Enabling Git Truck to scale for use on large repositories

Thomas Hoffmann Kilbak (thhk@itu.dk), Kristoffer Bruun Højelse (krbh@itu.dk)

IT University of Copenhagen - December 15, 2023

***Abstract*—Visualizing the structure of, and how people collaborate on Git projects, was made easy with the tool Git Truck. User testing showed that Git Truck was useful for small projects, but that it could not scale to larger projects. In this paper, we present research and improvements that we have implemented in Git Truck to enable it to scale to be used on bigger projects. We do this by halving the analysis time by adding multi-threading to the analysis step and solving memory issues that cause crashes. We make the chart more responsive to interactions, by implementing it with WebGL and utilizing the browser's profiling tool to identify and eliminate unnecessary, expensive computations. As a result, we made it viable to use Git Truck on repositories as large as the Linux kernel.**

I. Introduction

Visualizing the file structure of Git projects, and getting an overview of where authors have contributed, was made easy with Git Truck[1]. However, the implementation has various scalability issues. Issues related to slow rendering hurt the user experience for large Git projects[1]. Issues related to the analysis of Git commit history make analysis of large projects very slow, and outright impossible for very large projects[2].

Previous work has shown that the method used to render in the browser, is highly impactful on the performance achieved[2][3]. A popular rendering method is WebGL, however, many libraries implement different functionality for different use cases. To take advantage of WebGL, we need to determine whether the existing features of Git Truck can also be implemented using one of the countless WebGL libraries available. Additionally, the analyzer of Git Truck has the potential to be made significantly faster, by utilizing multiple processor-threads.

In this paper, we will present the improvements made to the analyzer, the visualization, and other parts of the Git Truck system to make it usable on larger repositories. We will present and evaluate solutions for three primary problems that hold Git Truck back from scaling to work well on large projects: 1. Inability to display and animate projects with many visible components (e.g. 1000+ files) 2. Inability to analyze big projects (e.g. 1 million+ commits). 3. Excessive time to analyze large projects (currently 3 minutes for 150k commits).

To evaluate the improvements, we will use the Linux repository (86,000 files, 1.2 million commits) as a case study to test how usable Git Truck is on a large project. This includes benchmarking the rendering performance in frames per second (FPS), and the analysis time using various numbers of threads.

[1]Quotes from Git Truck survey: *"It was very slow and buggy (probably due to a lot of data - analyzed ApacheAirflow)"*, *"Easy to install and run, but slow"*, *"The bubble zooming is way too slow!"*

[2]Causes the error described in this issue: https://github.com/git-truck/git-truck/issues/576

II. Analyzer improvements

A. Memory utilization

When Git Truck retrieves the output from the Git process, it will get the entire output as a buffer, and make it into a string to run regex on it. This causes a problem when the output from Git is large, as Node.js strings can be at most 512 MB.[5] Thus, if the output exceeds this size, then Git Truck will crash. Exactly when this problem occurs, depends on commit message lengths, author name lengths, etc., however it usually becomes a problem when analyzing projects with more than 300k commits. We fixed this by only analyzing a subset of the total commits at a time and then analyzing repeatedly until all commits have been analyzed. This way, the string size does not exceed the limit. Figure 1 shows a flowchart of the commit analysis after all of the improvements mentioned in this paper have been implemented. The node "More commits?" loops such that each Git log output does not become too big.

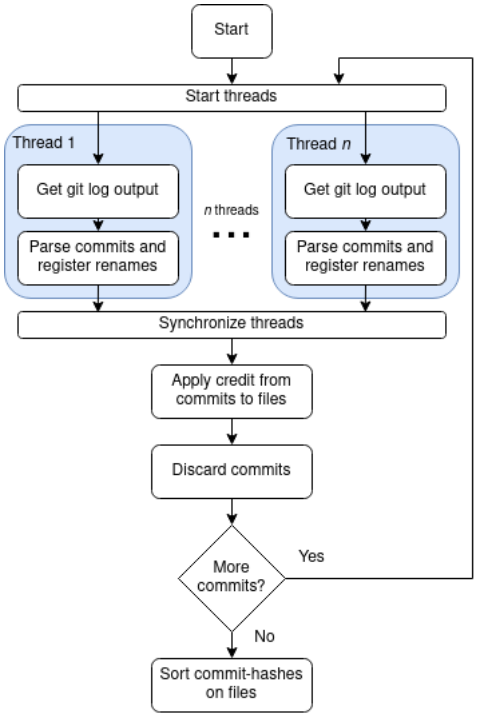

Figure 1: Flowchart showing the new structure of the hydrateData function. Multi-threading is achieved by spawning multiple processes to get Git log output.

