Project Report for Google Maps for Models

Summer Term 2021

Bennet Bleßmann, Mika Pöhls, Felix Jöhnk

Master Project
September 2021

Prof. Dr. Reinhard von Hanxleden
Real-Time and Embedded Systems Group
Department of Computer Science
Kiel University

Advised by
M.Sc. Niklas Rentz

Abstract

The topic of graph drawing is an old but still relevant topic in computer science. There are many applications for automatically generated graphs. To name two, one could use it as a graphical language to "write" programs or use it in an IDE to illustrate a program's code. Therefore, a major research topic deals with the automatic generation of graphs based on source code. In this work we represent changes made to the open source KLighD project initially developed at Kiel University as well as improvements to the back-end of the display tool used at the time of writing. The goal was to improve the overall performance of the *smartzoom* feature, which improves the graphical representation when zooming in and out in larger diagrams. This feature implemented by Wolff [3] was too slow to be used efficiently, but improved the overall readability of the graphs and was inspired by tools like Google Maps. To obtain the improvements we desired, we reworked how the model is translated into the representation of the graph and subgraphs. We also improved the solution provided by Wolff by adding some features that should further improve the readability and the overall usability. This includes title visibility of smaller subgraphs, bookmarks and changing the decision when a subgraph is fully rendered.

Contents

List of Figures

Chapter 1

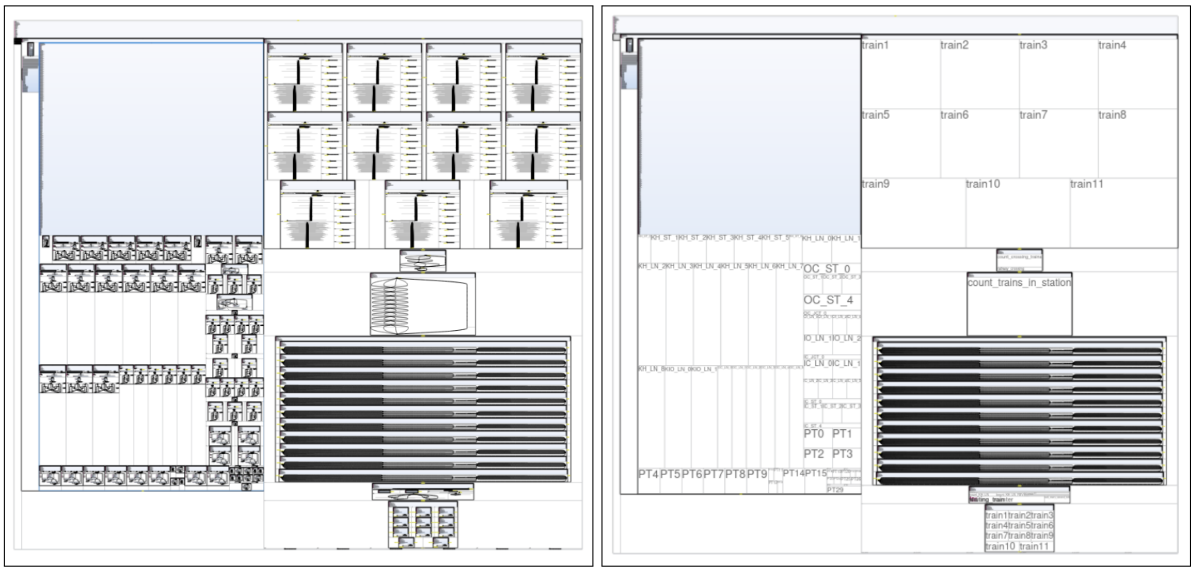
Introduction

With ever larger and evermore interconnected projects, it gets harder and harder to keep an overview using textual representations alone. Manually creating visual representations at that scale in a consistent manner and with constant changes is a practically impossible task. One suitable option left is to generate visuals from the textual representation by an automated process. For a graph with deep nesting elements, even with a reasonable layout, labels in the lower regions can get quite small. This level of detail may overwhelm the reader and small elements that are not readable might just be unnecessary clutter. To remedy this situation, information should not be shown at an unreasonable zoom level, similar to how mapping tools such as Google Maps do not show all shops in a city while zoomed in to a nation state scale. This can be applied in general to any form of text or more specific to whole regions of a diagram. The introduction of these principles to the display of diagrams was the main focus of the work by Wolff [3] explained in the following section.

1.1 Previous Work

The master project of Wolff [3] gave a good start in the topic to reduce the complexity of graphs. He implemented a feature which allowed hiding too small subgraphs. Therefore, the readability of larger graphs was drastically improved. The key idea was to only display subgraphs that are large enough, and not those that cannot provide valuable information due to being too small and instead make a white box with the regions title. A simple example for a graph that provides no information is given in Figure 1.1a. All the small subgraphs are not readable and therefore only give the information that there is a graph or give some hint to the structure. This information can be enough for a designer of the model because they may know the number of subgraphs in some situations. For example, they may know that there are eleven trains, which belong to the regions in the upper right part of the diagrams, but for a person new to the project this graph does not provide a lot of information. The implementation for the *smartzoom* feature feature checks the width and height of graphs after they are rendered and once any of these two values would be large enough the graph would be displayed.

Another improvement was that labels and texts were too small when viewing larger graphs, as towards the top left of Figure 1.1a. This resulted in text that was unreadable and sometimes it appeared that the text would disappear when zooming out. Additionally the rendering time needed for the text was larger than necessary because the text would be unreadable till the user would zoom in, offering some way to enhance the performance. The solution was to check for the text size and only display it when the size would surpass a certain threshold and if the size was too small to display it as a block in the color of the text. This block would show that there is text and if one wants to read it one would need to zoom in. When comparing the text of the resulting graphs in the upper left blue box the labels look almost identical with the version of David Wolff being darker. This results in the improved graph as shown in Figure 1.1b.



