise (all in the same direction). The algorithm alternates between sorting the array row-wise and column-wise until the elements are ordered in a snake-like order.

Again, a two-dimensional array can be mapped onto a two dimensional grid with rows and columns and the algorithm can be applied the same way as for Odd-even sort.

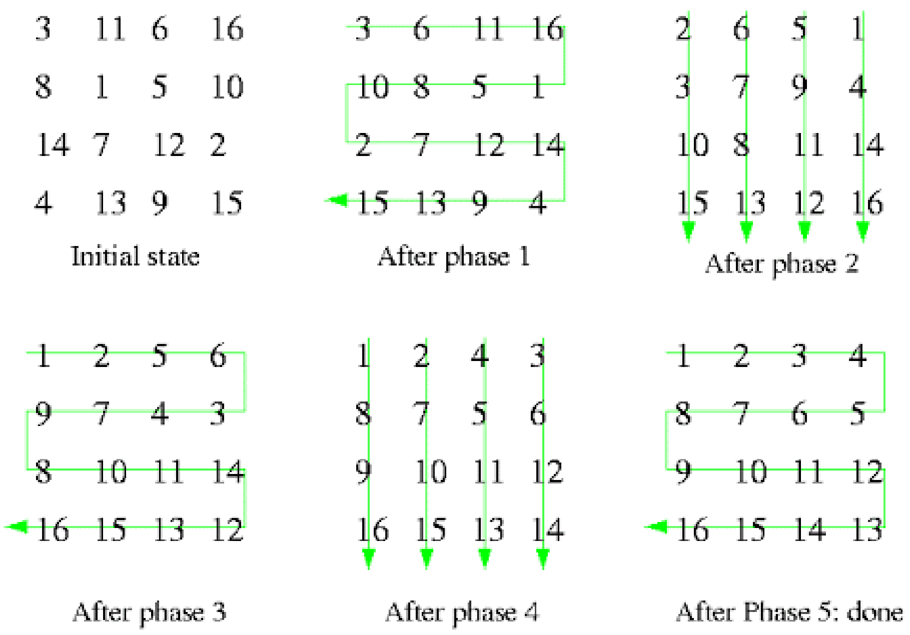


Figure 1.6: Shear sort (cpp.edu)

1.2.3 Multiple elements per node

The algorithms behave slightly differently when the number of elements exceeds the number of nodes. This results in some nodes containing more than one elements in the starting placement. This adds an extra initial phase to the algorithms, during which the elements are sorted locally.

When two nodes compare and exchange elements, the result of the operation is the same as merging the elements in a list, ordering them and then splitting the list in halves. The higher order node keeps the half with higher order elements, leaving the rest to the other node.

1.3 Target audience

The target audience for the application are university students, studying parallel processes and more specifically parallel sorting algorithms. It is meant to be a tool, used alongside regular lectures and tutorials, to visualize the algorithms and help the students understand how they work more easily. A typical user will therefore have a technical background and detailed knowledge of the subject. The main focus of the tool should therefore be simulation and visualization of algorithms themselves without overwhelming the user with unnecessary details.

1.4 Application specifications

Based on the problem definition and the previous analysis, it is possible to derive both functional and non-functional requirements of the application.

1.4.1 Functional requirements

1. The application runs as a web application and is access through a browser.
2. Algorithm Selection
   a) The application allows user to select Odd-even sort for simulation.
   b) The application allows user to select Shear sort for simulation.
3. Simulation configuration
   a) Size (width and height) of a grid, on which an algorithm will be running, can be adjusted. The grid dimension should be appropriately limited depending on which algorithm is selected. For example, Odd-even sort algorithm can only run on a one dimensional grid. The default size of a grid is 1x1.
   b) Number of elements can be specified. If no number is specified, the nubmer of elements is considered to be equal to the number of nodes.
   c) A simulation can be initialized only if it has been configured. Once initialized, the initial placement of the elements in a grid is randomly generated. However the elements have to be spread out evenly, i.e., the number of elements in each node can differ by at most one. After simulation initialization, all steps are calculated and the simulation can be played. Initial simulation step is the first step where the grid is in its initial state.
4. Algorithm simulation
   a) A simulation is either playing or paused and this state can switched. When playing, the simulation continually progresses through individual simulation steps.
   b) A simulation can be progressed one step ahead unless the simulation is at the end.
   c) A simulation can be progressed one step back unless the simulation is at the beginning.
   d) A user can skip in time to a specific simulation step.

   e) A user can nagivate to both the beginning or the end of a simulation.
   f) A simulation will stop automatically upon reaching the last simulation step.

1.4.2 Non-functional requirements

There is a heavy focus on the graphical aspect of the simulator:

1. Simulation visualization
   a) The grid on which the simulated algorithm is running is displayed during a simulation. The nodes and the connections between them are identifiable.
   b) The state of the data of the current simulation step is displayed. The elements and their current position in the grid (and in nodes) are displayed. The elemenets must be distinguishable.
   c) Transition between simulation steps is clear. For example, it should be indicated how the positions of the elements in a grid changed during a simulation.
   d) The simulator highlights which nodes communicate with each other and which elements are evaluated.
2. Simulation speed is adjustable.
3. The application must be designed in a way that allows for new algorithms to be added and simulated.

1.5 Use cases

The application has only one user role with access to all features. The only other actor is the application itself, which reacts to user action or the changes to its state.

1.5.1 Select algorithm

This case is independent of any other cases and always available to the user.

1. User views a list of available parallel sorting algorithms by expanding a drop-down menu.
2. User selects an algorithm by clicking on its respective line in the expanded drop down menu.

After these steps, grid configuration and simulation initialization become available.

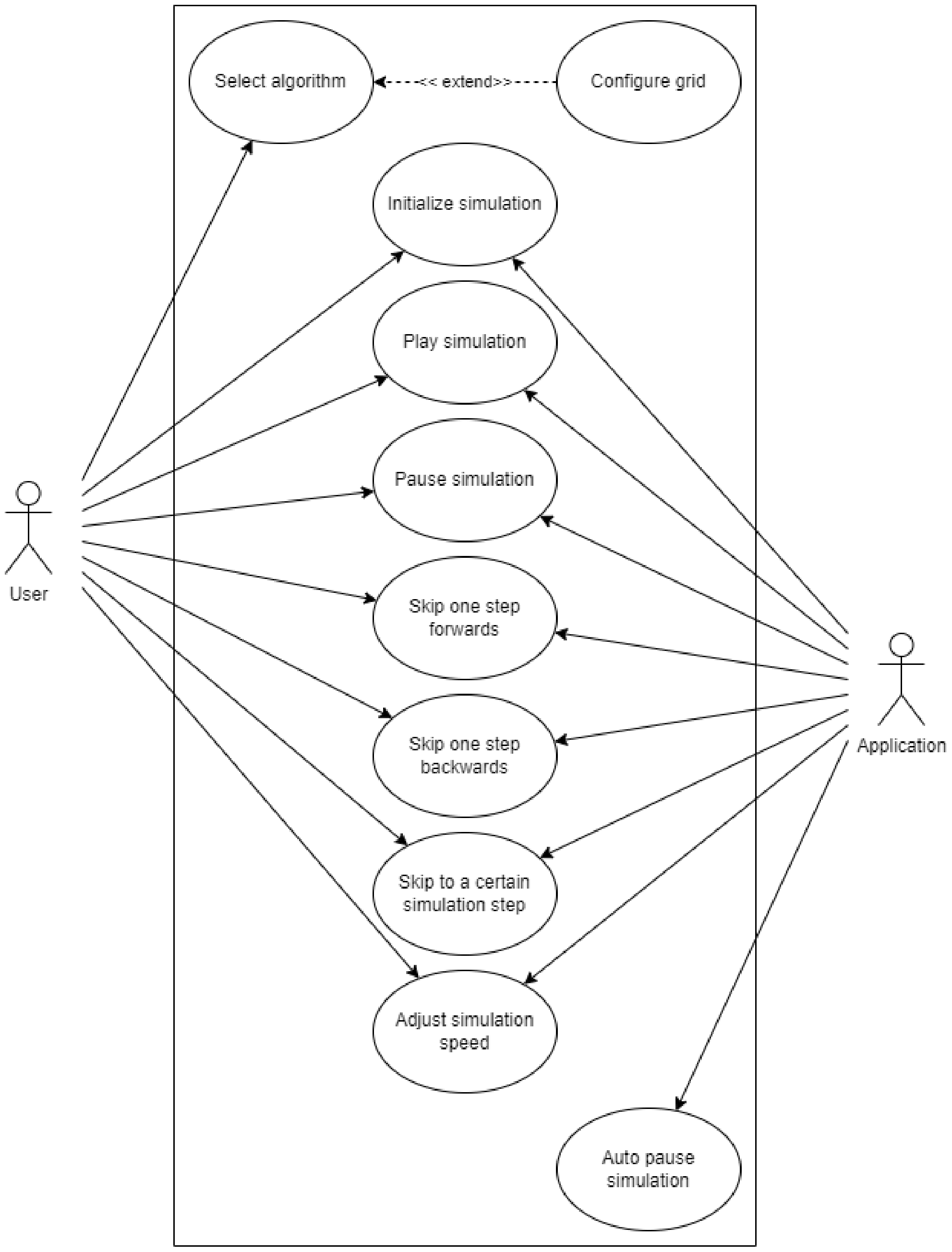


Figure 1.7: Use cases

1.5.2 Configure one dimensional grid

This scenario is only possible when an algorithm that works in a one dimensional grid is selected. The following independent steps can be completed in any order.

- User inputs grid width or leaves it at a default value.
- User inputs number of elements or leaves at default value.

1.5.3 Configure two dimensional grid

This scenario is only possible when an algorithm that works in a two dimensional grid is selected. The following independent steps can be completed in any order.

- User inputs grid width or leaves it at a default value.
- Set grid height or leave at default value.
- User inputs number of elements or leaves at default value.

1.5.4 Initialize simulation

This scenario is only possible when an algorithm is selected.

1. User finalizes simulation configuration and initializes it with a simple click of a button.
2. The application calculates the simulation steps.
3. The application displays the initial step and puts the simulation in a paused state.
4. The application enables user input for simulation manipulation.

After this, all the following use cases related to timeline manipulation and simulation speed become possible.

1.5.5 Play simulation

This scenario is only possible if a simulation has been initialized and is in a paused state.

1. User clicks a play button.
2. The simulation state changes from paused to unpaused (playing).
3. The application starts progressing steps, based on the simulation speed, and displays transition changes.

1.5.6 Pause simulation

This scenario is only possible if a simulation has been initialized and is in an unpaused (playing) state, and is a reverse use case to the previous one.

