



university of  
 groningen

FACULTY OF MATHEMATICS AND NATURAL SCIENCES  
DEPARTMENT OF COMPUTING SCIENCE

SOFTWARE MAINTENANCE AND EVOLUTION

---

# The Blender Project

## A Software Evolution Analysis

---

*Authors:*

Thomas HOEKSEMA (S2349639)

Michel MEDEMA (S2396009)

*Supervisor:*

prof. dr. Alexandru TELEA

November 15, 2015

# Contents

<b>Contents</b>	<b>I</b>
<b>List of Figures</b>	<b>II</b>
<b>List of Tables</b>	<b>IV</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Assignment . . . . .	1
1.2 Software Repository . . . . .	1
1.3 Tools . . . . .	3
1.4 Overview . . . . .	5
<b>2 Basic Repository Investigation</b>	<b>6</b>
2.1 Versions and commits . . . . .	6
2.2 Folders and files . . . . .	8
2.3 Developers . . . . .	14
<b>3 Initial Visual Overview</b>	<b>18</b>
3.1 Choice of views . . . . .	18
3.2 Stable development periods . . . . .	20
3.3 Unstable development periods . . . . .	20
3.4 Current state . . . . .	24
<b>4 Author Analysis</b>	<b>25</b>
4.1 Main contributors . . . . .	25
4.2 Replacing chief developer . . . . .	28
4.3 Type correlation . . . . .	31
4.4 Location correlation . . . . .	33
<b>5 Code Size Analysis</b>	<b>37</b>
5.1 Size evolution . . . . .	37
5.2 Most turbulent files . . . . .	43
5.3 Source code fraction . . . . .	45
<b>6 Complexity Analysis</b>	<b>47</b>
6.1 Most complex files . . . . .	48
6.2 Changes in complexity . . . . .	49
6.3 Activity correlation . . . . .	53
6.4 Size correlation . . . . .	56
<b>7 Dependency Analysis</b>	<b>59</b>
7.1 Data modelling . . . . .	59
7.2 Data visualisation . . . . .	62
7.3 Conclusion . . . . .	65

<b>8</b>	<b>Evaluation</b>	<b>66</b>
8.1	Summary . . . . .	66
8.2	Tools . . . . .	67
8.3	Analysis . . . . .	68
8.4	Conclusion . . . . .	68
<b>9</b>	<b>Acknowledgements</b>	<b>69</b>
<b>10</b>	<b>References</b>	<b>69</b>
<b>11</b>	<b>Appendices</b>	<b>70</b>
11.1	Time Tracking . . . . .	70
11.2	File extensions . . . . .	71
11.3	Revisions of potential replacements for chief developer . . . . .	74
11.4	Details of authors metric per file type . . . . .	77
11.5	Details of authors metric per (sub-)directory . . . . .	79
11.6	Relevant file views . . . . .	81

## List of Figures

1	Version velocity of the Blender Project . . . . .	6
2	Filelight overview of repository . . . . .	9
3	Filelight overview of repository without the .git folder . . . . .	9
4	Filelight overview of repository without non-source files . . . . .	12
5	Filelight overview of /blender/source, without non-source files . . . . .	13
6	Filelight overview of /blender/source/blender, without non-source files . . .	13
7	File view of Lines of Text (LOT) metric . . . . .	19
8	File view of Lines of Code (LOC) metric (code size) . . . . .	19
9	Evolution view of file count (including code size metric) . . . . .	19
10	Views for determining (un)stable periods . . . . .	21
11	Stable development periods . . . . .	22
12	Unstable development periods . . . . .	23
13	View of all files including authors metric . . . . .	25
14	Legend of authors for entire repository . . . . .	26
15	Contributions of the top 20% authors . . . . .	26
16	Evolution view of all authors . . . . .	27
17	Evolution view of top 20 authors . . . . .	27
18	Evolution view of top 10 authors . . . . .	27
19	View of all files including authors metric (without Campbell Barton) . . . .	29
20	Evolution view of top 10 authors (excluding Campbell Barton) . . . . .	29
21	View of all files including authors metric (detail) . . . . .	30
22	Legend of file types . . . . .	31
23	File view of entire project including file types metric . . . . .	32
24	File view of authors correlated with file types . . . . .	32
25	Legend for the folder metric (top-level directories) . . . . .	34
26	File view of the entire project including folders metric . . . . .	34
27	File view of authors correlated with folders . . . . .	34
28	Legend for the folder metric (/source/blender sub-directories) . . . . .	35

29	File view of /source/blender files including folders metric . . . . .	36
30	File view of authors correlated with folders (in /source/blender) . . . . .	36
31	Legend for Code Size metric . . . . .	37
32	Code size metric for all files . . . . .	38
33	Code size metric for all source files . . . . .	38
34	Code size metric for all source files (excluding header files) . . . . .	39
35	Code size metric for all source files (sorted by creation without headers) . .	39
36	Trend view of code size metric for all source files (excluding header files) . .	39
37	Periods for evaluation the code size evolution . . . . .	40
38	Evolution view of code size for S1. . . . .	41
39	Evolution view of code size for S2. . . . .	41
40	Evolution view of code size for S3. . . . .	41
41	Evolution view of code size for S4. . . . .	42
42	Evolution view of code size for S5. . . . .	42
43	Evolution view of code size for S6. . . . .	42
44	Evolution view of code size for S7. . . . .	43
45	First step to obtain growing/shrinking files . . . . .	44
46	Second step to obtain growing/shrinking files . . . . .	44
47	Most growing and/or shrinking files . . . . .	45
48	Fraction of files with a code size metric . . . . .	46
49	Legend for the Total Complexity metric (under Complexity) . . . . .	47
50	Evolution view of Total Complexity for all files . . . . .	47
51	All source files, highlighting highest complexity category . . . . .	48
54	Files with lowest complexity . . . . .	50
55	Files with lowest complexity in current revision . . . . .	50
56	Files that increase most in complexity . . . . .	51
57	Files that decrease most in complexity . . . . .	51
58	Files increasing most in complexity (folder metric) . . . . .	52
59	Files decreasing most in complexity (folder metric) . . . . .	52
60	Files increasing in complexity (sorted by activity) . . . . .	53
61	Files decreasing in complexity (sorted by activity) . . . . .	53
62	Currently most complex files (sorted on activity) . . . . .	54
63	Currently least complex files (sorted on activity) . . . . .	54
64	Complexity of all source files (sorted on activity) . . . . .	55
65	Complexity of all source files (highest complexity and activity) . . . . .	55
66	Complexity of all source files (lowest complexity and activity) . . . . .	56
67	Source files grouped by lowest complexity . . . . .	56
68	Source files grouped by lowest complexity (code size metric) . . . . .	57
69	Source files grouped by highest complexity . . . . .	58
70	Source files grouped by highest complexity (code size metric) . . . . .	58
71	Proposed data model for the visualisation tool . . . . .	60
72	Visualisation of a call graph for the proposed visualisation technique . . . .	63
73	Revisions of Campbell Barton . . . . .	74
74	Revisions of Sergey Sharybin . . . . .	74
75	Revisions of Bastien Montagne . . . . .	74
76	Revisions of Lukas Toenne . . . . .	75
77	Revisions of Brecht van Lommel . . . . .	75
78	Revisions of Tamito Kajiyama . . . . .	75

79	Revisions of Antony Riakiotakis . . . . .	76
80	Revisions of Ton Roosendaal . . . . .	76
81	Revisions of Joshua Leung . . . . .	76
82	Detail of authors metric for .cmake files . . . . .	77
83	Detail of authors metric for .jpg files . . . . .	77
84	Detail of authors metric for .png files . . . . .	77
85	Detail of authors metric for .rst files . . . . .	78
86	Detail of authors metric for .spild files . . . . .	78
87	Detail of authors metric for .osl files . . . . .	78
88	Detail of authors metric for /doc folder . . . . .	79
89	Detail of authors metric for /extern folder . . . . .	79
90	Detail of authors metric for /intern folder . . . . .	80
91	Detail of authors metric for /source/blender folder . . . . .	80

## List of Tables

1	Distribution of files and sizes . . . . .	10
2	Distribution of files and sizes (only source code files) . . . . .	12
3	Distribution of files and sizes (only source code files) . . . . .	70
4	File extensions in entire repository . . . . .	71
5	File extensions in the source folder . . . . .	72
6	File extensions in the extern folder . . . . .	72
7	File extensions in the intern folder . . . . .	72
8	File extensions in the release folder . . . . .	73

## Listings

1	Git installation and pulling the Blender project . . . . .	6
2	Fetching amount of commits . . . . .	7
3	Fetching the initial commit . . . . .	7
4	Fetching the most recent commit . . . . .	8
5	Bash script for counting file extensions . . . . .	10
6	Commands for deleting non-source files . . . . .	11
7	Fetching the commit in the middle of the project . . . . .	14
8	Fetching the date-based middle commit . . . . .	14
9	Obtaining amount of commits in date-based halves . . . . .	15
10	Counting developers in halves and entire project . . . . .	15
11	Obtaining activity per author in commit-based halves . . . . .	16
12	Obtaining activity per author in date-based halves . . . . .	17
13	Files that grow/shrink the most in terms of LOC (60 files) . . . . .	81
14	Files that increase most in complexity (63 files) . . . . .	82
15	Files that decrease most in complexity (30 files) . . . . .	83

# 1 Introduction

This report was created as part of the final assignment for the course of Software Maintenance and Evolution at the University of Groningen, in the course year of 2015/2016. The course serves as an introduction to the main principles and techniques behind software maintenance and evolution. In the course, several topics are described such as the theory of software evolution and various maintenance types and techniques, as well as the tools that are involved in software maintenance, such as tools that measure quality metrics, detect evolution trends, or analyse various types of artifacts involved in large software repositories.