(a) Previous version **(b)** David Wolff's version

Figure 1.1. Comparison between David Wolff's version on the right and the previous version on the left.

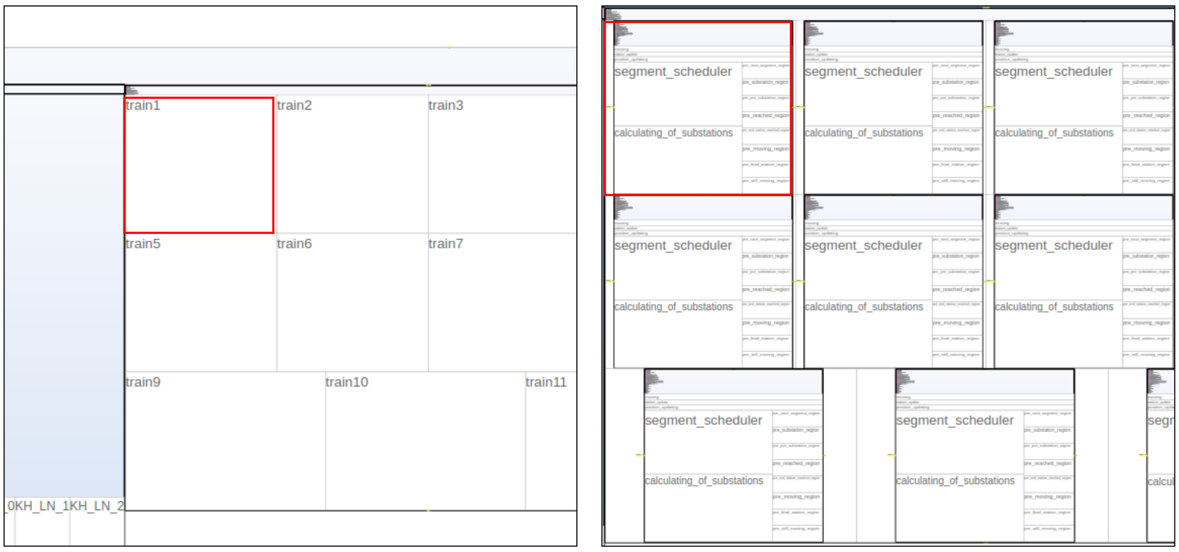
The other goal which was addressed by this project was to improve the performance since only parts of the graph should be rendered and thus decrease the time needed for the rendering.

1.2 Motivation

While the changes made during the last project improved the general visibility and readability of graphs as seen in the previous chapter, they were implemented in a way that the display of large diagrams suffered from serious performance issues. This resulted in a bad user experience, because the zoom and pan actions would execute with a noticeable delay, leaving the user waiting and making precise movement nearly impossible. The first and biggest goal of this project was to regain a rather smooth movement by improving the current implementation and therefore the performance.

An additional decrease in the user experience can be seen when displaying very big graphs with many subgraphs. To properly read different parts of the graph, the user needs to zoom in, zoom back out, pan to the position of the new subgraph, and zoom in again. If these switches between subgraphs are frequent, this procedure can become very annoying and time-consuming. Therefore, we wanted to allow the user to switch between positions in a graph by the press of a button.

The *smartzoom* introduced by the previous project only displays the enlarged title of the region, if the contained parts would not be visible. As a consequence, when zooming back into a collapsed region, the title may switch suddenly from a big and easily readable size into its original and possibly not readable size. This is demonstrated with the region highlighted using a red border in Figure 1.2. In the left diagram the region titles are visible, but after zooming in an additional step, they seem to disappear on the right-hand side. Even though the titles are still at the same position, they are now too small to be read. For a better user experience, we wanted to display a readable title even if the region is no longer collapsed and allow users to keep track in which region they currently are and



(a) Regions collapsed **(b)** Regions expanded

Figure 1.2. Region titles with different zoom values

keeping their mental map of the diagram intact. The importance of preserving the user's mental map while analysing diagrams was discussed by Misue, Eades, Lai, and Sugiyama [2].

Another issue was the compatibility with other projects ongoing at the same time. One project wanted, among other things, to send the new model from the language server to the user step by step and no longer as one big packet, to reduce the big overhead of the transmission process and quickly display at least parts of the diagram. Another project built a VS Code extension to replace the outdated KIEL Environment integrated in Theia (KEITH) framework. We therefore attempt to make sure that the changes made by us and the preceding project were compatible with the new projects.

1.3 Outline

In Chapter 2 the important terms and concepts are explained. After building the necessary foundation, the improvements and new features of the project are presented in Chapter 3. To show the impact of these changes, the final performance of the project is displayed and compared to older versions in Chapter 4. To conclude this report, we introduce some ideas to further improve this project in the future in Chapter 5 and recapitulate our results in Chapter 6.

Chapter 2

Used Terminology

In this chapter, we will introduce some concepts and fundamental terms used throughout the work.

2.1 SModel

The SModel is the class used to represent the model of the current diagram. The SModel instance is organized in a tree structure and each of the vertices inherits from SModelElement. An SModelElement provides two fields for parent and children to induce the tree structure, as well as a type to convert it into a DOM element using the corresponding view.

2.2 Region

The regions used by David Wolff, not to confuse with regions used in SCCharts, separate the SModel into hierarchical groups at a less granular level. At this region level it is then determined with which level of detail a rendering object should be displayed. For this, we currently differentiate between `FullDetail`, `MinimalDetail` and `OutOfBounds`. An example for the different detail levels is depicted in Figure 2.1. The region `scheduler` is expanded and thus its detail level is `FullDetail`, but the region `train2` is minimized and therefore `MinimalDetail`. A region that is not in the Viewport is `OutOfBounds`. The child regions contained within a `MinimalDetail` region are not shown and either `OutOfBounds` or `MinimalDetail` depending on the prior state.