1. User clicks a pause button.
2. The simulation state changes from unpaused (playing) to paused.
3. The application stops progressing steps and stops displaying transition changes.

1.5.7 Skip one step forwards

This scenario is only possible if a simulation has been initialized.

1. User clicks a skip forward button.
2. The application evaluates whether the simulation is at the end or not. If not, the simulation step is changed to next one.
3. The application updates the displayed grid to reflect the step change.

1.5.8 Skip one step backwards

This scenario is only possible if a simulation has been initialized.

1. User clicks a skip to previous button.
2. The application evaluates whether the simulation is at the beginning or not. If not, the simulation step is changed to previous one.
3. The application updates the displayed grid to reflect the step change.

1.5.9 Skip to a certain simulation step

This scenario is only possible if a simulation has been initialized.

1. User selects desired simulation step from an input range.
2. The application changes current simulation step to the one selected.
3. The application updates the displayed grid to reflect the step change.

1.5.10 Adjust simulation speed

This scenario is only possible if a simulation has been initialized.

1. User adjusts simulation speed by moving a slider.
2. The application updates the inner state.

The results of this action are visible when a simulation is playing.

1.5.11 Auto stop simulation

This scenario is triggered when a simulation is initialized, in an upaused (playing) state and the simulation has reached the last simulation step.

1. The simulation state is changed from unpaused (playing) to paused.
2. The application stops progressing steps and stops displaying transition changes.

CHAPTER 2

Design

2.1 User interface

Previously defined specifications and use cases also outline what kind of user input is expected. It is therefore possible to create a mock-up of a user interface as seen in figure 2.1.

1. Left panel contains input fields for simulation configuration. Initially, only algorithm selection is enabled. Once an algorithm is selected, other input elements become active.
2. A drop-down menu showing a list of all available algorithms to be simulated and allowing algorithms selection.
3. An input field for entering the grid width. It becomes active once an algorithm is selected and only allows positive integer values.

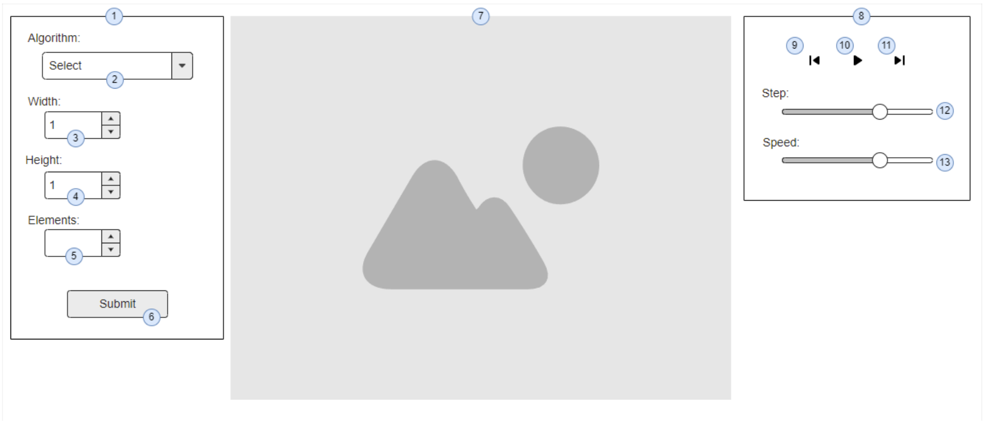


Figure 2.1: User interface mock-up

4. An input field for entering the grid height. It becomes active only if an algorithm that works on two dimensional grids is selected. It only allows positive integer values.

5. An input field for entering the number of elements in the grid. If left empty, the number of elements is considered to be the same as number of nodes. Only positive integer values are allowed.

6. A submit button, which triggers input validation and if successfu, simulation initialization. As described before, during initialization, the application generates a grid and initial elements placement. After the first time a simulation is initialized, other panels become activated and/or enabled.

7. Center panel contains a rendered area where the simulation is visualized. This is further described below.

8. The right panel contains input for simulation control. Initially, these inputs are disabled until the first time a simulation is initialized.

9. Skip to previous simulation step button displayed as an icon.

10. Play/pause button displayed as an icon. Depending on whether the simulation is in a paused or an unpaused state, this button will show either a play or pause icon repectively.

11. Skip to next simulation step button displayed as an icon.

12. A slider input that serves as simulation step selector. It allows a user to navigate to the start or to the end of a simulation by dragging the slider to the start or the end respectively. Minimum and maximum are calculated during simulation initialization and the slider position reflects at which step a simulation is at.

13. A slider input that serves as simulation speed modifier. The speed will be on a scale from 1 to 3, with 3 being the fastest. The slider in its default position will be of value 2.

2.2 Simulation visualization

Next is the visualization of the simulation. The grapphical representation of a grid has to be intuitive and self-descriptive. Figure 2.2 shows an example of a 2x2 grid with my chosen graphical design.

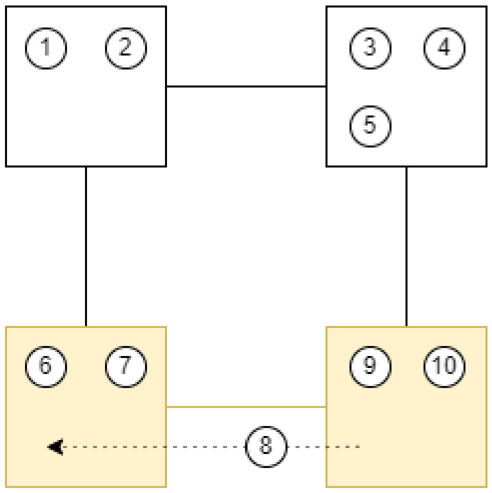


Figure 2.2: Grid mock-up

2.2.1 Grid

Grid nodes are displayed as squares or cubes, to represent containers. Lines between two squares illustrate connections, through which the nodes communicate and exchange elements. All nodes in a grid are of the same size and all connecting lines are of the same length. Furthermore, the length of a connecting line is the same as length of a side of a square, representing a node.

This simplistic design should allow scaling while also keeping the grid easy to read.

2.2.2 Elements

Elements are represented as circles with numbers, which identifies them and also indicates their ordering. Individual element size scales depending on how many elemements there are in its associated node. The elements all fit in a node and must not overlap

2.2.3 Indicators

Nodes and highlighted with a contrasting color to indicate that they are executing local operations or communicating with other nodes. Connecting lines are also highlighted to indicate which nodes are communicating with each other.

Highlighting occurs as soon as simulation step changes and the activity they illustrate are the activities happening in order to arrive at the state of the grid in the next step.

2.2.4 Animations

Element transition is animated to easily track both the original position and the position of the element in the next step. If the element size differs in the destination and origin nodes, the size is also gradually changed and is part of the animation.

The transition occurs during the delay between two simulation steps (while the simulation is playing). The animation starts after a delay and ends some time before to add a "settling" effect to an element. This is prevents the simulation from becoming a chaotic continuous motion of elements by adding a pause between transitions. This pause is further defined during the implementation phase.

2.3 Technologies

All the previous steps show that all the functionality of this web application fits on one page and there is no need for any kind of routing. The application does not require any communication with the server, that is hosting it. All the functionality can be delivered to the client with a single load. Once delivered, the application can run fully client-side.

These are all characteristics of a Single-page application (SPA), which is a recent trend replacing the previously common Multi-page applcations. Amongst the most popular web development frameworks are ones, that are based on building SPAs.[4] As with other frameworks, these make the development easier by taking away the neccessity of reimplementing the most common functionality. Developers can only focus on the business logic of an application, making the code shorter and more maintainable.

Additionally, changes on a Single-Page application are redrawn dynamically (on the client side). This makes transitions smoother and makes a website feel like a native app. This can be commonly observed on common social platforms and other website like Google, Facebook, Twitter and others.

2.3.1 JavaScript

The dynamic aspect of a web page is achieved by delivering all the neccessary HTML, JavaScript and CSS code to the client's browser. While it is possible to develop web applications in various languages, it is mainly these three technologies that are used to develop the front-end. That is also what is delivered to the client-side and thus the majority of what is developed in an SPA.

There is however an alternative technology called WebAssembly, which also runs on the client side. It is a low level compiled language with the

[4]`https://towardsdatascience.com/top-10-in-demand-web-development-frameworks-in-2021-8a5b668be0d6`

aim of being fast. In comparison, JavaScript is an interpreted language that runs in a browser using a Just-in-time compiler. This allows programmers to develop high-performance front-end in other languages, such as C or C++.

Nevertheless, the technology is relatively recent. It was released in 2017[5] and as of 2021, it is still not fully supported by all the web browsers.[6] Moreover, the common frameworks are all JavaScript based, and so will be the development of this project.

2.3.2 Vue

A large framework popularity means there is a large community using it and a large demand on constant maintenance and improvement. Such a community also extends the framework with third party libraries and plugins that bring even more functionality or compatibility with other technologies.

Among the most popular SPA frameworks are Angular, React and Vue.js.[7,8] I have decided on an implementation on top of Vue (short for Vue.js) framework due to its properties that are advantageous for this application.

- Model-View-Controller pattern allows for separation of concerns as well as a well structured code. Separating data, logic and display allows easier maintainability and also extensibility, which is one of the rquirements for this project. It should be noted that Angular is also an MVC framework.

  React, on the other hand, functions as a library and has no predefined structure and the software architecture has to be designed from the ground

- The framework itself is small in size and does not contain many built in features. Instead, needed functionality is added by extending the framework by existing third party solutions. The lightweight property faciliatiates faster loading and gives the developer control over what is loaded on the client side. In comparison, Angular is a heavyweight solution that is rich with many native features. When an Angular application is loaded, it preloads all its libraries regardless of whether they are utilized or not, which impacts loading time.

- Both Vue and Angular have built in two-way data binding. This means that any changes on the presentation (View) layer update the model and vice versa. This is of course advantageous if some part of a view is supposed to always reflect the state of the model without the need of

[5] `https://en.wikipedia.org/wiki/WebAssembly`

[6] `https://caniuse.com/wasm`

[7] `https://medium.com/javascript-scene/top-javascript-frameworks-and-tech-trends-for-2021-d8cb0f7bda69`

[8] `https://tsh.io/blog/javascript-trends-in-2021/`

implementing additional update logic. Again, it should be noted that Angular also has

- As a bonus, the framework is the easiest out of the three to pick up. The templates are in classic HTML, which makes makes them readable and easier to design with limited web development knowledge.

  Furthermore, Vue uses "pure" JavaScript with support for TypeScript (extended Javascript with syntax for types). Angular also supports development in both languages, however, TypeScript is neccessary in order to fully utilize all of Angular's features. In contrast, React highly recommends using JSX[9], which is Javascript extended with XML syntax.

  These differences make Vue stand out for beginner web developers.