As part of the course, a practical assignment is given in which the students analyse an Open Source repository of their choice with the tools, techniques and theory that have been introduced during the lectures. Several questions are to be answered concerning the structure and evolution trends of the maintenance activities performed in the repository. This report will contain the answers to the questions that are set in the description of the assignment.

## 1.1 Assignment

We have chosen Assignment A, since we were more interested in applying the techniques and theory that we have seen during the lectures on a large Open Source repository rather than programming (part of) an application that applied software analysis techniques or provided visual overviews of the maintenance and evolution of a repository.

The assignment consist of the task, given a software repository, to perform several analyses in order to assess the maintainability, modularity, complexity, and quality of the software in the repository, as well as the development process. To reach this conclusion, the assignment is divided in five distinct steps of increasing difficulty. Before these steps are elaborated upon, we must first elaborate on our choice for the repository, on the tools that we have used, and finally we will give an overview of this document.

## 1.2 Software Repository

In this section we will describe the process of choosing a repository for this assignment. First of all, there are some requirements that are applicable to the repository that indicate its suitability for this assignment. In particular:

- The repository should be written in C or C++, since the metric calculators of the tools that are provided accept these files when calculating metrics.
- The revisions to the repository should be performed by a many developers, so that the analysis w.r.t. authors is not trivial.
- The repository should contain more than 5,000 files, and no more than 10,000 (approximately).
- The repository should contain more than 10,000 revisions, and no more than 100,000 (approximately).
- We should be able to contact the repository without the need of creating an account, or creating an account and acquiring read-only access should be relatively simple.

- The repository software should not randomly block connections when requesting the content of a large amount of files in a short time period, since the contents of the files are needed for some analysis methods in the tools that are provided.

### 1.2.1 Proposals

Before coming to our final choice for the repository to analyse, we considered various options. The following is a list of alternatives that we considered:

- **Mozilla Thunderbird:** a free, open source, cross-platform email, news, and chat client developed by the Mozilla Foundation. The software is written mostly in C-based languages, which is suitable for the project. Furthermore, the project has a large amount of developers and a very active release cycle, but may be even too big to analyse for this assignment.
- **VirtualBox:** a hypervisor for x86 computers from Oracle Corporation. The project is mostly in C with a fraction of C++. The downside of this repository is that it has not existed for a very long time yet.
- **KeePass:** a free, open source, cross-platform and light-weight password management utility. Unfortunately, it seems that a large amount of the repository is in C#, which we do not think is supported very well by the metric calculators used in SolidTA.

### 1.2.2 Blender

We established that the Blender project, by the Blender Foundation, would be the repository that is the subject of the first five steps of this assignment. Blender is a free and Open Source 3D creation suite, which supports rigging, animation, simulation, rendering, compositing and motion tracking, as well as video editing and game creation, which is basically the entirety of the 3D pipeline-modelling. Blender also gives advanced users the possibility to use their API for Python scripting to customise the application and write specialised tools. [1] [6]

Blender is cross-platform and interfaces with OpenGL to provide a consistent experience. The project guarantees extensive testing by the development team for the list of supported platforms and drivers. The project is listed under the GNU General Public License. [1] [6] We both think that the Blender project is a particularly interesting software application that one of us has used quite a bit in the past, and therefore we were interested in analysing this repository for this assignment.

From a more technical side of view related to the assignment, we have determined that the Blender project has around 60,000 commits, around 250 to 300 unique developers who have submitted revisions, and consists of around 8500 files in the latest version of the project. The majority of the source base is written in C, another large fraction in C++, and finally there is a minority of source code that is written in Python, particularly for plug-ins and for the API for advanced users as described above. All of these observations lead to our belief that the repository is suitable for the assignment.

Furthermore, the Git repository of the project, listed further on in this introduction, supports anonymous access and we have experienced no connectivity issues that were related to the repository software rejecting our requests for data. This simplified the process of obtaining the files, revisions, and content of those files and revisions greatly.

### 1.3 Tools

This section will describe the tools that will be used for the elaboration of the first five steps of the assignment.

#### 1.3.1 Repository access

The Blender project has an active repository, which is Git-based, and also has a web interface at <https://git.blender.org/gitweb/gitweb.cgi/blender.git> while the repository itself can be pulled from <https://git.blender.org/blender.git>.

To get access to the repository via command in Linux, which is necessary for the first step of this assignment, we have used the Git command line tool that is available in the repositories of all major Linux distributions. In *Basic Repository Investigation*, the installation procedure of this tool will be described.

SolidTA does not come with Git supported integrated, instead the Git functionality needs to be manually installed by the user and SolidTA must be able to find this tool via the PATH variable on Windows. We installed msysgit, a Git implementation for Windows-based machines, for which we put the path in the PATH variable. After installing this tool and restarting SolidTA, the latter was able to access the former and we could access the Blender project repository through SolidTA.

#### 1.3.2 Repository analysis

We will make use of SolidTA, or Solid Trend Analyzer, which is a software application that helps understanding, analysing and managing the evolution of software projects recorded in software repositories such as Subversion or CVS, but is also capable of communicating with Git repositories, provided the correct tools are installed as indicated above.

Due to the fact that we have made heavy use of the selections feature in SolidTA, and these selections are also delivered as part of this assignment, we will elaborate on the contents of each selection here:

- **blender** : Contains all files in the entire project.
- **blender/source** : Contains all files in the /source top-level directory.
- **blender/source/blender** : Contains all files in the /source/blender directory.
- **blender/release** : Contains all files in the /release top-level directory.
- **blender/extern** : Contains all files in the /extern top-level directory.
- **blender/intern** : Contains all files in the /intern top-level directory.



- **blender/doc** : Contains all files in the /doc top-level directory.
- **blender/build\_files** : Contains all files in the /build\_files top-level directory.
- **Lines of Text** : Contains all files for which the "Lines of Text" (LOT) metric could be calculated.
- **Code Size** : Contains all files for which the "Code Size" (measured by Lines of Code, LOC) metric could be calculated.
- **Code Size (without headers)** : The same as Code Size, without all header files. (.h, .hpp, .hh, etc.)
- **Most Growing or Shrinking** : A list of files that have the highest variety of change regarding the Code Size (LOC) metric.
- **Complexity** : Contains all files for which the "Total Complexity" (part of Complexity) metric could be calculated.
- **Most Complex** : Contains all files for which a revision falls in the highest category of the Total Complexity metric.
- **Least Complex** : Contains all files for which a revision falls in the lowest category of the Total Complexity metric.
- **Most Complex - Currently Max** : The same as Most Complex, without the files that are currently not a part of the highest category of Total Complexity.
- **Least Complex - Currently Min** : The same as Least Complex, without the files that are currently not a part of the lowest category of Total Complexity.
- **Complexity - Max Increase** : All files of "Most Complex - Currently Max" that have ever had a revision that falls in the lowest category of Total Complexity.
- **Complexity - Max Decrease** : All files of "Least Complex - Currently Min" that have ever had a revision that falls in the highest category of Total Complexity.
- **S1 to S7** : File categories that are used in the analysis in the *Code Size Analysis*.

We used the Filelight software in the *Basic Repository Investigation* to display the structure, relative sizes, amount of files and contents of the directories of the repository. Filelight is available on most Linux repositories via apt-get.

Finally, in the *Basic Repository Investigation*, we also made use of some bash scripts to count the amount of files and the total file size for all extensions of a given folder.

## 1.4 Overview

The structure of the rest of this document will now be described.