2.3 Childarea

The child area of a region is the area in which all the subregions are rendered. In Figure 2.2 the red area is the childarea and containes the subregions `A` and `B`.

2.4 DepthMap

The DepthMap handles the access and initialization of regions. It determines which region's visibility state needs to be re-computed. For this, the DepthMap caches the initialized regions and stores the last viewport and threshold values. It also takes care of correctly handling a change of the SModel by resetting its state. The recalculation of a regions detail level may occur on a viewport or threshold change. The threshold determines the ratio between a regions side length and the corresponding display area side length at which a region will change from `MinimalDetail` to `FullDetail`. Therefore, each region has its own threshold value which determines if it should be expanded or collapsed and

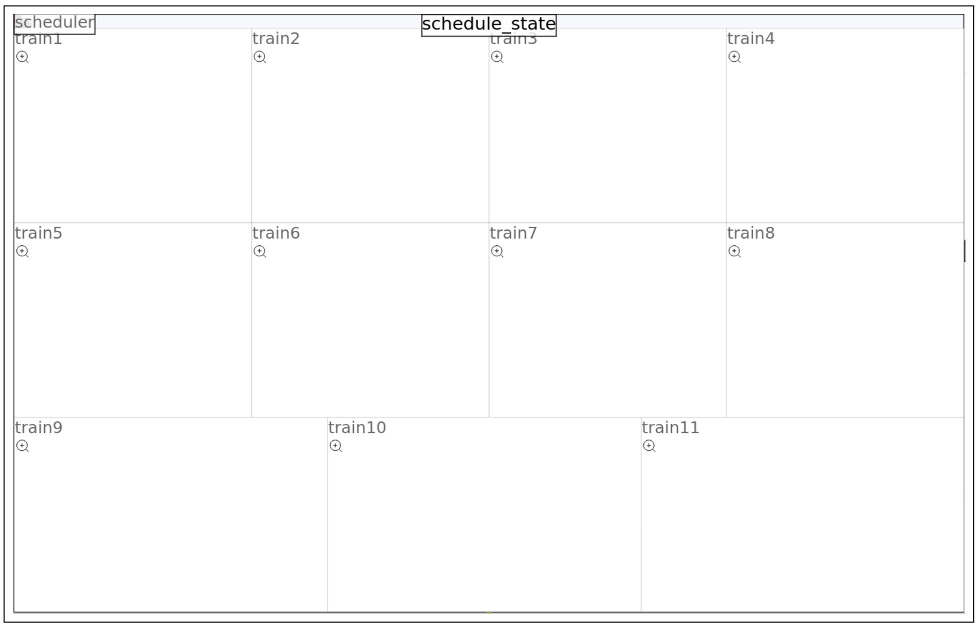


Figure 2.1. Depiction of different level of detail

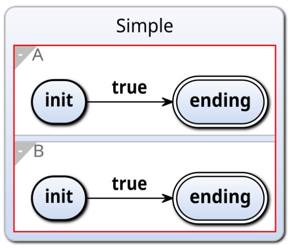


Figure 2.2. Childarea

gets updated on every zooming action. Both the width and height ratio is calculated and the smaller ratio is then compared to the configured threshold.

2.5 Viewport

The term viewport in general refers to the whole visible area of the diagram, while the viewport object usually corresponds to the root of the SModel. The viewport object defines the visible area by an offset defining the top left corner and a scaling factor, which are changed by panning and zooming, respectively. Additionally, the extent of the visible diagram is also influenced by the canvas size on which the Viewport is drawn. As a result the same offset and scaling values will have a different extent to the bottom and right side when the canvas is smaller or larger in these dimensions.

Chapter 3

Improvements

In this chapter, the actual changes are explained, starting with the general improvements of the previous project.

3.1 General Improvements

While getting accustomed to the code base, we noticed a couple of improvements that could be made to the DepthMap and related code to improve performance and memory usage.

One of them was the way the DepthMap was initialized, where originally the regions would be generated and then in another pass child and parent regions would be associated. As the parent regions were known at the time a new region was created, it was simple to associate them right there, rather than doing this in a separate pass.

Additionally, we noticed that the re-calculation of the region's visibility state was performed more often than necessary, as the function performing this calculation did this for most of the regions in one pass and the pass was performed for each node of the model. We changed the pass to perform this for all instead of most regions and moved the execution of the pass to be performed once per render rather than for each node per render.

3.2 Use SModel Layout Information

The DepthMap we inherited was initialized completely on first access and reinitialized when the model changed. This initialization was performed at the start of the rendering while accessing the DepthMap instance to add it to the rendering context. To reduce the impact on the responsiveness, the goal was to make the DepthMap initialization independent of the rendering. Most information to achieve this were already available in the layout information of the SModel. The main problem was that in the SGraph model a child node stored its offset relative to the parent's child area and not relative to the parent itself. The child area's offset to the parent was not easily accessible. The relative offset to the parent was required to compute the absolute position in the viewport which is used to determine whether a node is `OutOfBounds`.

To fix this, we adjusted the conversion to an SModel in the language server to incorporate the offset of the child area relative to the parent into the position of the child relative to the parent. Correspondingly, we no longer apply the child area offset during rendering as to not apply it twice. This made it possible to rewrite the initialization code to not require layout information provided by the browser DOM and instead relying on the layout information provided by the SModel.

3.3 Lazy Initialization of the DepthMap

The problem with the approach from Section 3.2 is that it still initializes the DepthMap eagerly for the whole diagram, even if only a fraction is required. Especially for large diagrams, this consumes unnecessary time and memory. Additionally, the master thesis from Kasperowski [1] deals with the transmission of the model piece by piece rather than in one chunk. This would mean that in the future the full model might not be available as a whole from the start and as such lazy initialization would be ideal.