A potential disadvantage to using Vue, if compared to the other two mentioned alternatives, is it's lower popularity, which means less communtiy support and less available third party libraries. This is of course irrelevant, since it still belongs in the top web development frameworks and its popularity in the west has been on a rise.[10]

2.3.3 Vuex

Vuex is a state management pattern and a library for Vue.js applications.[11]. Its main features are:

- It stores the global state of the application in a centralized **store**. These can be read from anywhere within a Vue application through *getter* methods, however they can only be changes with mutations and actions described below.
- **Mutations** are atomic operations that modify the state of an application. They ensure that the state is changed in a predictable and correct (predefined) fashion. Mutations are *commited* from the rest of the application and can not be commited from inside other mutations.
- **Actions** are more complex operations that can commit mutations. They generally contain more complex business logic and are *dispatched.*
- The store can be split into smaller logical units, called **modules**. These modules groups state variables together with related getters, mutations and actions under a common namespace.

[9] `https://reactjs.org/docs/introducing-jsx.html`
[10] `https://statisticsanddata.org/data/the-most-popular-javascript-frameworks-2011-2021/`
[11] `https://vuex.vuejs.org/#what-is-a-state-management-pattern`

- Using Vuex library functions, store variables and functions can be easily mapped onto local variables and methods. This prevents unnecessary implementations of getters and setters and keeps the code clean and readable.

A clear example of a global state variable would be whether a simulation is paused. Using a centralized store, that is accessible from anywhere, prevents us from having to pass the value between various components.

2.3.4 Vuetify

Therer are no explicit requirements for the visual design of the User Interface. Instead of designing the color, the size or special effects of the GUI from scratch, I have decided to pick one of the many existing Vue UI libraries.

These extensions to a framework are generally easy to set up and are delivered with complete styling, layouts, icons, etc., making it easier to focus on the application logic and less on the input field design.

For this project I have chosen Vuetify for the following reasons:

- It adheres to Material Design guidelines, developed by Google. This, among other advantages, implements responsive animations and transitions,[12] which is a must in a world of varying device sizes.

- Secondly, apart from delievered styling, it also offers a selection of designed page layouts, some of them applicable for the application based on the previous GUI mock-up.

- Finally, the framework is well documented with a community backing and many existing solutions online that can serve as an inspiration.

2.3.5 Three.js

It was very difficult to find the right JavaScript graphics library for rendering a grid, regardless of a framework compatibility. Most libraries I have found were designed to visualize charts and none were designed to easily visualize graphs with edges and vertices. Furthermore, given the custom design of the grid, it makes sense to resort to low level drawing.

Three.js is a JavaScript library and API for 3D animated graphics chosen as the technology to use for visualization of the simulation. It combines and encapsulates both the power of HTML Canvas element and WebGL (short for Web Graphics Library), an API for rendering 2D and 3D Graphics roughly based on OpenGL. Moreover, it supports hardware accelerated graphics through local graphics processing unit.

[12] `https://material.io/`

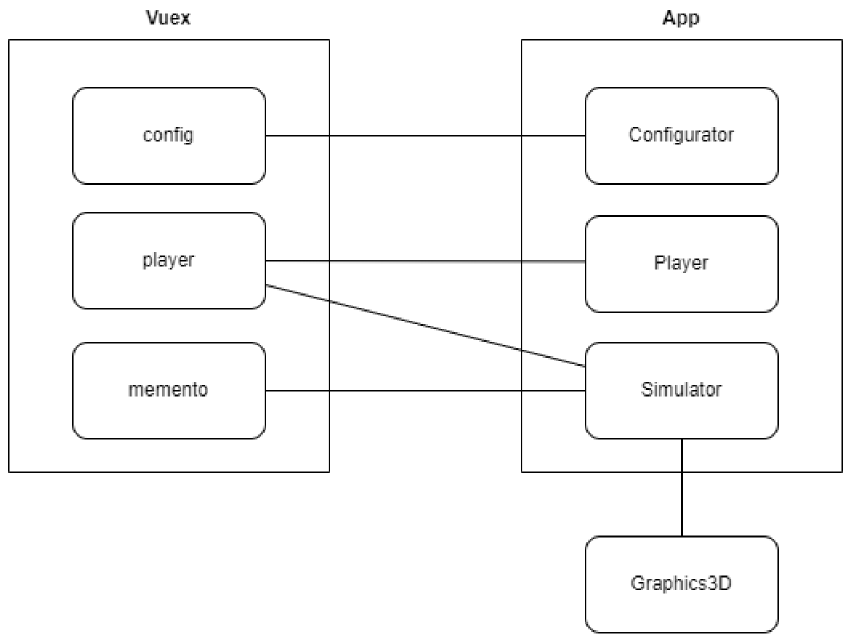


Figure 2.3: State modules and components

2.4 Application structure

With the technologies at hand, it is now possible to design the code structure. Figure 2.3 shows Vuex modules (on the left) and Application components (on the right), the main building blocks on the application. Graphics3D is a complex class, that calculates and renders the simulation 3D graphics inside the Simulator component.

2.4.1 State modules

The state consists of three modules. They are designed to work as a logical unit without dependency on the others. This allows the modules to be further expanded or improved without too much impact on the others.

- `config` module stores the current configuration state, i.e. the selected algorithm, grid size and number of simulated elements.
  - Selected algorithm that is being simulated.
  - Grid size, i.e. width and height.
  - Number of simulated elements.
- `player` module stores the current state of the player component:
  - Whether the simulation is playing or paused.

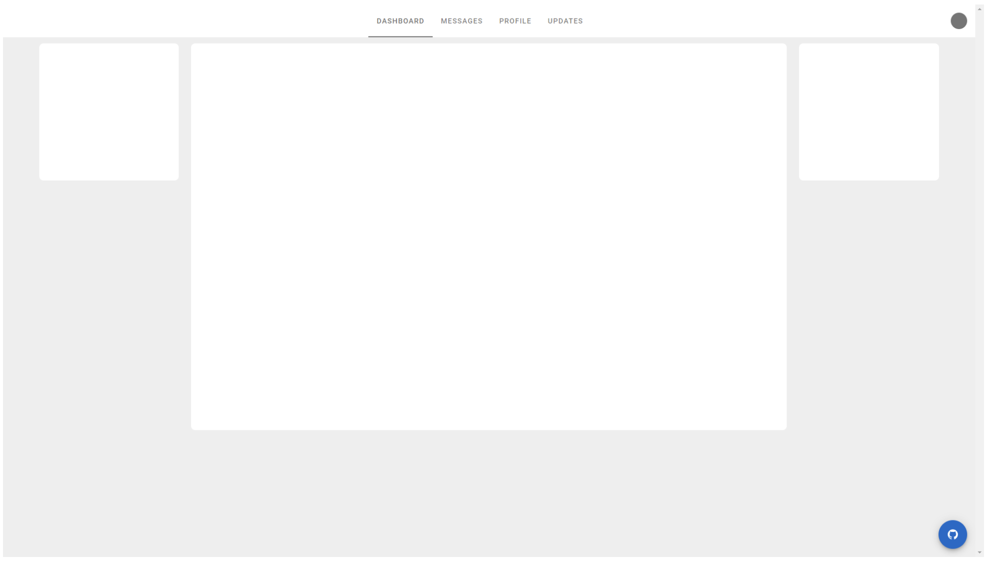


Figure 2.4: Three Column wireframe (vuetifyjs.com)

  - The point of time the simulation is at.
  - The speed of the simulation.

- `memento` module stores all the simulation steps.

As described before, module mutations must ensure state validity.

2.4.2 Components

Components are the basic building brick in Vue.js. Components generally represent a part of a web application formed from three parts: template, script and style; trough which they define how they look and how they work.

A component can be a part of another component, establishing a parent-child relationship, through which they can pass parameters and communicate. Another way of communicating is via state changes and state change triggers.

This application consists of four components:

- **App** component is the base component of the application. Its main purpose is the page layout and placement of other components. The layout will be based of the ”Three column” Vuetify wireframe,[13].

  It will also contain the main application loop, which sends a signal to other components to trigger their respective update logic.

[13]`https://vuetifyjs.com/en/examples/wireframes/three-column/`

- **Configurator** component implements user interface for simulation configuration, designed above. It is located in the left column of the application layout.

  The main functionality of the component is user validation.

  Unlike the rest of the components, this one is independent of any state, and only updates the configuration state.

- **Player** component implements user interface to navigate through a simulation. The visual elements (the input sliders) must match the state stored in the `player` state module.

  In reverse, any user input must appropriately update the application state. This is easily achieved by combining the the two-way binding property of Vue.js and mapping functionality of Vuex.

  The component is located in the right column of the application layout.

  Every application loop, the *Player* component updates the `player` state module, based on whether a simulation is paused or playing.

- **Simulator** component is a container component for rendered graphics. It communicates `player` state module changes and passes relevant calculated data from `memento` state module to `Graphics3D`, which then visualizes it.

  The change detection is, again, implemented simply by taking advantage of the mapping functionality of Vuex.

  The component is located in the center column of the application layout.

  The *Simulator* component sends new data to `Graphics3D` every application loop.

2.4.3 Graphics3D class

Rendering logic is a generally complex process and it has been separated from the *Simulator* component. Another reason for the separation is that the rendering logic can be replaced with another, providing the option of different visualizations simulator visualizations. The application can offer a user 2D visualization instead of a 3D one, or render graphics using another library.

Graphics3D class will visualize the simulation using the aforementioned *Three.js* library.

The class initializes graphics objects as part of simulation initializes, once the `config` state module is updated. The objects are only initialized, since they do not change during the simulation, they only change their position.

The object positions are calculated from data passed through the *Simulator* component.

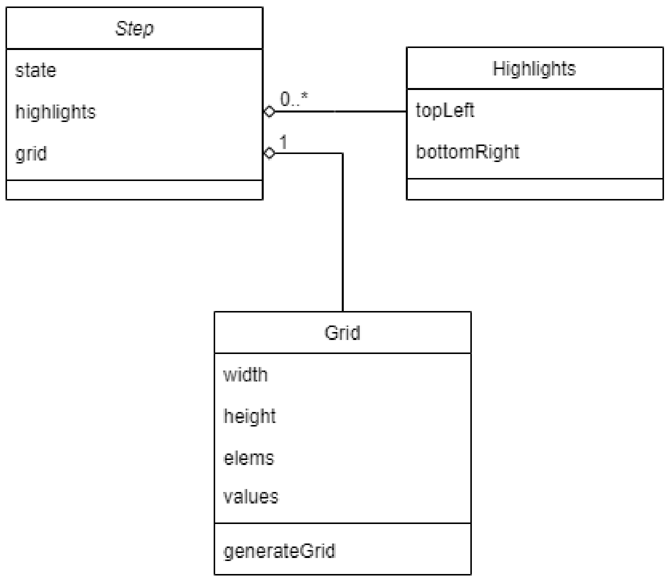


Figure 2.5: Data classes

2.4.4 Data structures

The `memento` state module stores all simulation steps of a simulation. Each step must include information about element placement in the grid and which nodes are highlighted. This data is passed to Graphics3D class, which interprets it and renders visualizes it, and is defined in classes `Grid`, `Highlight` and `Step`.