- Section 2 (*Basic Repository Investigation*) covers the analysis of the structure of the repository, as well as finding out the amount of revisions, authors, and the distribution of files among its top-level directories and sub-directories. Furthermore, a list of most active developers in halves of the project will be given.
- Section 3 (*Initial Visual Overview*) utilises SolidTA to give an initial overview of the stability of the repository over the entire lifetime of the project.
- Section 4 (*Author Analysis*) performs several analyses regarding the authors of the revisions to the project. The main contributors, chief developer, and replacements for the chief developer will be determined, as well as correlations between files and developers.
- Section 5 (*Code Size Analysis*) gives an analysis on the evolution of the code size in the repository. Furthermore, the fastest growing and shrinking files will be determined.
- Section 6 (*Complexity Analysis*) performs a complexity analysis on the repository, in particular the most complex files will be determined and the fastest increasing/decreasing files w.r.t. complexity. Several correlations between activity/code size and complexity will also be investigated.
- Section 7 (*Dependency Analysis*) consists of an essay that was written based on the insight accumulated during this assignment, in which a method will be described to model and visualise the call graphs and their evolution of a specific repository.
- Section 8 (*Evaluation*) gives an evaluation on the assignment, including a summary of the obtained results, our experiences and difficulties during the assignment and some final words.
- Sections 9, 10 and 11 (*Acknowledgements*, *References* and *Appendices*) includes all of the detailed knowledge bases that were also utilised in the document, but were not suitable to include in-line.

## 2 Basic Repository Investigation

The structure of the Blender repository will be investigated and described in this section. The Blender repository is a git repository, so to analyse the structure of the repository, the git client was installed which is available through the software repositories of all major Linux distributions. On Ubuntu 14.04, the following set of commands were executed to install the git client and to pull the contents of the Blender project repository:

Listing 1: Git installation and pulling the Blender project

```
> sudo apt-get install git
> git clone https://git.blender.org/blender.git
```

In the following sections, we will make use of this git client when answering questions about the repository.

### 2.1 Versions and commits

The Blender project has been through many versions as is shown in Figure 1, which depicts all of the official (stable, not including alpha/beta builds) releases of the Blender project. The graph shows that the amount of time per version between versions 2.25 and 2.36, as well as between 2.57 and onwards, is very regular and does not show major variations in development time. However, between 2.36 and 2.57, there is a large variation in the time it takes to advance a version. We have looked at various sources on the Blender Foundation websites to find an explanation for this irregular behaviour and sudden stabilisation from 2.57 onwards.

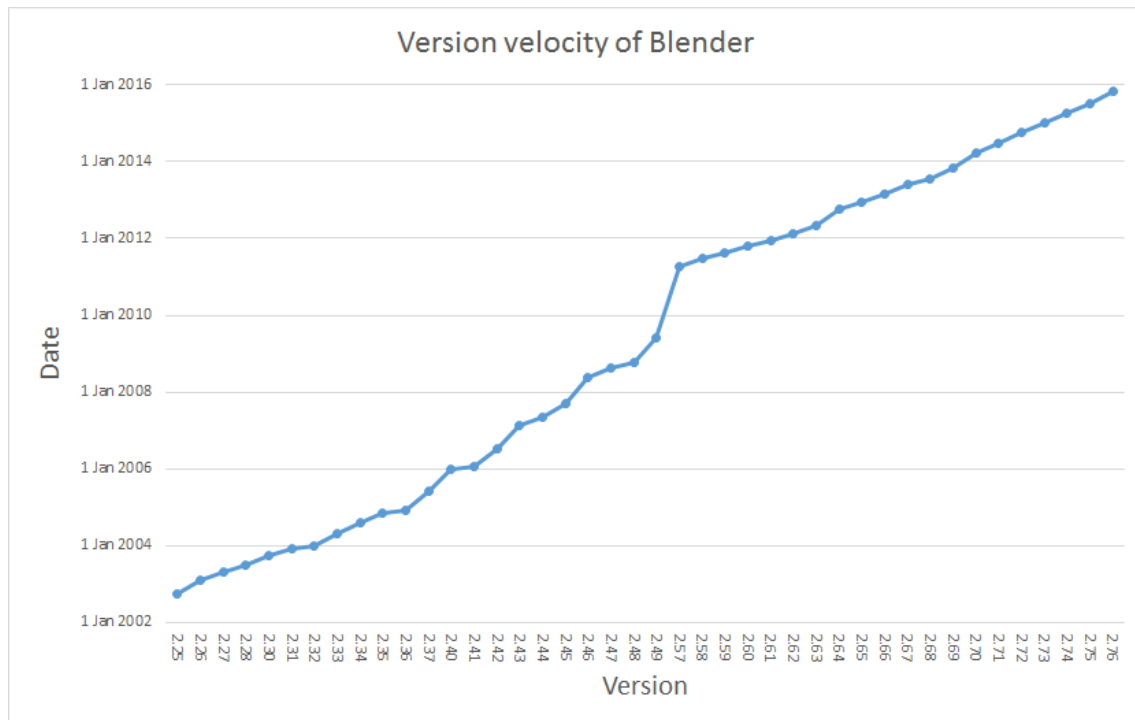


Figure 1: Version velocity of the Blender project. [3] [4]

In fact, the reason for the stabilization of the graph is mentioned on [2], which is an "Ask Me Anything" page for developers to the chief developers of the project. In particular, it is stated that before 2.57, the project simply released versions of the software when it was ready instead of having a fixed release schedule. The chief developers describe several disadvantages of the former method which led to the choice of a fixed schedule. They claim that the switch had to be made due to a large amount of volunteering developers in many active areas of the project and the high degree of complexity of the Blender application, which made it difficult to pick a moment at which all areas of the application were in a finished state.

Furthermore, due to the long period between releases that were caused by those aspects, bugs and broken code were allowed to exist within the stable releases for too long. Developers would also try to push changes right before the release of a new version, since otherwise they may have to wait for another 6 months to a year to incorporate their updates into the project, which led to improperly tested code in the release versions. Users also had to wait a long time before they could utilize new features of a version due to the long release times. All of these arguments led to the adoption of a new release schedule, where the project would generate a stable version every so many months.

Now, we will look into the structure of the active git repository of the Blender Foundation, which hosts the Blender project. First of all, the total number of revisions in the software will be determined, by counting the total number of commits that have been performed on the repository. Executing the following command results in the number of commits:

Listing 2: Fetching amount of commits

```
> git rev-list HEAD --count
61918
```

Executing this command shows that the entire repository contains a total of 61918 commits. It is also interesting to see when the first and last commit of the project have been made, to determine the lifetime of the project up to this point. To fetch the first commit (which is the last in the log), the following command is executed:

Listing 3: Fetching the initial commit

```
> git log --reverse | head -5
commit 12315f4d0e0ae993805f141f64cb8c73c5297311
Author: Hans Lambermont <hans@lambermont.dyndns.org>
Date: Sat Oct 12 11:37:38 2002 +0000
```

Initial revision

The first part of the command results in the full history of all of the commits. By default, the history is sorted in such a way that the last commit that has been made is at the top of the list. The option that is being passed along with this command makes sure that the ordering is reversed, listing the first commit that has been done at the start of the project at the top instead.

The result of this command is passed along to the "head" command, which takes the first n lines from the output. Since a single commit consists of multiple output lines, the first five lines are requested, which is sufficient to obtain the date on which the commit was performed. The output shows that the first commit seems to have been made on Saturday, October 12, 2002, at 11:37:38 +0000.

Besides the initial commit, the most recent commit that has been pushed must also be identified. To obtain the last commit, the following command is used:

Listing 4: Fetching the most recent commit

```
> git log -n 1
commit 46f452e96baaf9424582003e87736840ccbbffca
Author: Campbell Barton <ideasman42@gmail.com>
Date: Fri Nov 13 00:03:12 2015 +1100

    Fix error cutting node links

    Accessing theme from outside drawing code isn't reliable, pass space
    -type.
```

Passing the option "-n 1" to the command ensures that only the newest commit in the log is shown. As can be viewed in the output, the most recent commit at the time of performing this command was on Friday, November 13, 2015, at 00:03:12 2015 +1100. To summarise, this means that the lifespan of the repository at this point is around 13 years, from 2002 to 2015, with 61918 commits performed during that time.

## 2.2 Folders and files

We will now look at the file structure, especially the top-level folders, of the Blender repository and the amount and types of the files within the project. To show an overview of the top-level folders and their respective sizes in terms of disk space, we use Filelight to display the contents of the repository, as seen in Figure 2. We want to find out what the largest top-level directories are, which are defined as the top-level directories that either contain a large amount of files respective to the other folders, or take up a large amount of disk space respective to the other folders.

The first impression of this figure is that the ".git" folder takes up the majority of disk space of the entire project. Since this folder contains metadata which is necessary for the operation of git, we decided to leave this folder out of the analysis and simply to delete it from our analysis. After deleting the ".git" folder, we obtain Figure 3 using Filelight on the folder of the repository. This figure is much more interesting, since it only shows the components that are actually involved in the Blender application. The data that has been obtained through the inspection of the folders and Figure 3 can be found in Table 1. The "Properties" window of the file explorer in Ubuntu 14.04 was used to obtain the data in the table. The conclusion is that "source", "release", "extern" and "intern" are the largest top-level directories both in terms of disk space and the amount of files. The other top-level directories, namely "doc", "build\_files", "tests" and "scons", are significantly smaller in both definitions of size.