For this we once again rewrote the (re-)initialization of the DepthMap to be as minimal as possible, it now only returns the DepthMap to an empty state. The generation of regions is now handed lazily an initialization function indirectly used through two accessor functions. One of the accessor functions returns, if present, the region corresponding to the given node, the other returns the region that contains a given node. The information for each node is cached by the initialization function in a map associating the node id with a record containing one or both region references, depending on the node. When the map contains such a record it is returned, otherwise a new record is computed, cached and returned. The corresponding field of the returned record is then returned by the accessor function. All accesses to DepthMap regions have been adjusted to go through one of the accessor functions and the old functions for accessing regions have been removed. Initialization is therefore again performed during the rendering though, due to the lazy behavior, imposes significantly less strain on the resources and now allows for the model to be provided incrementally.

3.4 Bookmarks

To make navigating large diagrams easier, we added a bookmark feature. The bookmark feature allows saving a position in the diagram as a bookmark, which enables to return to that position easily. The Figure 3.1 shows a diagram with the bookmark panel open, two saved bookmarks and the bottom bookmark title currently being in edit mode.

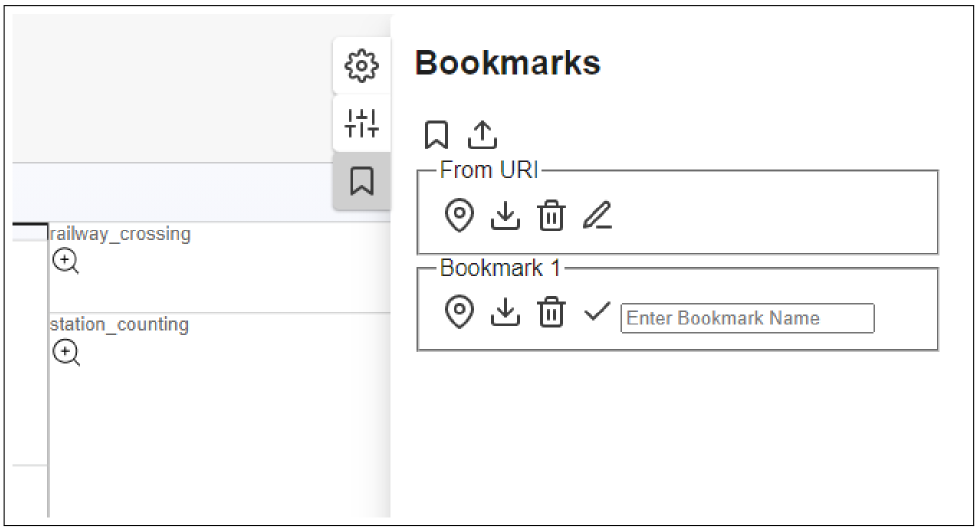
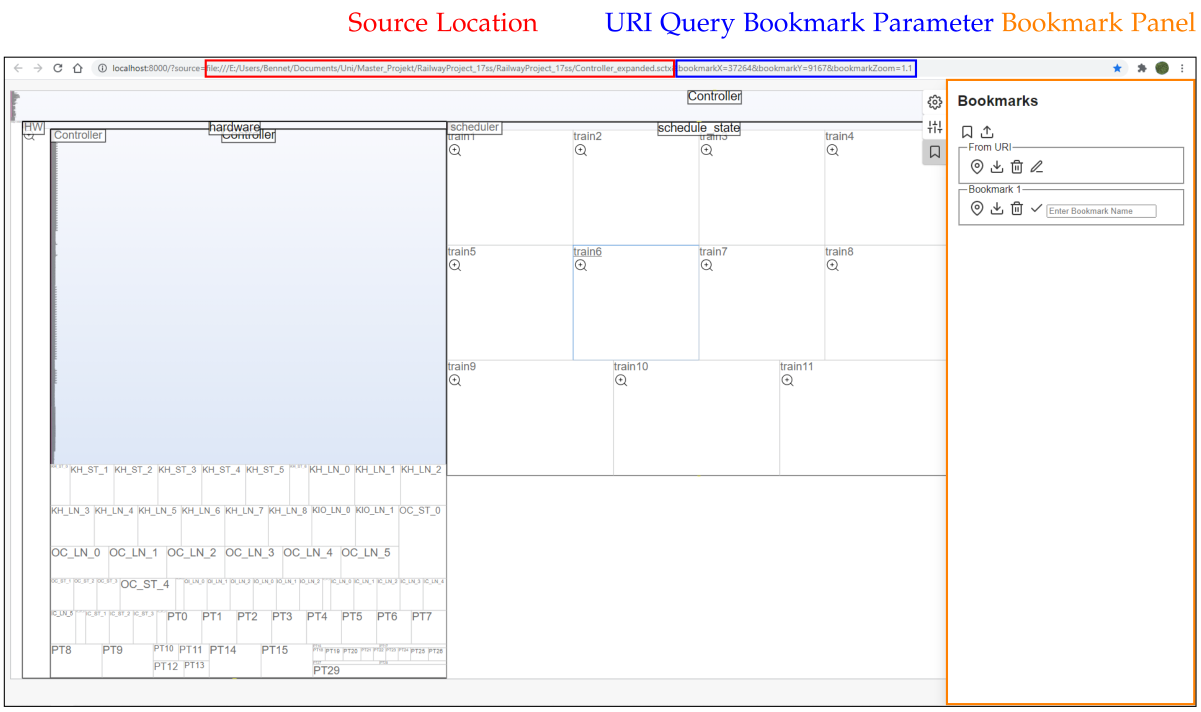


Figure 3.1. Cropped render of an SCChart with the bookmark panel visible and two bookmarks

The two icons at the top of the bookmark panel allow the user to create a new bookmark and load a bookmark from the clipboard respectively. The four icons on each bookmark allow you to go to the bookmarks position, save the bookmark to the clipboard, delete the bookmark and edit the bookmark title. The edit bookmark title button changes to a save title button with a text field next to it containing the current title, when clicked. When the save title button is clicked or enter is pressed while the text box has focus, the new bookmark title is saved and the edit bookmark title button is restored. The bookmark when saved to clipboard is a JSON-string containing the bookmark data.