- `Grid` class simply represents a grid. Among its attributes must be grid size (width and height), number of elements and the actual element placement.

  `generateGrid` is a static function that returns a new grid, given a size and number of elements. The element placement is random each time while maintaining an even spread.

- `Highlight` class encapsulates rectangle coordinates. When passed together with a grid, any nodes and connections within the rectangular bounds are meant to be highlighted.

  Currently, the only visual indicator of what is happening during a simulation, apart from element transitions, is node and connection highlighting. This very simple sctructure is put in place for potential functional improvements in the future.

- `Step` class is a simple structure combining the `Grid` class and `Highlight` classes together, which is then visualized. The highlighting reflects the

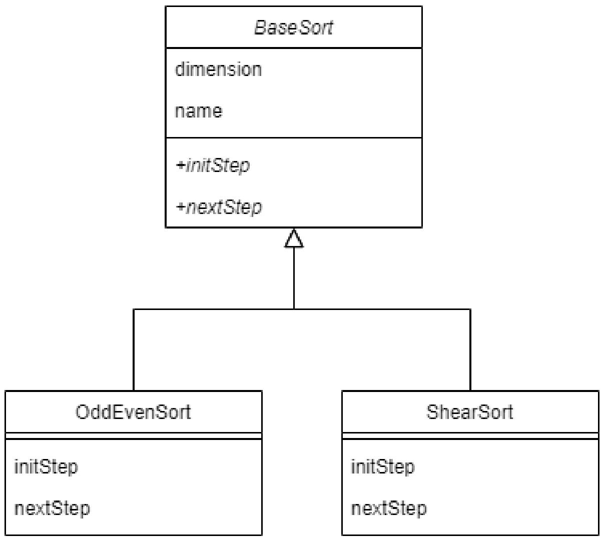


Figure 2.6: Sort algorithm classes

nodes that are being emphasized during a transition from a previous step.

Another class attribute is `state`. This attribute holds custom information for step calculation and its contents and interpretation is defined by each algorithm.

2.4.5 Sort algorithm representation

As seen in 2.6, there is an abstract class `BaseSort`, which is a parent of all sort algorithm classes. It has two attributes, which are inherited:

- `dimension` - this integer indicates the maximum dimension of a grid, the sort algorithm is albe to execute on. For the Odd-even sort algorithm, this value is 1, and for Shear sort algorithm, this value is 2.
- `name` - this is the display name of the sort algorithm. When an algorithm is integrated into the application, this name will be displayed in the drop-down menu for algorithm selection.

`BaseSort` also declares two static abstract functions, which are used to calculate simulation steps:

- `initStep` - this function takes a grid as an argument and returns it encapsulated in an initial step of an algorithm. This allows to set up the initial value of `state` attribute, which is declared in `Step` class.

This is important for algorithms that work in phases, which is a case for both Odd-even sort and Shear sort algorithms. It is possible to take advantage of the fact, that JavaScript is an untyped language. The `state` attribute has no limitation to how the information about phases are passed and interpreted, and it is in full control of a developer of a new algorithm instance.

- `nextStep` - this function calcultes a new simulation step following another step, which is given as the function argument, and contains the core logic of an algorithm.

  If there are no more simulation steps, i.e. the give step is the final one, it returns a `null` value.

Both of these functions are be implemented in `OddEvenSort` and `Shearsort` classes, and must be implemented for every future child classes of `BaseSort`.

Chapter 3

Implementation

The source code structure is listed in Appendix B. This chapter describes the implementated logic and technical solutions in detail.

3.1 State modules

Application state is implemented four files placed in `store` directory. The `index.js` file simply exposes the modules to the rest of the application. The rest contain individual state module implementations described below.

3.1.1 Namespacing

By default, actions and mutations are still registered under the global namespace - this allows multiple modules to react to the same action/mutation type.[14]

This is the opposite of what is desired from the design. The state modules of this application are more self-contained and independent and each module is namespaced with a simple line:

```
namespaced = true;
```

3.1.2 `config.js`

`config` module stores information about the current simulations configuration. This configuration state consists of:

- `algorithm` - currently selected algorithm.
- `width`, `height` - Simulation grid dimensions.
- `elems` - number of elements in the grid.

[14]`https://vuex.vuejs.org/guide/modules.html#namespacing`

3.1.2.1 Getters

Special getters are implemented for the `config` module:

- `algorithms` - generates a list of all algorithms as an array of objects with `text` and `value` properties. This format is compatible with the native Vuetify drop-down menu element.

  The algorithms are imported from `algorithms` directory, where they are placed. This is described in more detail later on.

- `config` - returns the state properties packed into one object to easily pass the configuration to other modules or components.

3.1.2.2 Mutations

Simulation configuration can be updated through module mutations and actions. The mutations must maintatin configuration validity.

- `setAlgorithm` - mutation that sets the currecntly selected algorithm. It resets all other configuration variables to 1.
- `setWidth` - mutation that sets the grid width to a new value. The new value must be a positive integer.
- `setHeight` - mutation that sets the grid height to a new value. The new value must be a positive integer and the algorithm must work on multi dimensional grids.
- `setElems` - mutation that sets the number of elements in the grid. The new number must be a positive integer.
- `setConfig` - action that sets all variables of the configuration state. This is simply done by consecutively committing changes through respective mutations. A new algorithm value is committed first.

3.1.3 `player.js`

`player` module stores the current player state of the simulation.

- `enabled` - this boolean variable indicates whether the Player component is enabled (grayed out and unclickable). When no simulation is initialized yet, there is nothing to be played and the play/pause buttons have no purpose. As soon as a simulation is initialized, these buttons are enabled and available to the user.
- `paused` - this boolean variable indicates whether the simulation is paused or not. Default value is *true*, since every simulation is in a paused state after initialization.

- `t` - this represents a point in time of a simulation as a a non negative decimal number, which can not exceed `t_max` described below. The simulation is at the beginning and show the initial simulation step when `t` is equal to 0.

  The integral part of the number ($\lfloor t \rfloor$) indicates the current step, the simulation is in (indexed from 0). The fractional part of the number represents transition between two steps and affects element position for animation purposes.

- `t_max` - this is the maximal value of `t` and reflects the total number of simulation steps, excluding the initial step.

- `delta` - this indicates by how much is `t` changed every application loop when playing.

3.1.3.1 Mutations

Player state can be updated through these mutations:

- `setEnabled` - sets the value of `enabled`.

- `setPaused` - sets the value of `paused`.

- `setT` - sets the value of `t`. If the new value is not valid, the mutation does no make any changes to the state.

- `modifyBy` - adds a specified the value to the value of `t`. If the new value of `t` would equal or exceed the `t_max`, it is set to the value of `t_max` instead and player is paused (`paused` is set to *true*).

  If the new value of `t` would be lower than 0, it is set to 0 instead.

- `incrementTurn` - if the value of `t` is lower than the maximum the value `t_max`, it will set the value of `t` to the closest larger integer. If the new value is equal to `t_max`, the player is paused.

- `descrementTurn` - if the value of `t` is greater than 0, it will set the value of `t` to the closest lower integer.

- `setMaxTurn` - sets the value of `t_max`. The new value has to be a non negative integer. If `t` is lower than `t_max`, `t` is set to the same value.

- `minimizeTurn` - sets the value of `t` to 0.

- `maximizeTurn` - sets the value of `t` to the current value of `t_max`.

- `setDelta` - sets the value of `delta`.

3.1.3.2 Actions

Player state can also be updated through these (more logical) actions:

- `enable`, `disable` - set `enabled` to *true* or to *false* respectively, by committing the `setEnabled` mutation appropriately.
- `play`, `pause` - set `paused` to *false* or to *true* respectively, by committing the `setPaused` mutation appropriately.
- `togglePause` - negates the boolean value of `paused`.
- `reset` - resets the player state to an uninitialized state, i.e. disables it, pauses it and sets both `t` and `t_max` to 0.
- `modifyByDelta` - commits `modifyBy` mutation with `delta` as its parameter.

3.1.4 `memento.js`

`memento` module stores calculated simulation steps, that are then displayed to the user during a running simulation.

- `steps` - an array of all simulation steps. By default holds an empty array.
- `algorithm` - selected algorithm used for step calculation. By default, this state has an `undefined` value.
- `complete` - this boolean value indicates whether all simulation steps have been calculated. By default, this state is set to `true`.

3.1.4.1 Getters

Special getters are implemented for the `memento` module, because the simulator does not need to read all the simulation steps at once, but only retrieves one at a time:

- `size` - returns the size of the `steps` array, i.e. the total number of calculated steps.
- `step` - returns a single step, given its array index as an argument. The function returns `null` is the index is out of bounds.

3.1.4.2 Mutations

`memento` module has simple mutations allowing state initialization and individual step calculation.

- `clear` - resets the state to default values, as described above.
- `init` - initializes the first simulation step of a simulation, given a simulation configuration (selected algorithm, grid dimensions and number of elements), only if the state is cleared, i.e. the state holds default values.

  This is also a point where a grid, on which the selected algorithm will run, is randomly generated.

  ```
  state.steps.push(algorithm.initStep(Grid.generateGrid(
    width, height, elems)));
  state.complete = false;
  ```

- `calcNext` - if the the calculation is not complete, i.e. `complete` is `false`, this mutation calculates the next step and adds it to the list. If there are no more steps to be added, sets `complete` to `true`.

3.1.4.3 `init` action

`memento` module defines two actions. First of them being `init`, which is a more logical alternative of the `init` mutation, combining it with the `clear` mutation:

```
init({commit}, config) {
  commit('clear');
  commit('init', config);
}
```

The simulation configuration has to be passed to the action in order to initialize the state. During the development, I have figured out how to access the state of another module from a different one. An alternative code, that would read the configuration from `config` module directly is:

```
init({commit}, rootGetters) {
  commit('clear');
  commit('init', rootGetters['config/config']);
}
```

However, this would tightly couple the two modules, which is in conflict with the design, and the logic of passing the configuration is moved to the *Simulator* component.