Following this fix, we discovered another memory issue, that occurred when the maximum memory of Node.js was reached. By default, this value is 2 GB.[6] This was resolved by discarding commits after their credits had been added to the file ("Discard commits"-node in figure 1 shows this). This means that we do not need to keep all commits in memory during the entire analysis, which reduces the peak memory consumption. This has the downside, that we do not have access to the per-file commit history when analysis is complete, so we implemented on-demand loading of commits. Now, during analysis, only the hashes of commits that apply to a file are saved, along with that commit's timestamp. When analysis finishes we sort the commits, such that each file has a chronologically ordered list of the hashes of commits that apply to that file. When the file is clicked in the visualization, those commits are loaded via a request to the server, which gets the commits from Git.

B. Performance

By default, JavaScript does not support using more than one thread, however, Node.js worker threads[3] allow for proper multi-threading. We investigated this for use in Git Truck, but soon discovered, that the primary bottleneck in the analyzer, is getting the output from the Git process. Because these are run as child-processes, spawning multiple of them, will utilize a thread per process, meaning that worker threads are not necessary to multi-thread Git Truck. The principle behind multi-threading the analyzer is therefore, that we should be able to spawn multiple Git processes outputting commits, and those outputs will be parsed in a single-threaded manner by the Git Truck server process. This is shown in figure 1, as the threads run in parallel between "Start threads" and "Synchronize threads", and applying credit is done synchronously afterward. The benefit of multi-threading is dependent on the speed the Git processes can generate output, which is done by reading from the .git-folder, so performance is likely quite I/O-bound. We did not look further into this, but further research could be done about the impact of disk speed on analysis time.

This multi-threaded approach stands in contrast to the old architecture of the commit analysis, shown in figure 2, where Git log output is gathered from a single thread, credit is applied in the same single-threaded way.

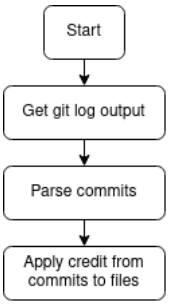

Figure 2: Flowchart showing the old structure of the hydrateData function.

[3]https://nodejs.org/api/worker_threads.html

Originally, the analyzer expected commits to be analyzed in reverse chronological order, to be able to follow renames of files. Since commits are now gathered from multiple Git processes, the order is no longer guaranteed. It is therefore necessary to synchronize the threads every *commitBundleSize* number of commits, to apply credits and follow renames. This creates a bottleneck, as some threads will have to wait for others before proceeding, so for performance, *commitBundleSize* should be as big as possible, without causing Node.js to run out of memory, as described in II-A. This synchronization is also used to provide the progress indicator, as described in section II-C, so a balance between frequent progress updates and performance has to be found. We decided to set the value dynamically between 10k and 150k depending on the number of commits to be analyzed. The time to analyze using different *commitBundleSize* is shown in figure 3. Improved performance is achieved up to 200k, after which the memory consumption increases to the point, that garbage collection hurts performance.

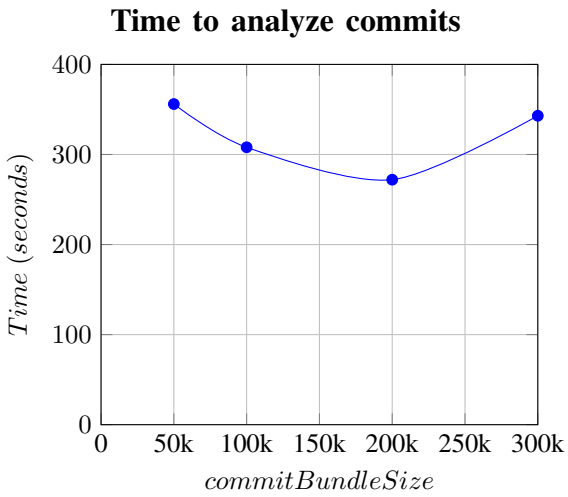

Figure 3: Time to analyze commits for the Linux repository with varying commitBundleSize. Tested with 4 threads.

Another performance optimization, it that whenever the analyzer needs to apply credit to a file, it finds that file by searching a file-tree structure with logarithmic look-up time. We improved this, by building a map of files instead, such that we get constant-time look-up.