When using the klighd-cli, it is possible to pass a bookmark using the URI as shown in Figure 3.2. This is possible with the use of the query parameters `bookmarkX`, `bookmarkY` and `bookmarkZoom`. The former parameters specify the position and the latter specifies the scale. The bookmark specified by the URI will be saved under the name "From URI" and the view will navigate to the bookmark once the diagram has completed loading.



localhost:8000/?source=file:///E:/Users/Bennet/Documents/Uni/Master_Projekt/RailwayProject_17ss/RailwayProject_17ss/Controller_expanded.sctx&bookmarkX=37264&bookmarkY=9167&bookmarkZoom=1.1

Figure 3.2. Render of an SCChart with the bookmark panel visible and two bookmarks

Implementationwise, the bookmarks are saved in the bookmark registry and each newly created and imported bookmark is assigned an index. The index gives each bookmark a non-volatile identifier, as bookmarks may otherwise be identical, e.g. when importing the same bookmark from the clipboard twice. The array index in the array backing the registry may change when bookmarks are deleted. This is especially important for unnamed bookmarks, as to differentiate them in the bookmark list. The index is part of the fallback name. Currently, the position saved in a bookmark refers to the top left corner of the viewport. This probably deviates from a user's expectation, as usually the viewport center would be the important part and not the top left corner.

3.5 Title Overlay

The previous project focused on enlarging the region titles only when the corresponding region was collapsed. First of all, the correct text elements for the titles had to be identified by checking if the `isNodeTitle` property was set by the language server. This property is set for all title texts, including regions, nodes, etc. So additionally it needs to be checked, if the parent of this text is a region to guarantee that this title is actually a region title. After that, the title can be dynamically scaled up to a user defined maximum factor.

Only scaling the titles if the region is collapsed can cause problems with the user's mental map and a difficult navigation through big diagrams, as discussed in Section 1.2. To address this issue, we applied the scaling independently from the expansion state of the region. To get the correct scaling factor, we calculate the maximum factor, at which the title will not exceed the region in either height or width. This guarantees that the title cannot overlap neighboring regions. The calculated scaling factor is used, if it is below the user defined maximum, otherwise this maximum scaling factor is used instead. This procedure is similar to the one of the previous project, therefore the scaling of titles for collapsed regions stays the same.

For expanded regions some additional steps are necessary. The first change is caused by the fact that the title can now overlap with other parts of it's corresponding region that became visible. As region titles are located in the top left corner in SCCharts, they would e.g. be drawn over variable declarations. To keep the title text visible and more importantly readable, a rectangle is placed behind the text with the same bounds as the text element. This background is slightly transparent, to indicate that there is text behind it. An example can be seen in Figure 3.3. In contrast to the same diagram in Figure 1.2, the title of the region `scheduler` is now visible in the upper left corner. Moreover, when comparing the regions `train9` and `train5`, it becomes clear that both titles can be easily read regardless of the different expansion state. The regions at the bottom, including sftrain9, are expanded, because they each have been given more space as there are only three of them. Without the title overlay feature, the title `train9` would be not readable.

Furthermore, the title should only overlap other parts of the region if it is necessary. This decision is based on the size of the text in the viewport. The exact threshold, at which the title should no longer overlay the region, can be set by the user in the options. To prevent a sudden decrease in the size of the title similar to the original issue, we use interpolation to create a smooth transition from the overlay title size to its actual size.

This fixes the visibility issue for region titles, but not yet for regular node titles. When applying the changes to these titles as well, they may be overlapped by other elements contained in the node due to their rendering order. To achieve a correct overlap in these cases, the title needs to be added to the diagram after any other element in this node. Instead of adding the title directly, it is saved in an array together with the transformations necessary for correct placement. The position in the array indicates the level at which the title needs to be retrieved and added. Therefore, after adding all other elements of the node to the diagram, the node title is added and can now be scaled while correctly overlapping other elements. This can also be observed in Figure 3.3 for `schedule_state` and the `station_scheduler` for the different trains.