3.1.4.4 `calc` action

The second action is named `calc` and provides an interface to calculate all of the steps in a simulation:

```
calc({commit, state}) {
  return new Promise((resolve) => {
    setTimeout(() => {
      while (!state.complete) commit('calcNext');
      resolve();
    }, 1000)
  })
}
```

This piece of code takes advantage of a basic difference between mutations and actions. While mutations are atomic, committed operations and must be synchronous[15], actions can be asynchronous and/or can contain an arbitrary number of asynchronous operations.[16].

In this case, `calc` is an asynchronous action, that returns a *Javascript* built-in `Promise` object, which represents a ”promise” of completion of an ansychronous operation.[17] The object contains not only the executed operations themselves, but can also contain the results of the operations in case of both success or failure. The results can be processed after the eventual completion of the execution.

This allows the full step calculation to run asynchronosly while the application can progress with, for example, displaying the first simulation step to the user or enabling and/or activating other user interface elements. This is useful for more complex simulations with numerous steps.

3.2 Components

All implemented Vue components are stored as `.vue` files in the `components` directory apart from the base *App*, the application entry point, located directly in the base `src` folder.

3.2.1 Component structure

The basic structure of a single file Vue.js component is split into three parts, denoted by their respective tags:[18]

- `<template>...</template>` - visual template of a component. This is by default designed in HTML.

[15] `https://vuex.vuejs.org/guide/mutations.html#mutations-must-be-synchronous`
[16] `https://vuex.vuejs.org/guide/actions.html`
[17] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises`
[18] `https://vuejs.org/v2/guide/single-file-components.html`

- `<script>...</script>` - component logic in JavaScript.
- `<style>...</style>` - component scoped styles. This is by default it is defined in CSS.

The *style* part of the components is left empty, since the application is using the UI elements and their design delivered in Vuetify, with no need for any kind of alteration.

The component logic is further structured as a JavaScript object with named properties defining its functionality such as:

- `data` - local component data and their default values.
- `computed` - derived or computed data, such as values derived or mapped from an application state variable.
- `watch` - callbacks triggered on local/computed data change.
- `methods` - invokable logic. Methods are accessible from outside as well, i.e. a parent component can call a child component method.

The exact syntax can be observed in the application source code.

3.2.2 Vuex mapping

The components generally reflect the state of the application and therefore most component variables are derived (*computed*) copies of state variables. As mentioned before, instead of defining and calling getters for every sought out variable, Vuex provides an interface an interface (`mapState` function) to easily map a state property onto a component property.[19]

Here is an example of `mapState` application in `Configurator.vue`:

```
import {mapState} from 'vuex'

/* ... */

computed: {

  ...mapState({
    enabled: state => state.player.enabled,
    paused: state => state.player.paused,
    t_max: state => state.player.t_max
  }),

  /* ... */

}
```

[19] https://vuex.vuejs.org/guide/state.html#the-mapstate-helper

This maps the three state variables `enabled`, `paused`, `t_max` onto computed variables with the same name. Furthermore, accessing the computed values is equivalent to reading the values directly from the state store.

Vuex also provides mapping interface for getters, mutations and actions in a form of `mapGetters`, `mapGetters`, `mapGetters` respectively. These are generally mapped onto components methods.

An example of mapping Vuex functions onto methods is in `Simulator.vue`:

```
methods: {

  /* ... */

  ...mapActions([
    'memento/calc',
    'memento/init',
    'player/enable',
    'player/reset'
  ]),

  /* ... */

}
```

This is an even more shorthanded way of mapping, since the developer does not indicate the name of the method the action is mapped onto. Instead, a method with the same name is automatically generated. For example a mapped `reset` actions from `player` module is called from within the component by typing:

```
this['player/reset']();
```

It is important to note that since the modules are namespaced, the namespace appears in the function name as observed above.

3.2.3 App.vue

App component is the base component of the application and implements the base layout of the application. The implemented layout is based on the existing Three Column wireframe offered by Vuetify.[20]

```
<v-main class="grey lighten-3">
  <v-container>
    <v-row>

      <v-col cols="12" sm="2">
        <v-sheet rounded="lg">
          <Configurator/>
```

[20]`https://github.com/vuetifyjs/vuetify/blob/master/packages/docs/src/examples/wireframes/three-column.vue`

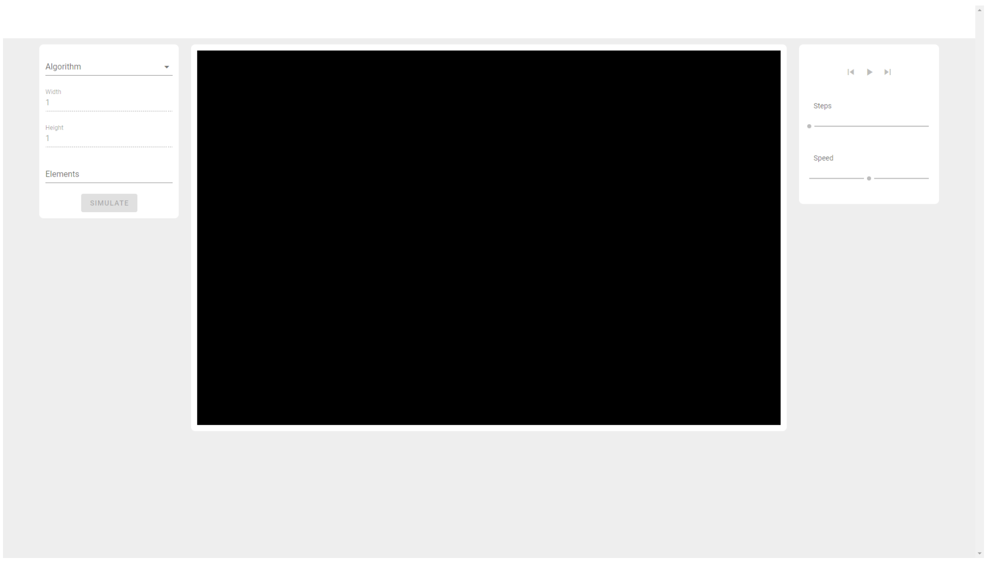


Figure 3.1: Application layout

```
        </v-sheet>
      </v-col>

      <v-col cols="12" sm="8">
        <v-sheet min-height="70vh" rounded="lg" class="d-flex">
          <Simulator ref="simulator"/>
        </v-sheet>
      </v-col>

      <v-col cols="12" sm="2">
        <v-sheet rounded="lg">
          <Player ref="player"/>
        </v-sheet>
      </v-col>

    </v-row>
  </v-container>
</v-main>
```

The template was slightly modified to take advantage of CSS Flexible Box Layout[21] in the central column (`class="d-flex"`). This allows the responsive elements inside the container to adjust to the screen size. In this case, the central cavas, where the simulation is visualized, adjusts to the full width of the central column. The application layout can be seen in Figure 3.1.

[21]`https://css-tricks.com/snippets/css/a-guide-to-flexbox/`

3.2.3.1 Application loop

The only logic defined in the base component is the application loop. Using JavaScript's `requestAnimationFrame` function, the operations in the loop execute 60 times per second, unless a web browser has a different refresh rate.[22] This corresponds with the usual animation framerate.

Every cycle, the *App* component instructs the *Simulator* component to update and rerender the graphics and *Player* component to progress the `player` state, i.e. progress the time of the simulation.

```
methods: {
  loop: function () {
    this.$refs.simulator.update();
    this.$refs.player.update();
    requestAnimationFrame(this.loop);
  }
},
```

The `mounted` property is an example of a lifecycle hook, provided by Vue. Using this property, a developer can specify what logic is executed once the component has been created and mounted, i.e. placed on the page. In this case, once the application is loaded, the aforementioned lopp is started.

```
mounted: function () {
  this.loop();
}
```

3.2.4 Configurator.vue

The `Configurator` component's purpose is providing a user with input fields and processes them. Some of the component logic is defined in the template itself by defining appropriate attributes in the input element tags.

The final design contains four Vuetify input fields, as seen in Figure 3.2, each bound to a local component variable.

- `algorithm` is a required input. The selection mirrors the generated list from the `config` module by mapping the `algorithms` getter onto a computed variable, which is then bound to the drop-down menu items list.

  A watcher is defined for `algorithm`, which when triggered checks the dimensions an algorithm can work in. If the algorithm works only on a one dimensional grid, it resets the value of `height` (defined below) to 1.

- `width` is a required input. The input is enabled only when an algorithm is selected and only allows positive integer values.

[22] `https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame`

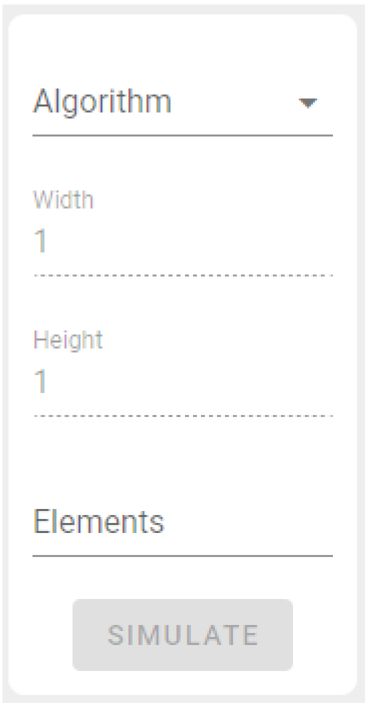


Figure 3.2: Configurator layout

- `height` is a required input. The input is enabled only when an algorithm, which works on two dimensional grids, is selected. It also allows only positive integer values.

- `elems` is not required and indicates how many elements there will be in a generated grid. The input is always enabled and only allows positive integer values.

  If no value is entered, it is assumed the number of elements is the same as the number of nodes.

Once all required inputs are entered and all input is validated, a confirmation ”Simulate” button is enabled, which updates the `config` state.

3.2.4.1 Form validation

Vuetify provides a shorthanded way of defining input validation through a `:rules` attribute.[23] It needs to be supplied a list of expressions that evaluate to `true` or an error message.

For example, `height` validation rules are as follows:

[23] `https://vuetifyjs.com/en/components/inputs/#rules`

```
rules: {

  /* ... */

  height: [
    v => !!v || 'Height is required',
    v => v > 0 || 'Height must be positive'
  ],

  /* ... */

}
```

It is not neccessary to check whether the input is a number, since the input is limited to numbers only due to its type. The rules are then supplied to the element in the template:

```
<v-text-field :disabled="!algorithm || algorithm.dimension < 2"
  v-model="height" :rules="rules.height" label="Height"
  type="number" min="1" required/>
```

The form itself is goes through a valid or invalid status, based on the validation of its individual inputs. This status is bound to a local variable, which affects whether the confirmation button is enabled or disabled.

3.2.5 Simulator.vue

The *Simulator* component acts as mediator between state modules and the display. The template only contains a container element, which contains a canvas where the simulation graphics is drawn. Once the component is created and mounted, it initializes the graphics object `Graphics3D`. A canvas element, on which the graphics is rendered, is then supplied back to be placed in the container element.