C. Progress indicator

Git Truck does not provide a way for the user to know the progress of analysis, which causes a poor user experience.[4] The addition of multi-threading makes it more difficult to make an accurate progress indicator, as threads can be at different stages of progress. However, the point where threads are synchronized can be used as a checkpoint to know how many commits have been analyzed so far. The implementation we have made focuses on giving frequent progress updates when run on large projects, which have the longest analysis time. For small projects, analysis is over in an instant, so a

progress indicator is not necessary. As a result, for a project with 50k commits, progress will be updated in increments of 25%, while a project with 1.2 million commits, will be updated in increments of 12%. Figure 4 shows the progress indicator.

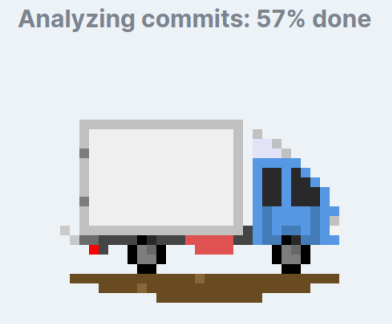

Figure 4: Progress indicator on the loading screen.

The progress indicator has three states:

- "Starting analysis", for the loading of initial information, such as the file tree.
- "Analyzing commits", which is accompanied by a percentage of progress. This step dominates the overall time spent on analysis, so a percentage of progress is most useful at this step.
- "Generating chart", which is shown during the time it takes for the browser to generate and render the chart.

III. Visualization improvements

A. Pruning small nodes

As a coincidence when we were looking at the code, we noticed that the function designed to avoid rendering sub-pixel-size elements had incorrect cut-off values. The configuration meant that for the bubble-chart, folders would be rendered regardless of their size. For the tree-map, it was even worse. Instead of checking the width and height of the rectangles, we checked the x- and y-coordinates of the upper left corner. This value is practically always above the cutoff, so this check did nothing. We resolved this issue and also added the functionality of allowing the user to set the cut-off. So if the user wants higher performance, the cut-off can be increased, if a higher detail level is desired, the cut-off can be lowered.

B. Color-generation

Through profiling using the Chrome dev-tools, we recognized that generating the colors for authors took a lot of time, roughly 30 seconds for 10,000 authors. Importantly, this is done whenever the chart is loaded, so even if a user had analysis cached, this would still take a long time. We found that the culprit for this performance was the library we used to generate the colors.[4] This library is designed to generate visually distinct colors, which likely means that the time-complexity scales exponentially with the number of colors to generate. In practice, we had often found that the colors generated were not always very distinct, so we decided that a solution could be to simply use random colors.

We now employ a library, which uses a string as a seed to generate a color. We use the author's name as this seed, which means that we get the added benefit of author-colors being consistent when changing branches. However, in some cases, the generated colors are not sufficiently distinct, so we have added a button to re-roll the colors. This new implementation means generating colors less than a second for 10,000 authors.

C. Rendering with WebGL

We investigate how using WebGL over SVG improves rendering performance. WebGL can utilize hardware acceleration[7], which has previously been shown to improve rendering performance over SVG.[2]

Many libraries utilize WebGL for different purposes. We implemented the visualization using React Pixi, which utilizes Pixi.js to render graphics with WebGL. This tech-stack suits our needs well, as it is designed for rendering 2D graphics, and has good feature parity with SVG because it can handle transitions with React Spring and on-click events.

For the proof of concept, we did however not achieve full feature parity. Notable missing features are:

- Text labels. These are supported, but we did not figure out how to place them like we do in SVG.
- Transitions on size. We did implement transitions on location and color, but not size. This means that when a folder is clicked, files snap to their new size instantly, and then animate the movement to their new location. Subjectively, this looks okay for tree-map but is quite disorienting for bubble chart.
- Smooth lines. Due to WebGL being rasterized, curved lines appear more jagged than when using SVG, despite anti-aliasing being enabled.

These problems can likely be solved with further development, though we believe that our implementation is sufficient to evaluate whether to adopt this render method. However, rendering text is usually one of the most demanding tasks, so omitting this from the testing means that performance will be better than if text was also rendered.

IV. Results

A. Analyzer run-time

We test the time to analyze the Linux repository[5] using different numbers of threads. The testing methodology can be found in appendix A. The results are shown in figure 5. We can see that going from 1 to 2 threads, reduces analysis time by 43%, and going from 2 to 4 threads reduces the time by 38% further. Beyond 4 threads, there are significant diminishing returns from additional threads, and in the case of 12 threads, the additional threads hurt performance. This is because each Git process needs its own memory, so the memory used scales roughly linearly with the number of threads, and if the entire

[4]https://www.npmjs.com/package/distinct-colors

[5]https://github.com/torvalds/linux

system memory is used, performance will be hit due to garbage collection and having to use swap memory instead of RAM.