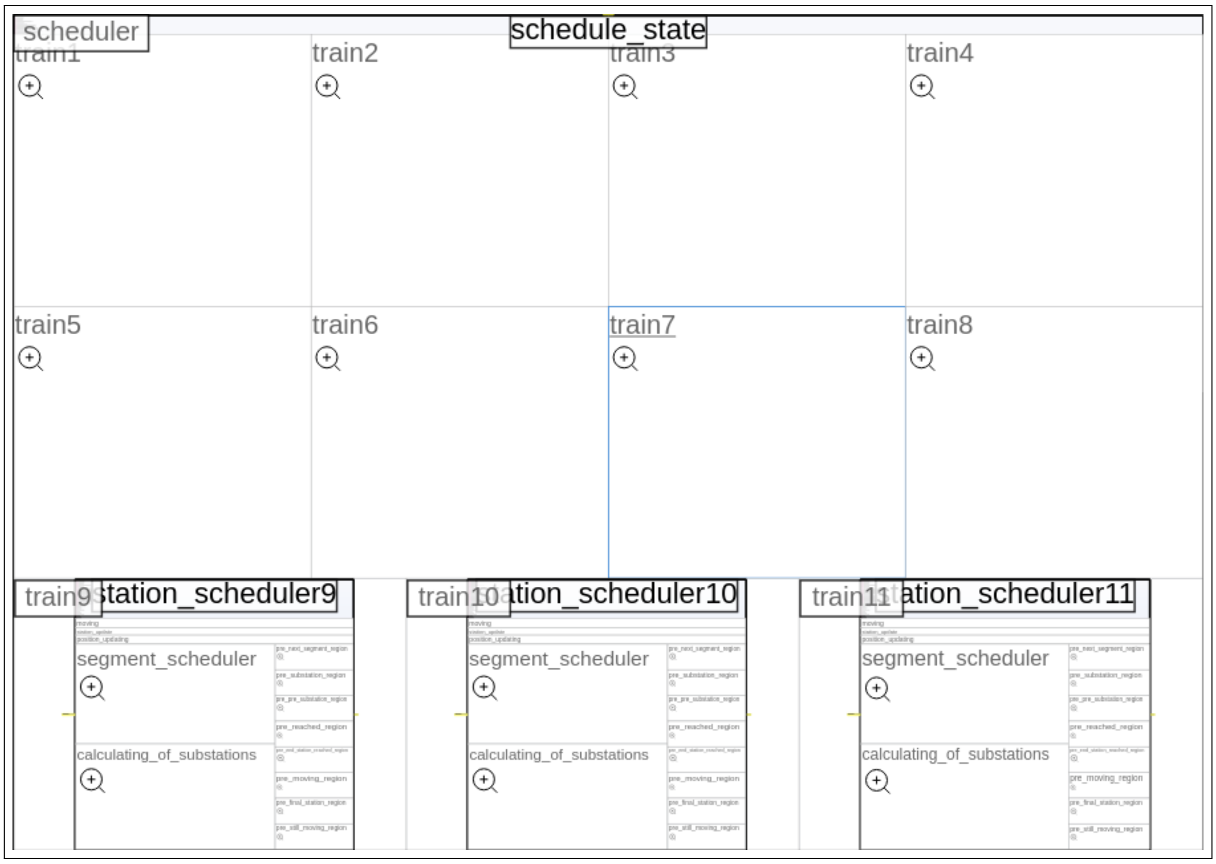
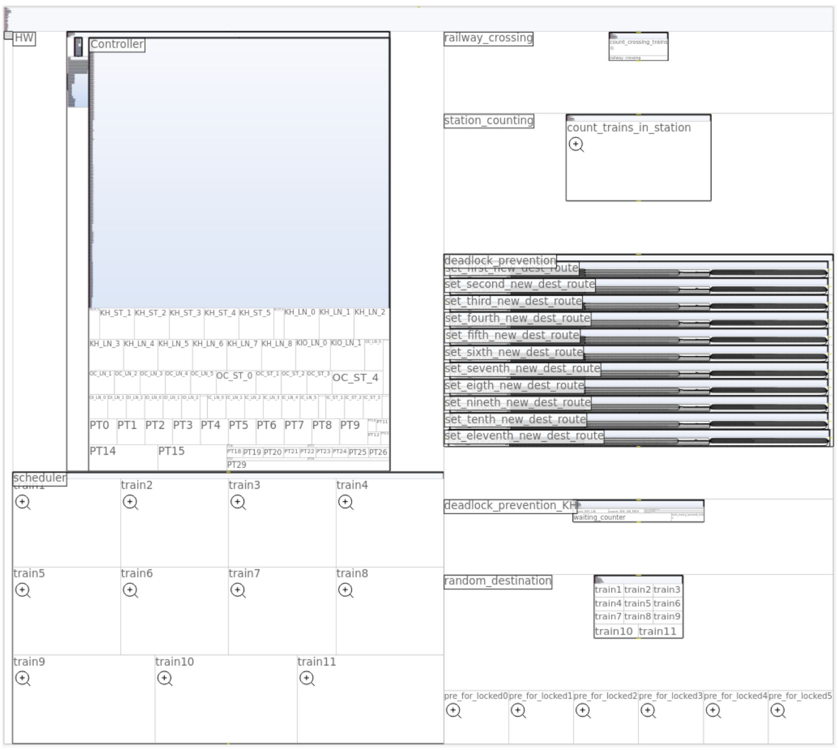


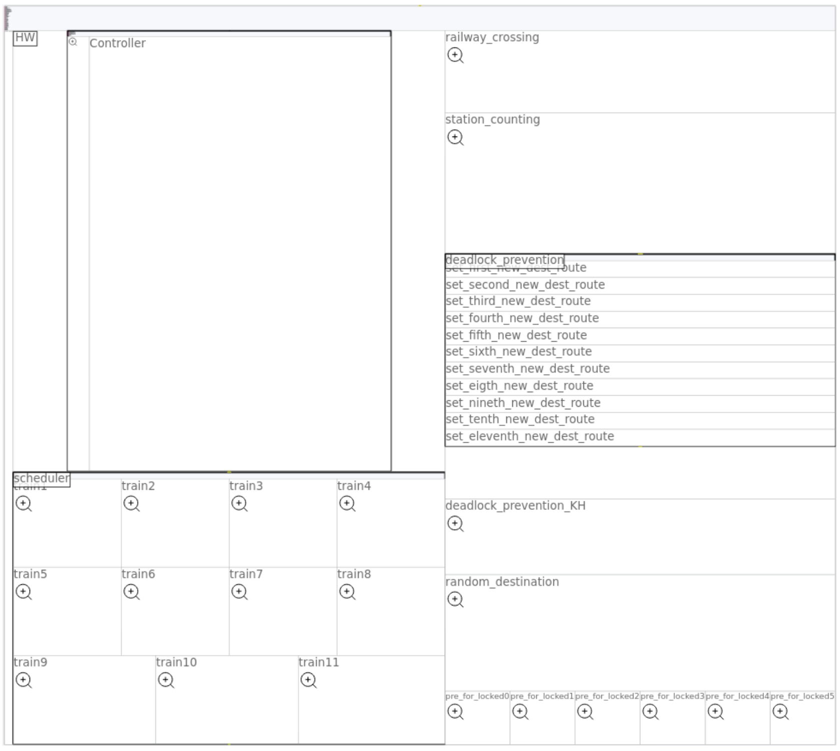
Figure 3.3. Region titles with overlay

3.6 Change of Visibility

Another minor problem we had when using the implementation from David Wolff was that he chose to display regions based on the maximum of width and height. This results in stretched graphs, as visible in the right of Figure 3.4a. This, however, is not appealing to the eye since it can result in large and mostly black blocks. The solution here was rather simple, we used the minimum of the region width and height to determine if the region should be displayed. This resulted in the improved graph Figure 3.4b. There one can directly see that the mostly black box in the top graph is not rendered but the regions with the corresponding titles give some information of the region's purpose. However, this change may also be bad because some regions may get minimized early. A example is also shown in the bottom right part in Figure 3.4a. When comparing it to the bottom right of Figure 3.4b we lose some information. Since both solutions have good and bad parts we leave more investigation to future work.