The component then updates the graphics and application states when triggered by watching for changes in mapped states variables:

- `t` and `delta` are mapped to be the integral and decimal parts of the `t` state variable in `player`.

  ```
  t: state => Math.floor(state.player.t),
  delta: state => state.player.t % 1
  ```

- `config` variable represents the `config` state.

  When a change is detected, it means that a new simulation configuration has been entered by a user, and following steps are taken:

1. The *Player* component is reset by resetting the `player` state module.
2. `memento` state module is reset and initialized with new configuration.
3. The calculation of all simulation steps is started by dispatching the `memento` state action.
4. The graphics object is initialized based on the new simulation grid. The initialization involves initializing graphics objects representing nodes, connections and elements and their starting positions, calculated from the element placement in the grid.
5. The *Player* component is re-enabled by updating the `player` state module.

- `currStep` variable represents the current simulation step, i.e. the $\texttt{t}^{th}$ simulation step. When no steps are calculated, `currStep` holds a `null` value.

  When a change is detected, `Graphics3D` object is told to update the positions of graphics objects, i.e. to update the position of elements. This of course also detects the change to and from the `null` value.

- `nextStep` variable presents the simulation step following the current one. When the simulation is on the last step, `nextStep` holds a `null` value.

  When a change is detected, `Graphics3D` object is told to update the color design of graphics objects, i.e. to highlight the nodes and connections based on the properties of the simulation step.

- `stepCount` variable holds the total number of simulation steps by referring to `size` of the `memento` state module. Every time this value changes, the steps slider range in the *Player* component is updated by updating the `t_max` value in `player` state module.

Furthermore, every application loop, the *App* component invokes update logic in the *Simulator*. This consists of, again, updating graphics objects positions based on the `delta` value in order to animate the element transition.

Finally, in order to keep the page responsive, the component has to react to window changes. Using the `v-resize` attribute, the container element in the template triggers custom logic. This consists of instructing `Graphics3D` to resize the render area to a newly calculated area, based on the size of the container.

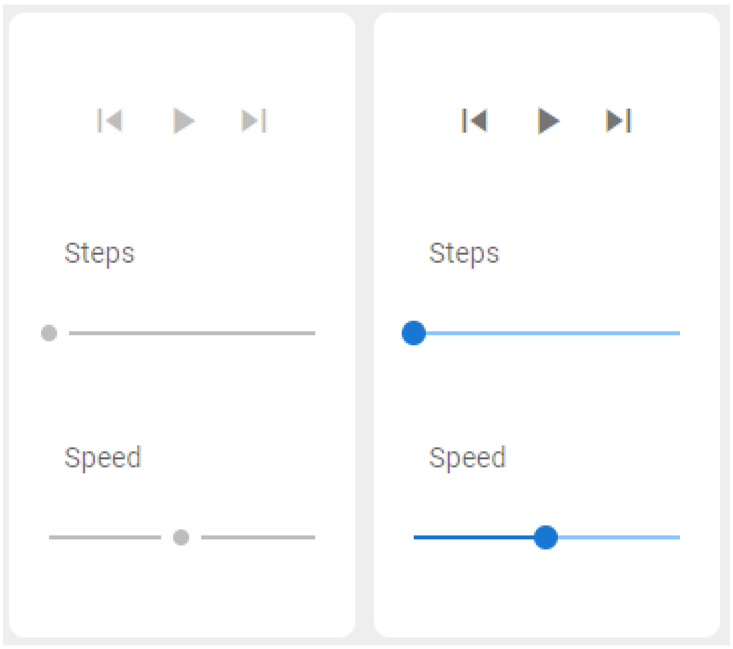


Figure 3.3: Player layout (disabled/enabled)

3.2.6 Player.vue

Similarly to the *Configurator* component, *Player* component provides an interface for user input. Contrary to *Configurator*, it does not have to validate it because input limited by the Vuetify form elements.

The values and states displayed directly reflect the `player` state module, through data mapping and binding. By default the component is disabled and this state reflects the `enabled` state of the `player` module. As described before, that is enabled when a simulation is configured and initialized.

The form inputs consist of (all mentioned actions and states are defined in the `player` state module):

- *Skip to previous* and Skip to next buttons trigger the `decrementTurn` and `incrementTurn` actions.

  All buttons, including the *Play/Pause* button, are displayed as icons from Material Design system, available through Vuetify.

- *Play/Pause* button display changes based on the `paused` state. Clicking it executes the `togglePause` action.

- *Steps* slider is bound to a local variable `t`, which is derived from the state variable of the same name.

The slider works as a simple navigation through the simulation steps. Apart from displaying a specific step, it allows navigating to both the start and the end of a simulation.

- *Speed* slider is bound to a local variable `speed`, which is derived from `delta` and allows switching between three different simulations speeds. As specified, the starting speed is set to ”2”.

Every application loop, `modifyByDelta` is dispatched in order to update the simulation time. This progresses the simulation based on the selected speed.

3.2.6.1 Two-way computed properties

From design, a component variable can not be directly bound to a state variable. This is because a state can only be updated through mutations. In order to reflect changes to a local variable to onto a state, it is possible to define logic triggered on value change.[24]

The following code defines computed variables `t` and `speed`, which are boung to the input sliders mentioned above.

```
t: {
  get() {
    return Math.floor(this.$store.state.player.t);
  },
  set(value) {
    this.$store.commit('player/setT', value);
  }
},
speed: {
  get() {
    return Math.round(this.$store.state.player.delta / BASE_DELTA
    );
  },
  set(value) {
    this.$store.commit('player/setDelta', value * BASE_DELTA);
  }
}
```

The simpler logic for `t` makes it always evaluate to its corresponding state rounded down. Any changes to the variable trigger the /texttsetT mutation and update the state.

`speed` is translated from and to `delta` state using `BASE_DELTA` constant. This is neccessary because `speed` represents an integer value between 1 and 3, while `delay` represents actual value of the change in simulation time.

`BASE_DELTA` is a constant defined within the component and is set to 0.005. This means that on default speed, one simulation step takes approximately

[24]`https://vuex.vuejs.org/guide/forms.html#two-way-computed-property`

1.4 seconds to animate, before the simulation starts animating the next step (assuming, a browser has a standard refresh rate of 60 frames per second). The value has been chosen based on my observation, outside of application specifications.

3.3 Utility classes

Apart from the Vuex states and Vue components, which conform to certain implementation models defined by their respective technologies, the implementation contains custom utility classes, placed in the `util` directory.

3.3.1 `Grid.js`

`Grid` class represents a one or two dimensional rectangular grid of connected nodes. As designed, it has four attributes representing width, height, number of elemens and their placement:

```
width = -1; // width
height = -1; // height
elems = -1; // number of elements
values = []; // values}
```

`values` is a three dimensional array, where first two indexes are coordinates of a node, which is represented by an array of numbers. These are individual elements uniquely indentified by integers from $[0; elems)$ interval.

Since the application works heavily with grids, deep cloning is implemented as a class method `cloneDeep`, in order to avoid the shortcomings of shallow copies. This is easily done by parsing a copied object to and from JSON:

```
this.values = JSON.parse(JSON.stringify(values));
```

Associated `Highlight` class is a very simple structure designed in previous chapters with no added logic. Similarly, `Step` class is a simple composite of a grid and highlights, with a custom `state` attribute, used for computation of future steps. For further reading, it is useful to note the `Step` constructor declaration (The `state` and highlights arguments are completely optional.):

```
constructor(grid, state = 0, highlights = [])
```

3.3.1.1 Randomized generation

`generateGrid` is a static method declared in the `Grid` class. Given a grid width, heigth and number of elements, it generates a new grid with randomized element placement.

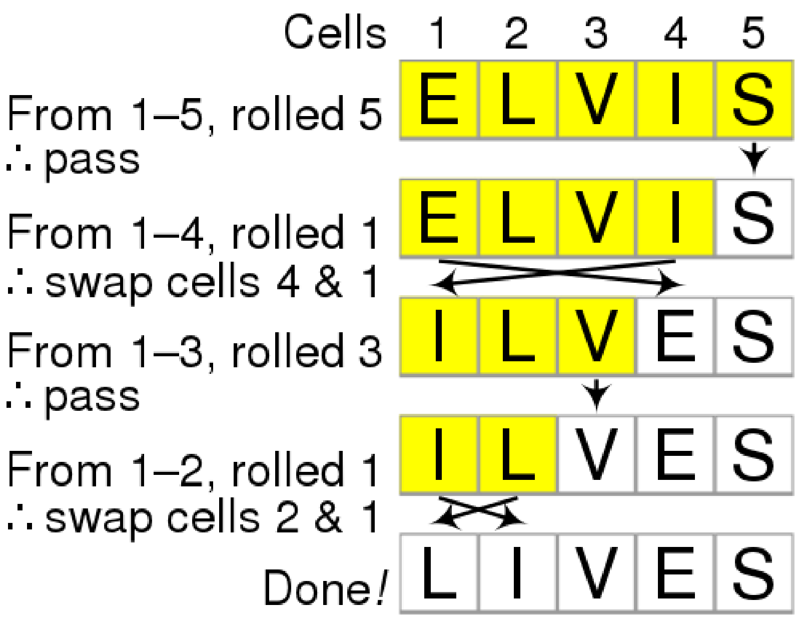


Figure 3.4: Durstenfeld shuffle (Wikipedia.org)

This is achieved by application of Fisher-Yates (a.k.a. Knuth) shuffle.[25] The modern version of this algorithm, called Durstenfeld shuffle, is an in place algorithm, which generates a random permutation of a finite sequence.

The algorithm iterates from highest index of an array to lowest. During each iteration contents of the current cell are swapped with a randomly chosen cell indexed not higher than the current cell. (The state can remain the same after one iteration.)

```
let vs = Array.from({length: elems}, (_, i) => i + 1);

// Fisher-Yates (aka Knuth) Shuffle
for (let idx = vs.length; idx > 0;) {
  let rnd = Math.floor(Math.random() * idx--);
  [vs[idx], vs[rnd]] = [vs[rnd], vs[idx]];
}
```

After the shuffle, the list of elements is evenly sliced into arrays, which are then placed in a grid. If the number of elements is not a multiple of number of nodes, the first r arrays contain one more element than the rest, where r is the remainder after division of the number of elements by the grid size $(= width \times height)$.

[25] https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle

3.3.2 Algorithms

Both currently implemented algorithms, `OddEvenSort` and `ShearSort`, are placed in the `algorithms` directory under `utils`.