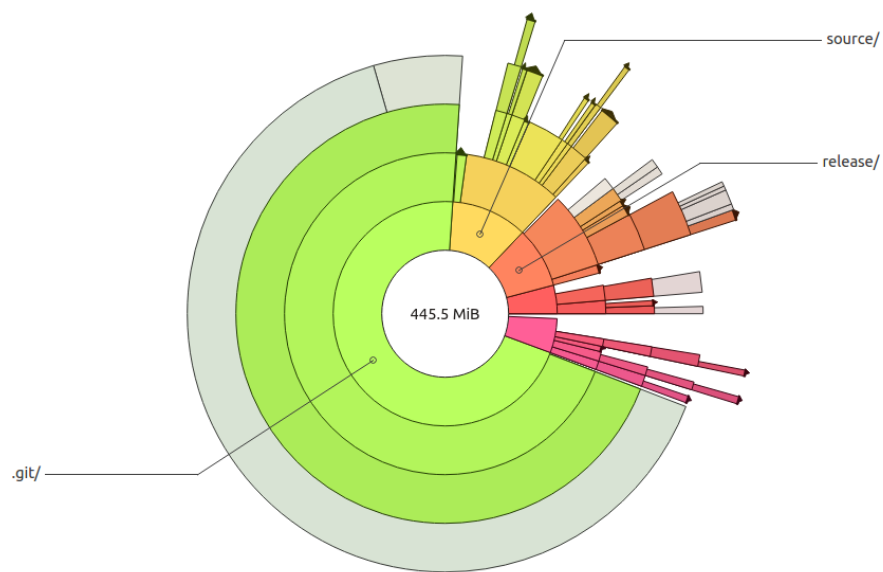


Figure 2: Filelight overview of repository

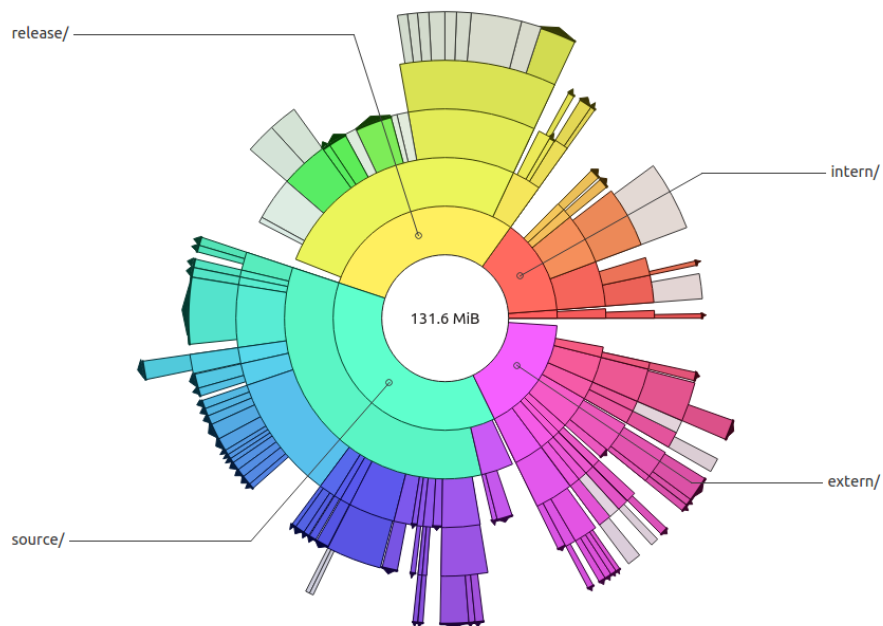


Figure 3: Filelight overview of repository without the .git folder

Folder	Size	Files
source	45.1MB (37%)	3659 (42%)
release	36.5MB (30%)	1612 (18%)
extern	20.3MB (17%)	1725 (19%)
intern	16.9MB (14%)	1280 (14%)
doc	1.2MB (1.0%)	201 (2%)
build_files	656KB (0.5%)	116 (1.3%)
tests	313.8KB (0.3%)	49 (0.5%)
scons	0B (0.0%)	0 (0%)
<b>Total:</b>	121.2MB	8654

Table 1: Distribution of files and sizes among top-level directories. The "Total" row is not so much a summation of the data in the other rows, but is the size and number of files in the entire repository, including loose top-level files.

To find out which of these top-level directories contain the most source code, we must first define which files qualify as source code and which files do not. An overview of all file extensions that occur within the repository and those of the four largest top-level directories (as defined above) are given in Table 4, Table 5, Table 6, Table 7 and Table 8. This data was gathered by running the following script inside of the directory that is to be analysed, which ignores files without extension:

Listing 5: Bash script for counting file extensions

```
#!/bin/bash

# getting all file extensions:
ftypes=$(
  find . -type f
  | grep -E "\.[a-zA-Z0-9]*$"
  | sed -e 's/\.[a-zA-Z0-9]*$/\1/'
  | sort
  | uniq
)

tab=' ' # \t character

# loop over each file extension:
for ft in $ftypes
do
  # print the file extension:
  echo -n "$ft$tab"
  # find all the files with the extension and
  # sum their sizes and count them:
  find . -name "*${ft}"
  -exec ls -l {} \;
  | awk '{amount += 1; size += $5} END {print amount "\t" size}'
done
```

We define source code as a file that contains a list of instructions which are written using some human-readable computer language, and can subsequently be compiled or interpreted using an appropriate compiler or interpreter. This definition excludes files which contain plain data, unstructured text data, image or font files, archive files, website/layout/markup data files, makefiles (since these are often automatically generated and do not influence the application after it has been released, while the actual source code affects the functionality of the application itself), configuration files, and files related to git functionality. If all of these extensions are excluded, we end up with the following list of valid source code extensions: .h (header file), .cpp (C++ source code), .c (C source code), .py (Python source code), .cc (C++ source code), .hpp (C++ header file), .rst (reStructuredText format for Python), .osl (Open Shading Language file), .glsl (OpenGL Shading Language file), .cl (Common Lisp file), .inl (Inline function files for C++), .mm (Objective C++ file), .m (Matlab source code), .hh (C++ header file), .cu (CUDA source code) and .js (JavaScript source code). From this list of file extensions combined with the amount of files as presented in Table 4, we can conclude that the Blender project is written foremostly in C, C++, Python and that shaders are included in both OpenGL Shading Language as well as Open Shading Language. The Python scripts are mostly related to Blender plugins and the Python API that is available to users and plugin developers. [5]

The commands below are used to remove all the files that are not source code:

Listing 6: Commands for deleting non-source files

```
# When variable EXT is the file extension to remove:
> find . -type f -name "$EXT" -exec rm {} +
# Removing all files without extension:
> find . -type f ! -name '*.*' -exec rm {} +
```

Figure 4 show the distribution of size of folders after the non-source files are removed, and Table 2 shows the corresponding data for each top-level directory. We can see that the "release" folder accounts for a much smaller part of the total size and files now, because it did not contain a large part of the source code. However, the "source" folder now accounts for 50% of the project, and the "extern" and "intern" folders each account for about 20%. We can conclude from this figure and table that these three folders contain the most source code, with "source" containing the majority of the source code.

Figure 5 shows the source files in the blender/source top-level directory. In "source", there are two folders that contain the majority of the source code, namely blender/source/gameengine and blender/source/blender, with the latter occupying over 85% of the entire folder. Therefore, we zoom in another level to blender/source/blender, displayed in Figure 6. This folder contains the core functionality of the Blender application. The majority of the source code files in this folder is located in the "editors" folder, followed by the "blenkernel", "makesrna", "freestyle" and "compositor" folders, as can be distilled from the figure.