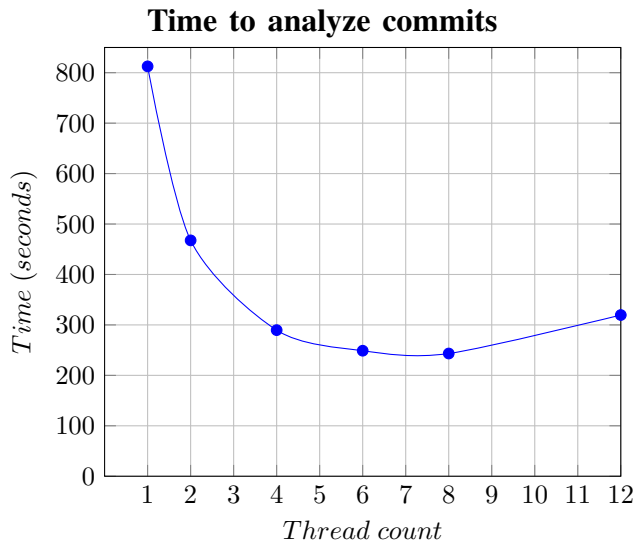

Figure 5: Time to analyze commits for the Linux repository.

Due to the diminishing returns and memory consumption, we have decided to use 4 threads when the system has >4 threads and 2 threads otherwise. This is because using all system resources will make the entire system unresponsive during analysis, which is a poor user experience.

B. Rendering performance

We test the performance difference between the implementation using WebGL, and the existing Git Truck rendering method of SVG. We vary the pixel cutoff value, which determines at what size elements should be considered too small to be worth rendering. Thus, lower cutoff means more items to render, with 0 meaning all items will be rendered. Figure 6 shows the results.

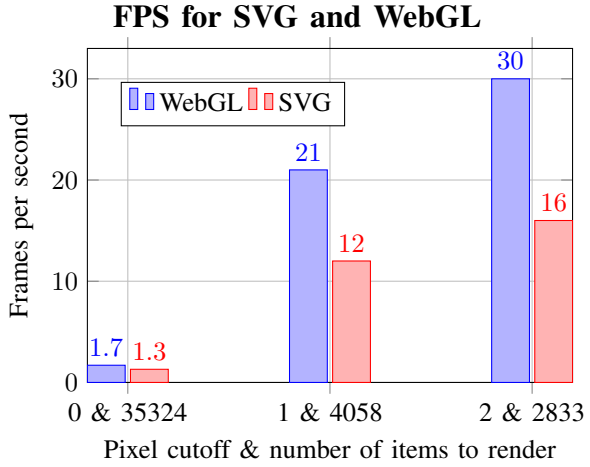

Figure 6: FPS using different rendering methods and pixel cutoffs.

Given these results, we decided to set the default cutoff value to 2, as this provides substantially better performance than when it is set to 1, and subjectively, the visual difference between the two is minuscule.

We observe big performance improvements by utilizing WebGL, with close to double the FPS. We can also see, that for a big project, the number of elements to render increases drastically if small elements are included, which hurts performance. The result of SVG with a 0-pixel cutoff is equivalent to what Git Truck was using before the performance improvements listed in this paper. Thus, we conclude that the biggest performance improvement comes from pruning small elements before rendering, rather than the method used for rendering. That said, if the WebGL implementation can be made to have feature parity with SVG, we should change the rendering method to WebGL, as this also brings substantial FPS gains.

One notable observation that is not visible on the chart, is that the animation for WebGL at pixel cutoff 0 took over 9 seconds to complete, whereas for SVG the duration was 1.5 seconds. In that case, SVG gave a better user experience, despite a lower FPS.

None of the rendering methods achieve the desired 60 FPS, however, we do not consider this a problem. The Linux repository is practically a worst-case scenario, and for many smaller repositories, Git Truck achieves close to 60 FPS for both SVG and WebGL.

For a video demo of the rendering performance as well as the WebGL implementation of the chart, see appendix C.

V. Conclusion

A. Summary of results

Through the implementation of the improvements mentioned, we have achieved the following, which increases Git Truck's scalability.