The folder also contains an `index.js`, which groups all algorithms and exports them under one module:

```
import OddEvenSort from './OddEvenSort'
import ShearSort from './ShearSort'

export default {
  OddEvenSort,
  ShearSort
}
```

This makes it easy to import all algorithms in `config.js` and loop through them:

```
import algorithms from '@/utils/algorithms'

/* ... */

algorithms: () => {
  let items = [];
  for (let prop in algorithms) {
    items.push({
      text: algorithms[prop].name,
      value: algorithms[prop],
      disabled: false
    })
  }
  return items;
},
```

This means registering a new algorithm means placing the code in the `algorithms` directory and adding appropriate lines in `index.js`.

3.3.2.1 `BaseSort.js`

`BaseSort.js` is an abstract class, providing an interface for algorithm implementations. From design, it provides two static attributes `dimension` and `name`. There was an attempt to keep them immutable. Defining them as getters makes them syntactically accessible like attributes:

```
static get dimension() { return -1; }
static get name() { return "BaseSort"; }
```

Such getters could be overloaded the same way as other methods and could have been redefined for every child class. However, due to an incompatibility with Vuetify elements, this implementation was reversed.

As specified, `initStep` and `nextStep` methods are declared abstract. An implementation of `initStep` must return the initial simulation step of a simulation, given a grid. It is expected, that the function sets up the initial value of the custom `state` attribute, if neccessary.

`nextStep` function must accept a step as an argument and returns thte following simulation step, unless the given step is the last step of a simulation. In that case, it returns a `null` value.

Apart from the previous two methods, `BaseSort` provides a third, helper method `numeralCompare`.

```
static numeralCompare(a, b) {
  return a - b;
}
```

This static function handles simple numeral comparison. It is useful because the JavaScript built-in `sort` function compares numbers as strings by default, which has, in the case of this application, undesirable effects. Instead, it is possible to supply `numeralCompare` as a comparison callback to the `sort` function, e.g.:

```
grid.values[0][i].sort(BaseSort.numeralCompare);
```

3.3.2.2 `OddEvenSort.js`

`OddEvenSort` class is the simpler (of the two) example of an implentation of `BaseSort`.

Static class attributes are set as following:

```
static dimension = 1;
static name = "Odd Even Sort";
```

The initial step decided based on whether there are multiple elements in any nodes in the grid. (It's enough to only check the contents of the first grid, due to how the grid is generated.)

```
static initStep(grid) {
  if (grid.values[0][0].length > 1)
    return new Step(Grid.cloneDeep(grid), [0, 0]);
  else
    return new Step(Grid.cloneDeep(grid), [1, 0]);
}
```

The custom `state` is in a form of $[phase, turn]$. When there more elements in a node, both algorithms first sort the elements locally in each node. If every node contains only a single element or none, this phase can be skipped.

```
static nextStep(step) {

  let [grid, [phase, turn]] = [Grid.cloneDeep(step.grid),
    step.state];
  const w = grid.width;
  let highlights = [];

  switch (phase) {

    case 0:

      for (let i = 0; i < w; ++i) {
        grid.values[0][i].sort(BaseSort.numeralCompare);
        highlights.push(new Highlight([0, i], [0, i]));
      }

      return new Step(grid, [1, 0], highlights);

    case 1:

      for (let i = turn % 2; i + 1 < w; i += 2) {

        highlights.push(new Highlight([0, i], [0, i + 1]));

        let arr = grid.values[0][i].concat(
          grid.values[0][i + 1]);
        arr.sort(BaseSort.numeralCompare);
        grid.values[0][i] = arr.splice(0,
          Math.ceil(arr.length / 2));
        grid.values[0][i + 1] = arr;

      }

      if ((turn + 1) === w)
        return new Step(grid, [2, 0], highlights);
      else
        return new Step(grid, [1, turn + 1], highlights);

    case 2:
    default:

      return null;

  }

}
```

turn indicates a number of iterations of the second phase. The phase only has to run w number of times, where w is the width of the grid, after which the data is guaranteed to be sorted.

Comparison and exchange on elements between nodes is simply implemented by grouping all elements into one array, sorting it, and splitting it

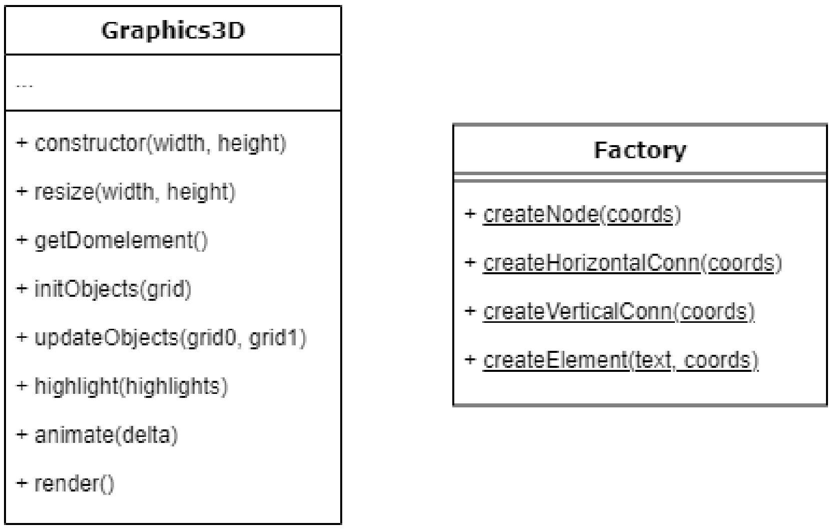


Figure 3.5: `Graphics3D` and `Factory` classes

halfway. From the code, it is obvious why `numeralCompare` helps simplify the call to the `sort` function.

3.3.2.3 ShearSort.js

The implementation of the `ShearSort` class is very simillar the implementation of `OddEvenSort`, since one is analogical to the other.

The `state` is implemented as $[phase, iter, turn]$. The first optional phase sorts the elements locally. After that, the algorithm alternates between two phases, either sorting rows or columns.

iter represents how many times has the algorithm completed both phases. Once *iter* is equal to $min(w, h)$, where w is the width of the grid and h is the height of the grid, the simulation ends, i.e. the `nextStep` does not return any more steps.

turn again indicates the number of iterations of a phase and is limited by w when sorting rows and by h when sorting columns.

3.4 Graphics3D

The most complex class of the development is `Graphics3D`, located in `utils/graphics` directory. Its interface is shown in a partial UML diagram in Figure 3.5. `Factory` class is a graphics objects factory and helps separating display and objects creation logic.

When an instance of `Graphics3D` is created, it creates an empty scene, a camera object and a renderer object from the Three.js library. Part of the

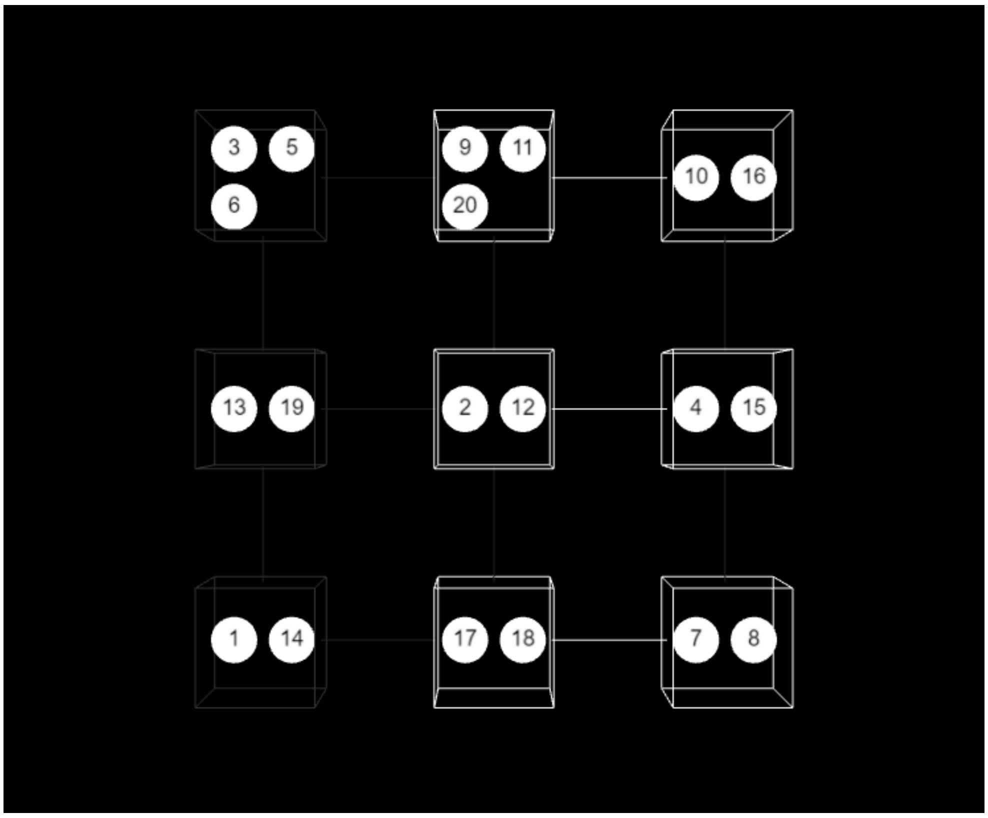


Figure 3.6: Grid design

renderer object is a canvas element, on which the graphics is being rendered, that can be retrieved via `getDomElement` method.

```
getDomElement() {
  return this.renderer.domElement;
}
```

This element is retrieved and registered on the page in the *Simulator* container.

3.4.1 Object design

A typical Three.js graphical object consists of a *geometry* (skeleton) and a *material* (texture). The `Factory` offers creation of objects representing all parts of a grid:

- A node is represented as line segments constructing a $1.0 \times 1.0 \times 0.5$ box. The default color of the lines is a shade of gray (hex: `0x333333`).

```
let geometry = new Three.BoxGeometry(1.0, 1.0, 0.5);
let edges = new Three.EdgesGeometry(geometry);
let node = new Three.LineSegments(edges, new Three.
    LineBasicMaterial({color: DEFAULT_NODE_COLOR}));
```

- Connections are represented as lines of length 1.0 connecting the center points of two closest box sides, of two neighbouring nodes. The default color of the line is a slightly darker shade of gray (hex: `0x222222`) to somewhat distinguish between the two objects.

```
let geometry = new Three.BufferGeometry().setFromPoints([
  new Three.Vector3(-0.5, 0, 0),
  new Three.Vector3(0.5, 0, 0)
]);
let conn = new Three.LineSegments(geometry, new Three.
    LineBasicMaterial({color: DEFAULT_CONN_COLOR}));
```

- Elements are more complex. In order to display text on an object representing an element, it is first drawn on a separate canvas, which is then applied as a texture to the element object. The shape of the object is a two dimensional circle of radius 0.5.

```
let canvas = document.createElement("canvas");
[canvas.width, canvas.height] = [40, 40];
let context = canvas.getContext("2d");

context.font = "14pt Arial";

context.fillStyle = "white";
context.fillRect(0, 0, canvas.width, canvas.height);

context.textAlign = "center";
context.textBaseline = "middle";
context.fillStyle = "black";
context.fillText(text, canvas.width / 2, canvas.height / 2);

let texture = new Three.Texture(canvas);
texture.needsUpdate = true;
let material = new Three.MeshBasicMaterial({map: texture});
let elem = new Three.Mesh(new Three.CircleGeometry(0.5, 60),
     material);
```

When a connection or a node is highlighted in `highlight` method, their respective line segment colors are changed to white. The color design of the elements remains unchanged during the entire simulation.