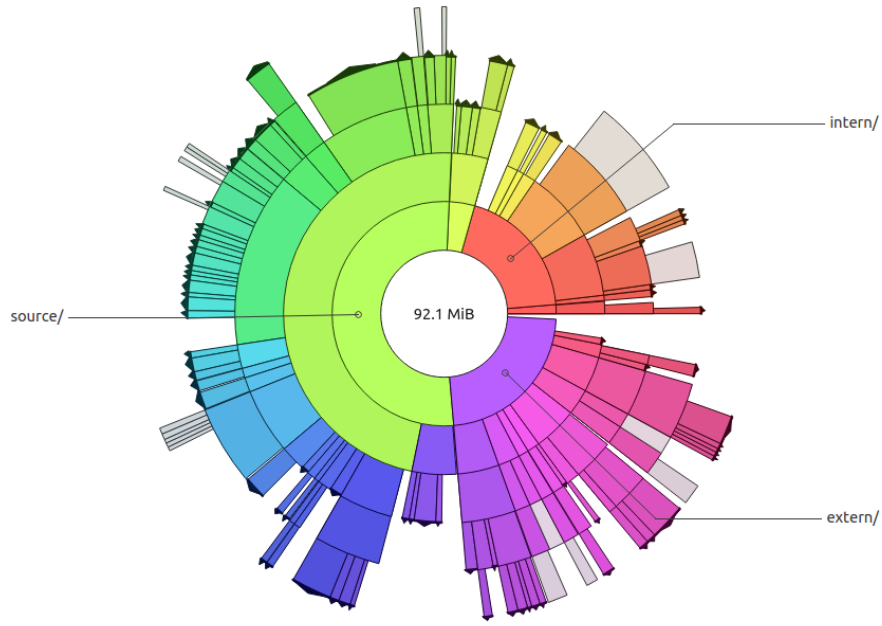


Figure 4: Filelight overview of repository without non-source files

Folder	Size	Files
source	44.3MB (52%)	3462 (50%)
extern	19.8MB (23%)	1575 (23%)
intern	16.6MB (20%)	1196 (17%)
release	2.9MB (3.4%)	443 (6.4%)
doc	844.4KB (1.0%)	177 (2.5%)
build_files	361.5KB (0.4%)	61 (0.9%)
tests	290.4KB (0.3%)	42 (6.0%)
scons	0B (0.0%)	0 (0.0%)
<b>Total:</b>	<b>85.1MB</b>	<b>6964</b>

Table 2: Distribution of files and sizes among top-level directories, including only source code files.

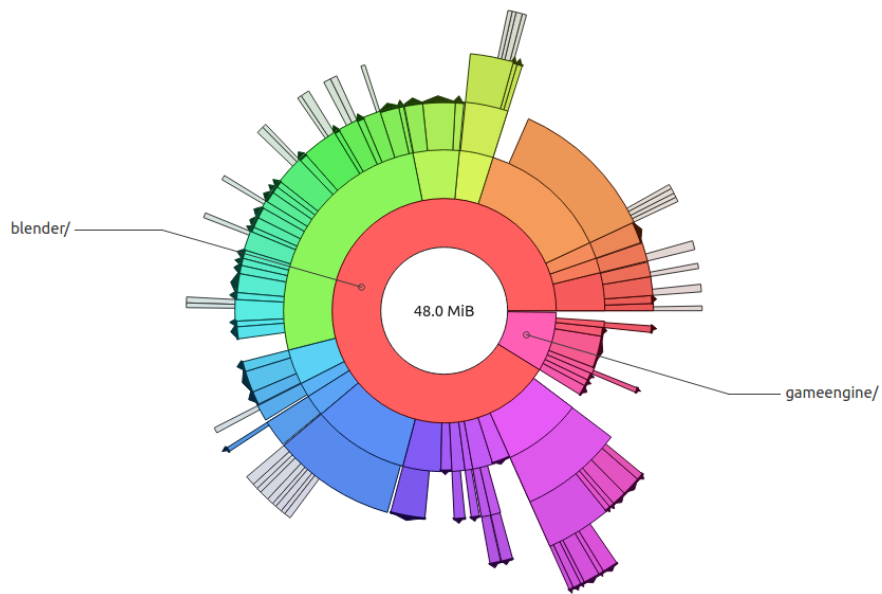


Figure 5: Filelight overview of /blender/source, without non-source files

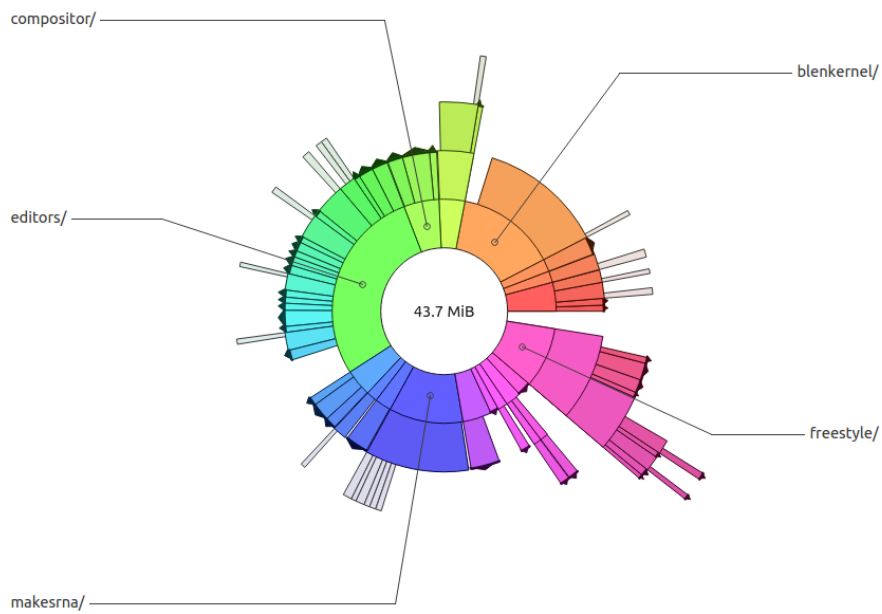


Figure 6: Filelight overview of /blender/source/blender, without non-source files

## 2.3 Developers

In this section we will describe the most active developers in the first and second half of the project, as well as show the amount of commits and developers in each half of the project, to get a better understanding of the development process of the Blender project. We define the activity of a developer in terms of the amount of commits that have been performed by that developer.

The midpoint of the project can be defined in two ways. First of all, it may describe the point before and after which there are an (nearly) equal amount of commits, so the midpoint based on the amount of commits. Secondly, it may describe the commit that happened (based on time) exactly between the first and last commit of the project. Both of these midpoints will be used and the similarities and differences of the results will be described.

To view data over a range of commits, the hash of the commits that are the extremes of the range should be known. The hash of the first and second commit can be found in Listing 3 and Listing 4 respectively. To obtain the midpoint based on the amount of commits, we simply divide the total amount of commits (61918) in two, which is the number of the commit that is the midpoint (30959th commit). Obtaining the commit is demonstrated in the following listing, which also includes the hash:

Listing 7: Fetching the commit in the middle of the project

```
> git log --skip=30959 -n 1
commit b07bdf367c8d3eb1a900ef85cec79c694d24afbe
Author: Campbell Barton <ideasman42@gmail.com>
Date: Sun May 22 05:36:11 2011 +0000

    file had non utf8 characters.
```

The date-based midpoint can be obtained by looking for a commit with a date that is exactly in the middle of the lifespan of the project. Since the project started on October 12, 2002 and lasted up to this point to November 13, 2015, the midpoint is somewhere around April 28, 2009. We manually search for a commit between April 27, 2009 and April 29, 2009 that is in the middle based on time and date, and end up with the following commit (and thus hash):

Listing 8: Fetching the date-based middle commit

```
> git log --after="2009-4-27" --before="2009-4-29"
(...)
commit 35c1b3134b60c5a2fb806fd18f50b70ee582ffe8
Author: Martin Poirier <theeth@yahoo.com>
Date: Tue Apr 28 17:55:48 2009 +0000

    etch-a-ton bugfix

    cut and trim didn't set normal properly.
(...)
```

It is clear that the date-based midpoint occurred much earlier than the commit-based midpoint. This indicates that the amount of commits in the first date-based half of the project is much lower than in the second half, so the amount of commits per time unit has increased during the lifespan of the project and develops erratically. We hypothesise this and prove this hypothesis in the next section, and can (in a way) show that this is the case here by counting the amount of commits before and after this particular commit:

Listing 9: Obtaining amount of commits in date-based halves

```
# Amount of commits in first half of project:
> git rev-list 12315f4d0e0ae993805f141f64cb8c73c5297311..35
  clb3134b60c5a2fb806fd18f50b70ee582ffe8 --count
15268
# Amount of commits in second half of project:
> git rev-list 35c1b3134b60c5a2fb806fd18f50b70ee582ffe8..46
  f452e96baaf9424582003e87736840ccbbffca --count
46649
```

The increase in commits could be caused by a larger code base, allowing and/or requiring more adjustments to the software to be made, or simply due to an increase in the amount of developers. Furthermore, software needs to evolve over time to conform to increasing user needs, which also supports the fact that the amount of activity per time unit increased as the project advanced. In the following listing, we will determine the amount of developers in each half for both approaches:

Listing 10: Counting developers in halves and entire project

```
# Get amount of developers in entire project:
> git shortlog -s -n | wc -l
236
# Get amount of developers in first commit-based half:
> git shortlog -s -n 12315f4d0e0ae993805f141f64cb8c73c5297311..
  b07bdf367c8d3eb1a900ef85cec79c694d24afbe | wc -l
106
# Get amount of developers in second commit-based half:
> git shortlog -s -n b07bdf367c8d3eb1a900ef85cec79c694d24afbe..
  46f452e96baaf9424582003e87736840ccbbffca | wc -l
180
# Get amount of developers in first date-based half:
> git shortlog -s -n 12315f4d0e0ae993805f141f64cb8c73c5297311..
  35c1b3134b60c5a2fb806fd18f50b70ee582ffe8 | wc -l
77
# Get amount of developers in second date-based half:
> git shortlog -s -n 35c1b3134b60c5a2fb806fd18f50b70ee582ffe8..
  46f452e96baaf9424582003e87736840ccbbffca | wc -l
198
```

For both approaches, the amount of developers in the second half was larger than the first half, but this is more dominantly the case for the date-based approach, most likely due to the much lower amount of commits in the first half. Note also that even though in the commit-based halves each half has the same amount of commits, the number of developers in the first half is significantly lower. This indicates that a smaller amount of developers

started the project and more developers joined as the project progresses, which seems logical for an Open Source project.