(a) Final implementation with minimum threshold



(b) Final implementation with minimum threshold

Figure 3.4. Comparison maximum threshold and minimum threshold

Chapter 4

Evaluation

In this chapter the improvements relevant to the performance of the project are evaluated. First we look at the general improvements explained in Section 3.1. To quantify the impact of this change, we ran both a version with and one without this change and manually performed a pan in each cardinal direction and one zoom in and out. Over the execution of these actions we measure the time spent in the `DepthMap.updateDetailLevels` function, as it is the entry point for the DepthMap's detail level recalculation. As a result the total time spent in the function as measurements using the browsers DevTools are presented in Table 4.1. The preferences and machine specifications used for testing are listed in Figure 4.2.

Due to the manual nature of this test, the pan distances and zoom amounts may differ between both runs. However, the performance of the `DepthMap.expandCollapse` functions clearly improved, given the quite notable reduction from 67.4% of the total execution time down to only 0.4% of the total execution time.

Table 4.1. Time spent in `DepthMap.expandCollapse` (renamed to `DepthMap.updateDetailLevels`)

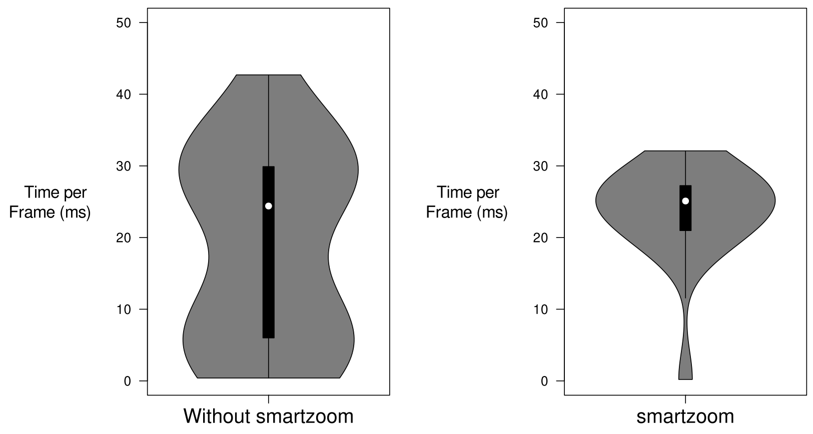
State	Commit	Time (%)	
Old	cdd29a27	15,776.6 ms	(67.4%)
New	3c72b4f8	55.9 ms	(0.4%)

Since the improvements of the changes from Section 3.1 were so dramatic, we are comparing the performance of our final implementation to a version where *smartzoom* was disabled completely.

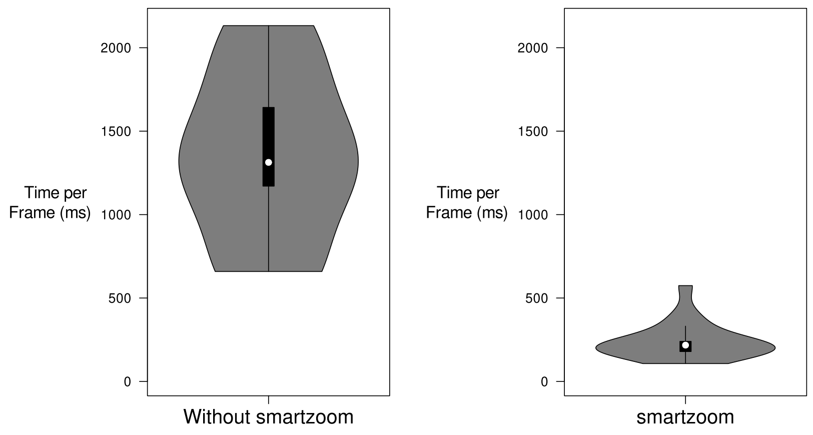
When using the DevTools to analyze the frame times of either version the pattern was mostly the same: a *dropped* frame time which was larger and a smaller frame time following it. Dropped frames can be caused by either a missed deadline, a script running too long and therefore blocking any new frames or because the browser wanting to speedup to recover latency.

The reason in our case is the fact that typescript is single threaded and thus while producing a new SVG of the model the browser cannot produce a new frame. For this reason we compare the *dropped frame* times and the *frame* times separately.

When comparing the *frame* times in Figure 4.1a the median is almost the same in both cases. This comes from the fact that in these cases a new SVG was produced by both backend instances and ready to be displayed by the browser. Since our implementation does not render all nodes, for example not those that are `OutOfBounds`, our final SVG to be displayed is smaller and therefore the time needed might be slightly smaller in the average case. However, these measurements could also be slightly imprecise and thus we cannot say that this is the case. In the end the *frame* times in both cases are small enough that they do not really contribute to the smoothness while using the tool.



(a) regular frames



(b) dropped frames

Figure 4.1. Performance of different project versions measured at the average time per frame in milliseconds. Nearly identical zoom actions were performed for each version and were obtained using the Google Chrome DevTools performance recording. In total we ran four tests for overall 40 frames recorded for both versions. Thus each graph includes 20 time values.