The size of the rendered objects is determined by the size of their geometry multiplied by the scale property.[26] For easier computations, all geometry sizes are simply set to 1.0 and scaled later.

[26]`https://threejs.org/docs/#api/en/core/Object3D.scale`

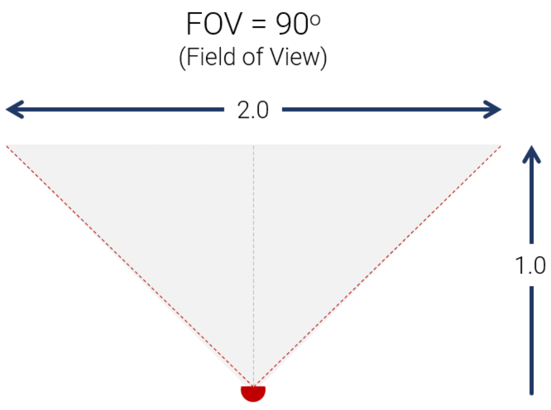


Figure 3.7: Field of View (VideoConferenceGear.com)

3.4.2 Object positions

To avoid complicated calculation when placing objects in the space, the coordinates of node $n_{i,j}$ (node in row i and column j of a grid, indexed from 0) are set to $[2 \cdot i \cdot s, -2 \cdot j \cdot s]$, where s is the width/height of the box representing the node. z-coordinate is left at default value, which is 0.

The creation and placement of objects occurs during their initialization when `initObjects` is called. The position of other objects is derived from the position of the nodes.

3.4.2.1 Camera

After node initialization, camera is repositioned to view and display the whole grid. The x, y coordinates of the camera are mid point between the positions of the the top left and bottom right nodes.

```
this.camera.position.x = (this.nodePos[h - 1][w - 1][0] +
  this.nodePos[0][0][0]) / 2;
this.camera.position.y = (this.nodePos[h - 1][w - 1][1] +
  this.nodePos[0][0][1]) / 2;
```

Alternatively, because of how node coordinates are calculated, the following equation also holds:

$$[x_{camera}, y_{camera}] = [(width - 1) \times s, (1 - height) \times s] \tag{3.1}$$

During `Graphics3D` initialization, the field of view of the camera object is set to 90°. This means that the distsance of the camera from the grid has to be half the total width or height (the larger of the two).

Depth of a node's box has to be accounted for in calculating the z-coordinate of the camera. Also some distance has to be added in order to display some empty area surrounding the grid:

$$z_{camera} = \frac{z_{node}}{2} + max(width_{grid}, height_{grid}) + s \tag{3.2}$$

This is simply implemented as:

```
this.camera.position.z = s * 5 / 4 + Math.max(w, h);
```

3.4.2.2 Connections

Connection lines are divided into horizontal and vertical. For a horizontal connection $c_{i,j}$, connecting nodes $n_{i,j}$ and $n_{i+1,j}$, its position is calculated as a shift in the direction of the x-axis from $n_{i,j}$:

$$pos_{c_{i,j}} = pos_{n_{i,j}} + (s, 0, 0) \tag{3.3}$$

Similarly for vertical connections, the shift from an associated node is in the direction of the y-axis:

$$pos_{c_{i,j}} = pos_{n_{i,j}} - (0, s, 0) \tag{3.4}$$

3.4.2.3 Elements

Element position calculation is a more complex operation, since a node can contain multiple elements. The way elements are organized can be seen in Figure 3.6.

They form a rectangular grid inside a node, whose width (number of columns) is either equal to its height (number of rows) or greater by only 1. Furthermore, the center of the element grid corresponds with the center of the node.

The number of columns of such a grid in node n is:

$$w_n = \lceil \sqrt{|n|} \rceil, \tag{3.5}$$

where $|n|$ is number of elements contained in node n.

The number of rows is:

$$h_n = \begin{cases} w_n & \text{if } |n| > w_n \times (w_n - 1) \\ w_n - 1 & \text{otherwise.} \end{cases} \tag{3.6}$$

The distance between elements is:

$$d_e = \frac{s_n}{w_n} \tag{3.7}$$

Finally, with the variables above it is possible to calculate the coordinates of the individual elements in a node. The x-coordinate of a the k-th element (in left to right, top to bottom order) in a node is:

$$x_{e_k} = x_n + ((k \bmod w_n) - \frac{1}{2} \cdot (w_n - 1)) \cdot d_e \tag{3.8}$$

and its y-coordinate is:

$$y_{e_k} = y_n + (h_n - \lfloor \frac{k}{w_n} \rfloor - \frac{1}{2} \cdot (h_n + 1)) \cdot d_e \tag{3.9}$$

This calculation occurs not only during the initialization of elements, but also every time the objects are updated through the call of `updateObjects`. The positions of elements is computed and remembered for both the current and next simulation step.

3.4.3 Object size

During the lifetime of the `Graphics3D` object, the size of nodes and connections does not change. The size of the elements however scales to visually fit inside a node, together with other elements.

The size of an element in a node n is defined as:

$$s_e = \begin{cases} 0.8 \cdot \frac{s_n}{\lceil \sqrt{|n|} \rceil} & \text{if } |n| > 1 \\ 0.5 \cdot s_n & \text{otherwise.} \end{cases} \tag{3.10}$$

When a node contains only a single element, it is more visually pleasing when the element does not fill up the node, especially when a node consists of a thin lined box skeleton.

In contrast, when a node contains more elements, making them too small would make the numbers difficult to read. Their size is derived from the division of the size of the node (s_n) by the number of elements on one row. This is then scaled down to 80%, in order to keep some space between the edges of the elements.

Like their positions, element size is computed and remembered for both the current and next simulation step every time `updateObjects` is called.

3.4.4 Animation

In every iteration of the applicaton loop, the *Simulator* component calls the methods `animate` and `render`, in this order. The latter simply instructs the renderer object to render the updated scene.

The `animate` method accepts `delta` / Δ as an argument and updates the positions and sizes of the elements before they are rendered. In order to add a delay to the animation, Δ, which has a value from the closed $[0, 1]$ interval,

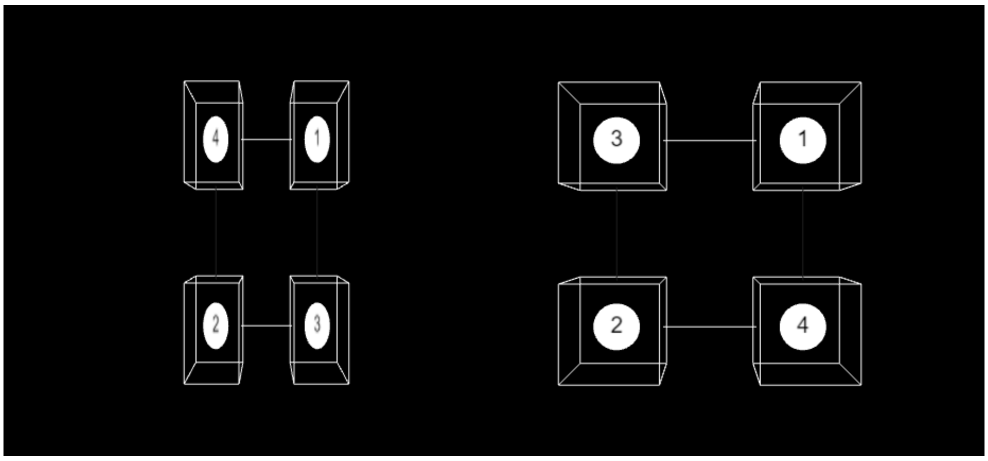


Figure 3.8: Responsive graphics

is transformed into Δ', a value in an interval $[-0.5, 1.5]$, using the following formula:

$$\Delta' = 2 \cdot \Delta - \frac{1}{2} \tag{3.11}$$

The imediate position of an element is then defined as:

$$pos = \begin{cases} pos_{curr} & \Delta' \leq 0 \\ \Delta' \cdot (pos_{next} - pos_{curr}) & 0 < \Delta' < 1 \\ pos_{next} & \Delta' \geq 1 \end{cases} \tag{3.12}$$

Similarly, the size of an element is defined almost identically:

$$size = \begin{cases} size_{curr} & \Delta' \leq 0 \\ \Delta' \cdot (size_{next} - size_{curr}) & 0 < \Delta' < 1 \\ size_{next} & \Delta' \geq 1 \end{cases} \tag{3.13}$$

3.4.5 Canvas resizing

When the *Simulator* component containing the canvas gets resized, it triggers a call to `Graphics3D`'s `resize` method. This does not only change the canvas size, to fit the new page layout, but it also must update the camera's aspect ratio and viewing frustrum, which was set during creation.[27]

Figure 3.8 shows the effect of resizing the canvas without and with updating the camera object as well.

The proper implementation is done using Three.js built-in methods:

[27]`https://discoverthreejs.com/book/first-steps/responsive-design/`

```
resize(canvasWidth, canvasHeight) {
  [this.canvasWidth, this.canvasHeight] = [canvasWidth,
    canvasHeight];
  this.camera.aspect = this.canvasWidth / this.canvasHeight;
  this.camera.updateProjectionMatrix();
  this.renderer.setSize(this.canvasWidth, this.canvasHeight);
  this.renderer.render(this.scene, this.camera);
}
```

Chapter 4

Testing

Conclusion

Bibliography

Appendix **A**

Acronyms

API	Application Programming Interface
CSS	Cascading Style Sheets
GPU	Graphical
GUI	Graphical User Interface
HTML	HyperText Markup Language
MVC	Model-View-Controller
SPA	Single-page applcation
UML	Unified Modeling Language

Appendix **B**

Code structure

`readme.txt`	the file with CD contents description
`exe`	the directory with executables
`src`	the directory of source codes
`src/wbdcm`	implementation sources
`src/thesis`	the directory of LaTeX source codes of the thesis
`text`	the thesis text directory
`text/thesis.pdf`	the thesis text in PDF format
`text/thesis.ps`	the thesis text in PS format