Lastly, it seems to be the case that only a minority of the developers who participated in the first halves also participated in the second halves, since the sum of the two halves does not exceed the amount of developers in the entire project greatly (286 for the commit-based halves and 275 for the date-based halves). This could indicate that there are a small amount of developers who performed the majority of the commits, and that there is a larger group of developers who only performed a small amount of commits in a certain restricted period of time within the project's lifespan, and thus not participated in both halves.

In the following two listings, the top developers (in terms of the amount of commits) are listed for both approaches, commit-based and date-based halves of the project. The first ten results are displayed for each command.

Listing 11: Obtaining activity per author in commit-based halves

```
# List amount of commits per author in first half:
> git shortlog -s -n 12315f4d0e0ae993805f141f64cb8c73c5297311..
  b07bdf367c8d3eb1a900ef85cec79c694d24afbe

6976 Campbell Barton
4415 Ton Roosendaal
2172 Brecht Van Lommel
2047 Joshua Leung
1185 Martin Poirier
1145 Matt Ebb
1012 Nathan Letwory
 597 Kent Mein
 516 Nicholas Bishop
 501 Daniel Genrich
(...) (...)

# List amount of commits per author in second half:
> git shortlog -s -n b07bdf367c8d3eb1a900ef85cec79c694d24afbe..
  46f452e96baaf9424582003e87736840ccbbffca

11711 Campbell Barton
4881 Sergey Sharybin
2368 Brecht Van Lommel
1910 Bastien Montagne
1260 Antony Riakiotakis
1124 Thomas Dinges
 996 Tamito Kajiyama
 827 Joshua Leung
 493 Lukas Toenne
 450 Joseph Eagar
(...) (...)
```

### Listing 12: Obtaining activity per author in date-based halves

```
# List amount of commits per author in first half:
> git shortlog -s -n 12315f4d0e0ae993805f141f64cb8c73c5297311..
  35c1b3134b60c5a2fb806fd18f50b70ee582ffe8

3437 Ton Roosendaal
2214 Campbell Barton
 761 Brecht Van Lommel
 723 Joshua Leung
 686 Martin Poirier
 546 Kent Mein
 440 Ken Hughes
 399 Daniel Dunbar
 366 Matt Ebb
 364 Nathan Letwory
(...) (...)

# List amount of commits per author in second half:
> git shortlog -s -n 35c1b3134b60c5a2fb806fd18f50b70ee582ffe8..
  46f452e96baaf9424582003e87736840ccbffca

16473 Campbell Barton
 5150 Sergey Sharybin
 3779 Brecht Van Lommel
 2151 Joshua Leung
 1910 Bastien Montagne
 1531 Thomas Dinges
 1392 Ton Roosendaal
 1260 Antony Riakiotakis
  996 Tamito Kajiyama
  783 Matt Ebb
(...) (...)
```

The top three developers, as requested by the assignment, can be found in the lists for each approach. The results of both approaches is nearly the same, with the exception of the order of Ton Roosendaal and Campbell Barton. In the commit-based approach, Ton Roosendaal is listed as the second top developer, while in the time-based approach he is listed as the first. Since the time-based first half is concentrated earlier on in the project than the commit-based first half (2002 to 2009 and 2002 to 2011 respectively), this indicates Ton Roosendaal did most of his commits in the earliest phases of the project, and that Campbell Barton's efforts increased substantially in the period of 2009 to 2011.

We can also conclude from the lists that Ton Roosendaal put most of his effort before 2011, since he is not listed as a top developer in the second commit-based half.

### 3 Initial Visual Overview

In this section, more advanced analyses will be performed on the repository using the Solid Trend Analyzer (SolidTA) tool, to determine the stable and unstable development periods of the project, as well as to come to a conclusion about the current state of the project.

#### 3.1 Choice of views

To find the stable and unstable development periods, we must first define what is a stable and what is an unstable development period. A development period is stable when the code base does not change much (based on amount of commits/file accesses) or grow significantly (based on code size metric and number of files). On the other hand, a development period is unstable when the code base does change or grow significantly.

The information that we need to determine the stable and unstable development periods are related to the total code size, the total amount of files in the project, the amount of commits per time unit and possibly the amount of file accesses, all of which are measured for each time unit in the project's time line. In Figure 10, the four views that we have chosen are presented. The top view is obtained by turning on the code size metric (which is not necessary for the top view, but was used in the second and third) on the time view which shows all of the files in the project, and then sorting the files based on creation time. The second view is the evolution view of the code size metric for the entire project over time. The third view is again an evolution view of the code size metric, this time displaying the amount of commits for all files which have a value for the code size metric. The fourth view, which shows the amount of file accesses (categorised by author, although this is not relevant in this step per se), was obtained by showing the evolution view of the authors metric over all files.

Some other views were considered for this step. In Figure 7, the Lines of Text (LOT) metric is displayed for all files which have a value for this metric, sorted on creation time. We were considering to use this view instead of first code size view, since it also gives a measure of the size of the files, and since the LOT metric is applicable on more files than the code size metric (due to also applying on non-source code), we thought that this could be relevant for this analysis. It should be noted that a stable code size metric does not imply no development has taken place, since changes could be made without changing the amount of LOC (Lines of Code) in a file, so other views are still needed. Similarly, we considered in Figure 8 to display only the files which have a value for the code size metric, sorted on creation time, for the same reasoning. For both figures, we decided in the end that it was important to consider all files in the project when determining the stable and unstable development periods, since files that are not source code or could not be counted in LOT could still require development efforts to integrate into the project (think configuration files, data archives and formats which are used during the execution of Blender that have to be accounted for). Finally, we also admitted that the corresponding evolution views of these figures did not show any significant differences in structure.

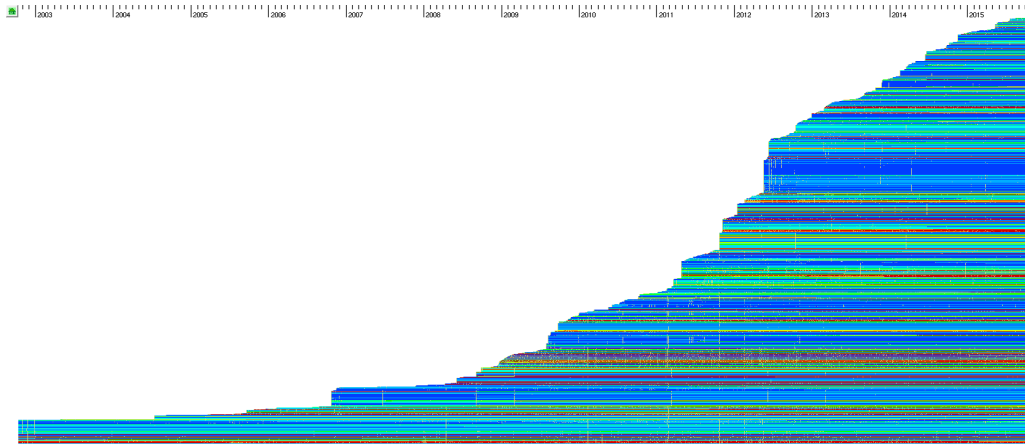


Figure 7: View of all files that have a value for the LOT (Lines of Text) metric, sorted by creation time. (Rejected)

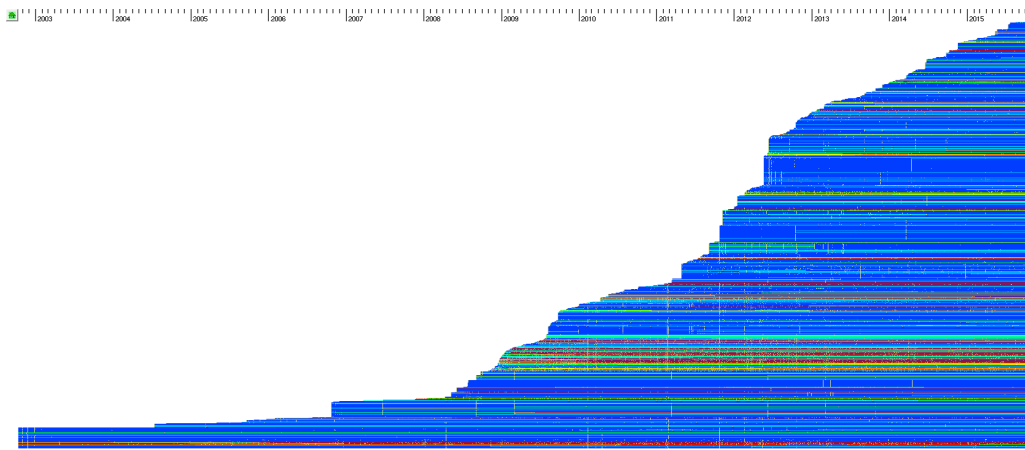


Figure 8: View of all files that have a value for the code size / LOC (Lines of Code) metric, sorted by creation time. (Rejected)