In contrast to the *frame* times the comparison of the *dropped frame* times in Figure 4.1b directly show the improvements. The *smartzoom* version is so fast that the slowest dropped frame is still faster than the fastest one from the version without *smartzoom*. In the average case a *dropped frame* takes up 1314ms in a version with *smartzoom* disabled, and only 239ms in the final version with *smartzoom* enabled, resulting in a speedup of 5.5. Comparing this to the result of David Wolff, who achieved a speedup of three in a smaller diagram, proves the worth of the contributions made in this project.

Setting/Spec	Value
Operating System	Kubuntu 21.04
Kernel-Version	5.11.0-31-generic
Operating System Kind	64-bit
Processor	20 × Intel® Core™ i9-10900 CPU @ 2.80GHz
RAM	31,2 GiB
VS Code	1.60.0
Google Chrome	93.0.4577.63
Diagram	Controller_expanded.sctx out of the railway project summer semester 2017
Render Options enabled (threshold)	Smart Zoom (0.2)
	Simplify Small Text (3)
	Title Scaling (1)
	Title Overlay (4)
	Constant Line Width (0.5)
Preferences	Resize To Fit
	Text Selects Diagram
	Animate GoTo Bookmark
Synthesis Options	Default

Figure 4.2. Benchmark Specifications

Chapter 5

Future Work

For bookmarks we currently save the top left corner of the viewport. When sharing bookmarks between different screen sizes this has the effect that not all relevant parts may be visible when changing to a smaller screen size, and when changing to a larger screen size the relevant part may only be a fraction of what is shown. To help with this problem, saving the viewport center may be a better position to save as this is usually the relevant part, and adjusting the extent of the visible diagram can then be adjusted by zooming in or out accordingly. Even better would be to not save the zoom level and instead save opposing corners of the viewport. This would allow for restoring the expected view more closely, only for changed aspect ratio one would require showing more of the diagram so that the whole original viewport is visible.

Additionally, it would be nice to use the persistent storage API to have the bookmark registry preserved between sessions.

Something else to tackle as depicted in Section 3.6 is a good heuristic to predict if a region should be expanded or collapsed. Right now the approach is to simply look at the width and height and if both values surpass a certain threshold it should be displayed. This, however, is quite bad for regions with very few nodes. In extreme cases the layout provided by the language server could be such that we have a slim but high region resulting in a late expansion due to being slim. If this region has only one node which is in the center of the region, one would need to scroll quite far to the center if the current position is at the top-left of the region where the title is. To tackle the problem, one could decide to display regions with fewer child nodes early. This however does not tackle the main problem of large empty areas in the graph and therefore a new layouting algorithm could be a good way of solving this issue.

The current version includes that upon using the *export as SVG* feature, the generated SVG displays the entire diagram without *smartzoom*, which means displaying the entire SModel. A better approach would be that the "exported as SVG" feature would ask the user to export the current viewport with the current *smartzoom* settings or the entire diagram. This would allow the user to send a specific cutout or part of the diagram which may help when investigating specific regions.

Another idea to improve the performance during the project was to only redraw the diagram if a region enters the viewport that was not rendered yet or was minimized previously. In the other cases, one could use the old rendered versions of nodes. The problem with this approach was that some lines and texts may get rendered differently on different zoom levels, therefore we discarded that option since the results we had already improved the performance to have good use ability.

The last idea for future work would be to have different diagrams based on the zoom level. This would require generating multiple diagrams in the language server and handling the transition from the diagrams one to another.

Chapter 6

Conclusion

The goal of the previous and current project was to improve the general user experience when displaying diagrams. While the previous project's main focus lied on the readability of large diagrams and especially on hiding too small elements in regions, our initial goal was to enhance the responsiveness of zooming and panning displayed diagrams. To achieve a better performance and reduce potential delay in these actions, the implementation done by the previous project was improved. As seen in Chapter 4 this led to a significant speedup in frame times, resulting in much less delay and a more pleasant user experience. Additional delays caused by excessive zooming and panning were addressed by merging consecutive actions into one before actually applying them. This prevents the build-up of these actions to some degree and allows a fast and at the same time smooth interaction with the diagram.

Apart from the lack of responsiveness, this project fixed two other smaller issues, to improve the user experience even further. When displaying expanded regions of large diagrams, the titles were often not or only hardly visible. By increasing their size and letting them overlap their corresponding region, the user is now able to keep track of the exact position in the diagram with all titles visible. The second feature added to the project was the bookmark feature. Bookmarks allow the user to save exact positions in the diagram to later go back to that same position. Especially in large diagrams this is very useful, as frequent switches to different positions would otherwise lead to annoying and repetitive zooming and panning.

Because another improvement was made to the same system by another project, an additional aspect of our project was to make the different changes compatible. These improvements changed the way of transmitting the SModel from sending it in one piece into sending it in several small pieces. To cope with the potential absence of the whole model, the DepthMap initialization was changed to work in a lazy manner, that means changes to the diagram will no longer lead to a whole new generation of the DepthMap, but rather to a change only of the affected regions.

Although there is still some future work to do, this project achieved several improvements not only to the performance, but the general user experience and the compatibility with other projects as well.

Bibliography

[1] Max Kasperowski. “A Top-Down Approach on Automatic Graph Visualization”. MA thesis. CAU Kiel, 2021. URL: `https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mka-mt.pdf`.

[2] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. “Layout adjustment and the mental map”. In: *Journal of Visual Languages & Computing* 6.2 (1995), pp. 183–210. ISSN: 1045-926X. DOI: `https://doi.org/10.1006/jvlc.1995.1010`. URL: `https://www.sciencedirect.com/science/article/pii/S1045926X85710105`.

[3] David Wolff. “Project Report for Google Maps for Models”. Tech. rep. CAU Kiel, Mar. 2021. URL: `https://rtsys.informatik.uni-kiel.de/confluence/pages/viewpage.action?pageId=75694276&preview=/75694276/94732327/dwo_mp_11pt.pdf`.