- Ability to analyze projects with more than 1 million commits without crashing. Up from about 300k commits before.
- Reduced the time to analyze by about 60% through the use of multi-threading, as well as adding a progress indicator.
- Improved animation FPS by more than 10x by avoiding rendering small nodes, and additionally made a proof of concept for how to use WebGL to improve FPS by a further roughly 75% on large repositories.
- Identified inefficient computations via the browser devtools profiler. Including the generation of author colors, and the calculation of author distribution percentages. Color generation would take 30 seconds for the Linux repository, which has been reduced to less than a second, so generating the chart is much faster. Calculating author percentages could take a few seconds whenever a folder or file was clicked, which has been reduced to being nearly instant.

B. Future work

We have identified the following ways to continue the work of this paper.

- Make a full implementation of the visualization using WebGL with all of the features currently in Git Truck re-implemented.
- Enable fast querying of the commits, in order to enable new metrics. Now that analysis and visualization are substantially faster, the main restriction to making more advanced visualizations is the lack of efficient querying of the data.
- Investigate the impact of disk read speed on analysis time. Perhaps the disk becomes the bottleneck on systems without an NVMe ssd, such that using multiple threads does not improve performance, and maybe it even hurts performance.

APPENDIX

A. Analyzer test methodology

All tests were run twice, and the reported result is the average of these two runs. The tests were run with the flag **–max-old-space-size=4096**, which sets the maximum memory nodejs can use to 4 GB. No other tasks were done while analysis ran. Before each run, the previous Git Truck process was killed, and a new one was started. The time was measured for the **hydrateData** function using **performance.now**[6] called right before and after execution.

The following table shows the specifications of the system used.

Table I: System specifications

CPU model	AMD Ryzen 5 5600
Cores / threads	6 / 12
Frequency, base / boost	3.5 / 4.4 GHz
RAM	16 GB 2933 MHz DDR4
GPU	AMD Radeon RX 6700 XT 12 GB
Graphics driver version	23.9.3
Storage	2 TB NVMe PCIe 3.0 ssd
Browser window	1920x1080, 60 Hz
Operating system	Windows 10 22H2 19045.3693
Browser	Chrome version 120.0.6099.71
Node.js version	18.12.1
Git version	2.38.1.windows.1
Git Truck version	1.8.4

B. Rendering test methodology

The system used is same one used in analyzer test, shown in appendix A. The animation duration was set to 1 second, which was done in CSS for SVG and in the configuration of React Spring for WebGL. The transitions were recorded at 60 fps with OBS Studio[7]. The number of frames were then counted manually, by stepping one frame at a time through the recording. FPS was then derived by dividing the number of frames, by the actual duration of the transition.

For the test, we visualized the Linux-repository as of commit **791c8ab**. We zoomed from the root-folder to the drivers-folder using the tree map.

C. Links

- Rendering performance demo: https://youtu.be/xDhPc29ToBg
- React Pixi chart implementation: https://github.com/git-truck/git-truck/blob/pixi/src/components/HierarchyChart.tsx
- Multi-threaded analyzation implementation: **hydrateData** function in https://github.com/git-truck/git-truck/blob/main/src/analyzer/hydrate.server.ts
- Pull requests related to improvements made: https://github.com/git-truck/git-truck/pulls?q=is%3Apr+milestone%3A%22Fast+truck+research%22

[6]https://nodejs.org/api/perf_hooks.html#performancenow

[7]https://obsproject.com/

REFERENCES

[1] K. Højelse et al. “Git-Truck: Hierarchy-Oriented Visualization of Git Repository Evolution”. In: *2022 Working Conference on Software Visualization (VISSOFT)*. 2022. DOI: 10.1109/VISSOFT55257.2022.00021.

[2] T. Horak, U. Kister, and R Dechselt. “Comparing Rendering Performance of Common Web Technologies for Large Graphs”. In: (2018). URL: https://mt.inf.tu-dresden.de/cnt/uploads/Horak-2018-Graph-Performance.pdf.

[3] A. Lindberg. “Performance Evaluation of Java-Script Rendering Frameworks”. In: (2020). URL: https://www.diva-portal.org/smash/get/diva2:1411632/FULLTEXT01.pdf.

[4] Brad A. Myers. *The Importance of Percent-Done Progress Indicators for Computer-Human Interfaces*. URL: https://dl.acm.org/doi/pdf/10.1145/1165385.317459.

[5] *Node.js: The Maximum Size Allowed for Strings/Buffers*. URL: https://www.kindacode.com/snippet/node-js-the-maximum-size-allowed-for-strings-buffers/. (accessed 06-12-2023).

[6] *Ouput from v8.getHeapStatistics()*. URL: https://nodejs.org/api/v8.html#v8getheapstatistics. (accessed 12-12-2023).

[7] *WebGL API*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API. (accessed 08-12-2023).