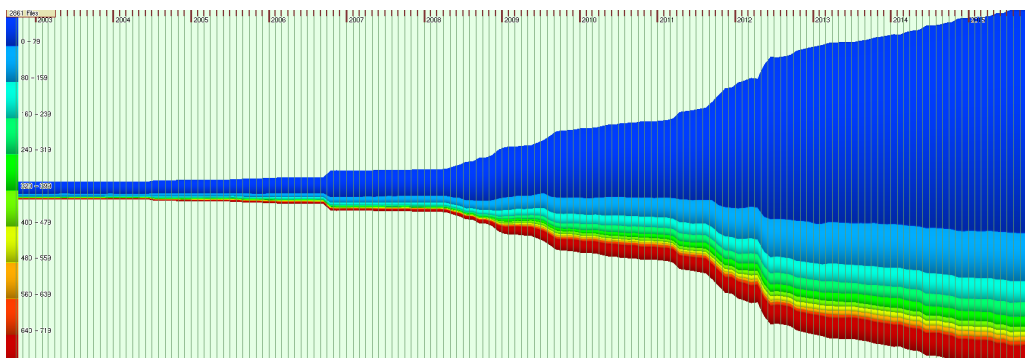


Figure 9: Trend view of the total amount of files with a value for the code size / LOC (Lines of Code) metric. (Rejected)



Lastly, we considered to also show the file count of the entire project over time instead of the total amount of file accesses, displayed in Figure 9. However, since the first view of the chosen views already depicts the growth of files over time, and the fact that this figure could only display the filecount for files that had a code size metric value, the figure turned out to be redundant. Instead, we now show the amount of file accesses to all files within the project, which is closely related to the amount of commits (the third view), although that figure only shows the amount of commits for files that have a code size metric.

Having determined the views and methodology that will be used to define stable and unstable periods, we will now determine these periods and the current state of the project in the next sections.

### 3.2 Stable development periods

In Figure 11, the stable periods of the project are depicted in green. We define these periods as G1, G2, ..., G7, where the number indicates the period read from left to right.

These periods were first identified simply by looking at the slope of the first view, which indicate the degree to which new files are created. If the slope is steep, many new files are created per time unit. If it is rather flat, it means that the amount of files stayed roughly the same. When the growth slope of files is relatively low, or at least stable over a period of time, we hypothesise that these periods are stable. We can verify this hypothesis by looking at the second view, which indicates the total code size of the project. Similarly to the first view, the code size metric shows stable (and/or low) growth during the periods that we have defined, but on its own this does not imply a little amount of development as pointed out earlier in this section. To further establish the fact that these periods are indeed indicated to be stable, the third and fourth view show a reduced amount of commits and file accesses for these periods opposed to the time before and after these periods.

### 3.3 Unstable development periods

In Figure 12, the unstable periods of the project are depicted in red. We define these periods as R1, R2, ..., R6, where the number indicates the period read from left to right.

Opposed to the stable periods, we identified these periods by pointing out the areas where the slope of the first view is quite high relative to the surrounding areas, which is the hypothesis for unstable periods. This hypothesis can again be verified by comparing the first view to the other three views that are given. In the corresponding periods, we see either an increase in commits and file accesses and/or a (sharp) increase or irregular development of the code size metric.

In R6, we notice a steep increase in the amount of files for the project, however, these files are not source code files, but .dat files that appear to be icons for the interface of part of the application. Regardless, this is considered an unstable period, due to the spikes in commit count and file accesses that can be viewed in the third and fourth view, which is supported by a slight spike in the code size metric in the second view as well.

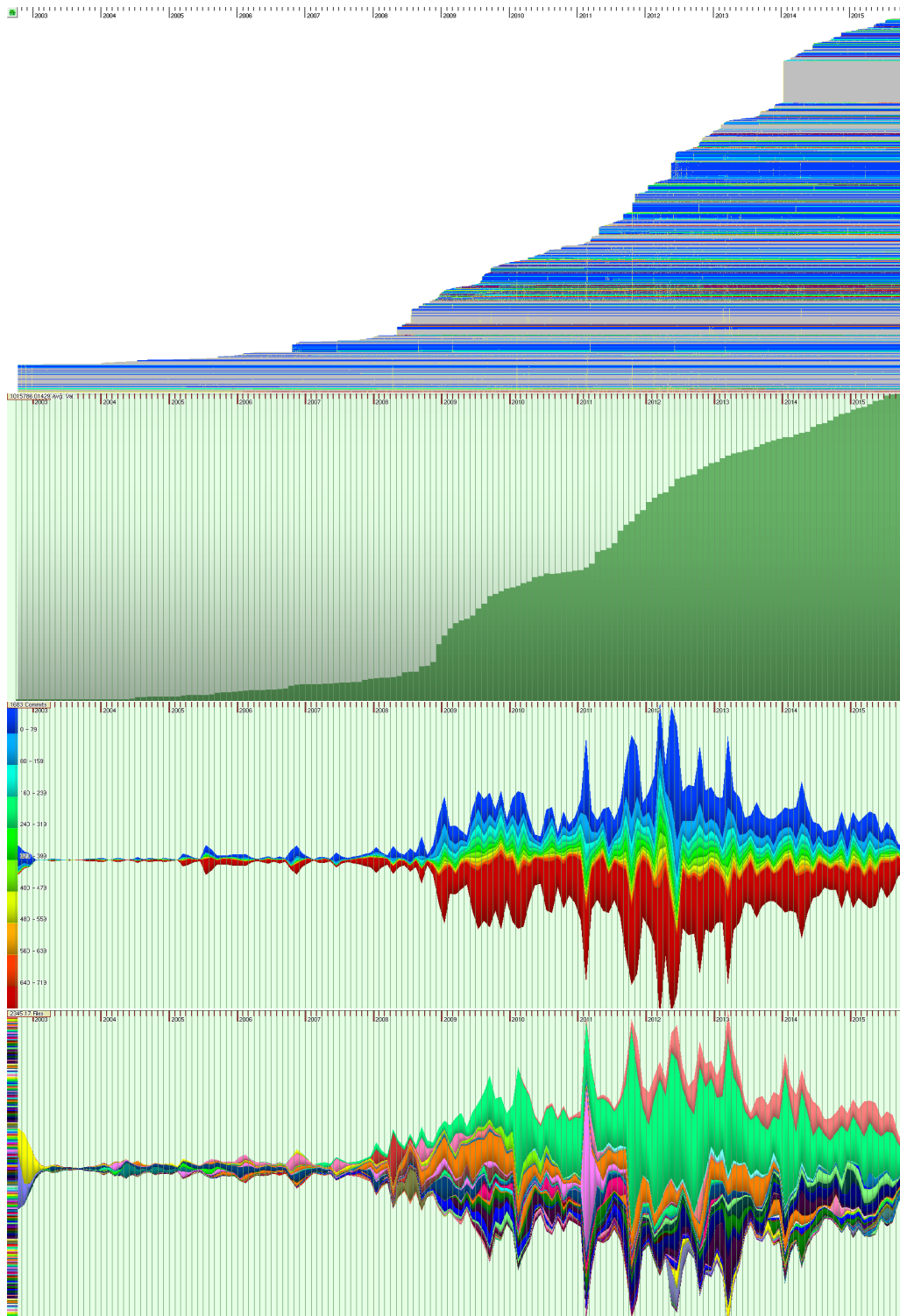


Figure 10: Views used for determining stable and unstable development periods. From top to bottom: View of all files (including the code size metric) sorted on creation time, trend view of total code size metric, trend view of number of commits over files with code size metric, trend view of file accesses (i.e. where changes have been made) of all developers over all files.

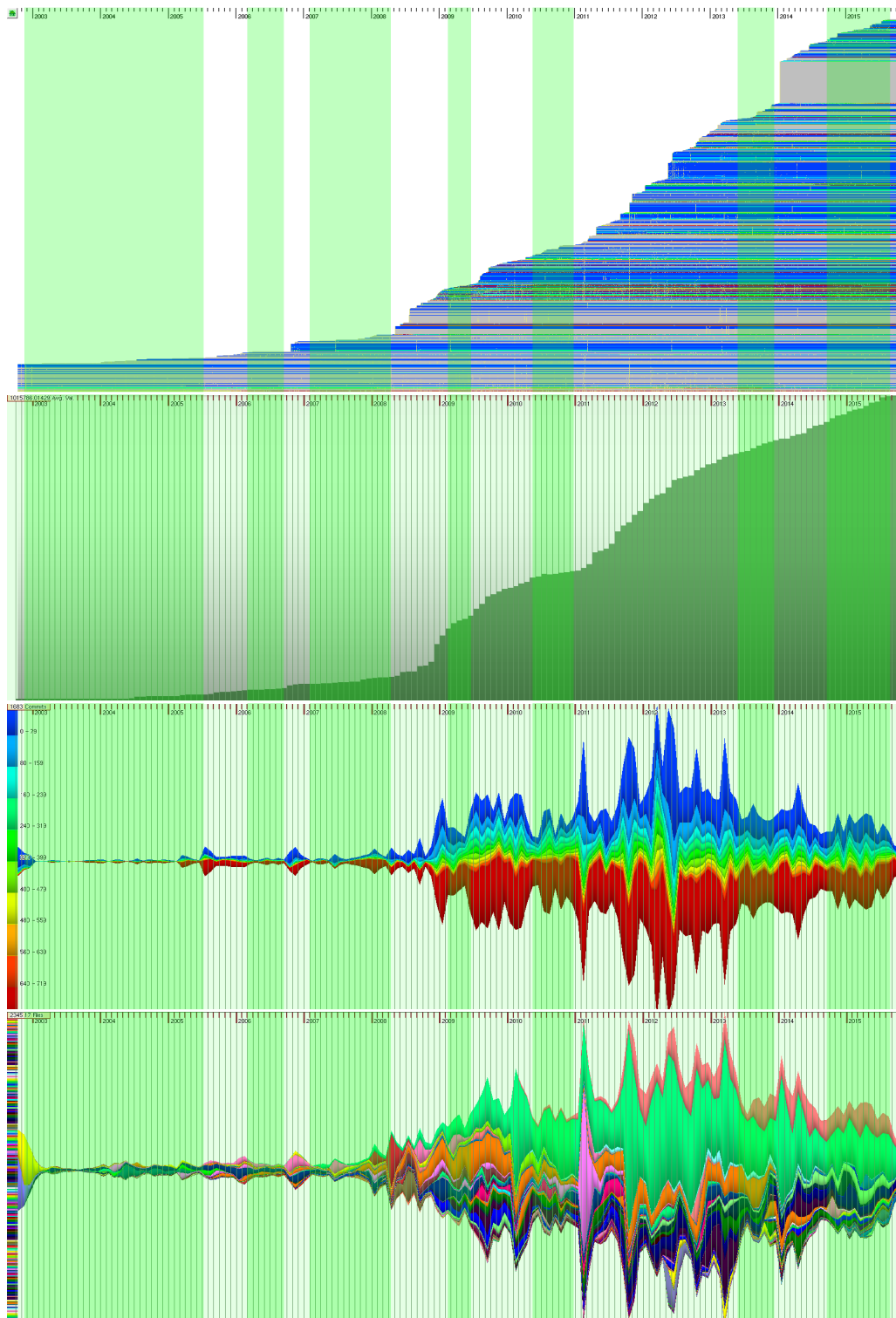


Figure 11: Stable development periods (depicted in green)

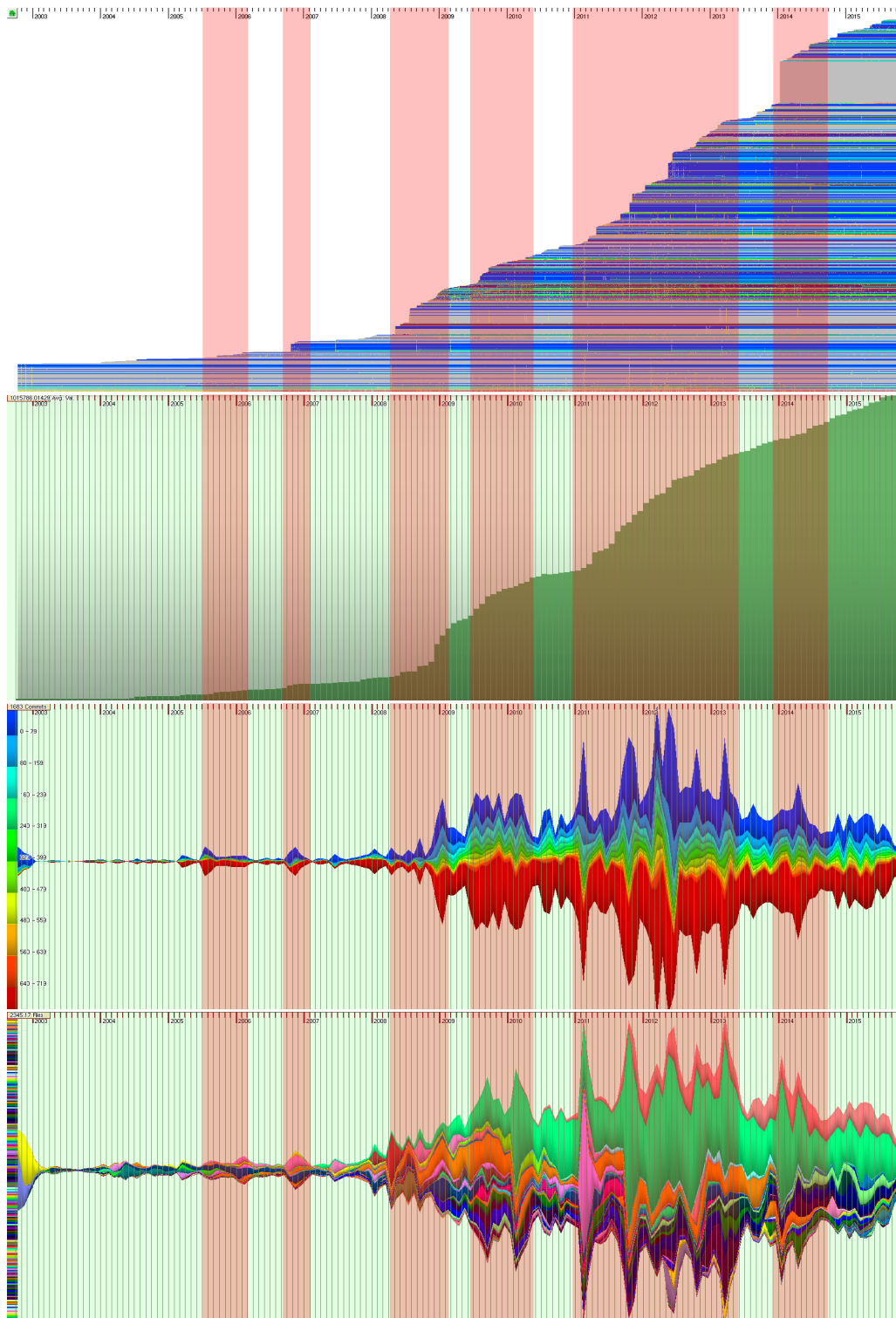


Figure 12: Unstable development periods (depicted in red)

### 3.4 Current state

The current state of the project, which we define to be the period of the last 3 months of the project, is relatively stable. The slope of the first view seems to have stabilised since the start of G7, similarly to the slope of the code size metric. Furthermore, the amount of changes performed in the last year is very regular and has decreased steadily in the past 3 months, which indicate a stable state. Furthermore, the amount of file accesses also do not indicate any rigorous changes to the code base either. Concluding these metrics, we claim that G7 is still in process and persists until the time of this analysis.

To further support this claim, we give some circumstantial information about the Blender project. Version 2.76b was scheduled to release on November 3, 2015, [7] so it makes sense that we are seeing a decline in the amount of file accesses and commits performed to the project, as well as a low growth in files and code size most recently in this data. This is due to the fact that the Blender release cycle describes that the last 1.5-2 months of each cycle are for stabilisation, bug fixes and code review. [8] We see similar anti-spikes of commits and file accesses in other places in G7. (e.g. February/May 2015) Version 2.76b was also a "bug-fix" and thus a secondary version, which inherently means that the amount of commits will be lower due to less functionality being added opposed to bug fixes and code review.

## 4 Author Analysis

In this step, we will perform several analyses that have to do with the developers/authors (terms used interchangeably in this section) of the Blender project and their revisions that they have performed on the various files of the project.

### 4.1 Main contributors

In Figure 13, the file view in SolidTA of all files of the project is given. The view is sorted on the creation time of the files and the authors metric was enabled, which will show the revisions that have been performed on the files per author in different colours. The most interesting authors for the analyses are those that have relatively performed a large amount of revisions. To find the main contributors, we perform the options "Show # Versions" and subsequently "Sort On # Versions" in the legend of authors in SolidTA. This sorts the authors based on the amount of revisions that they have performed. The top 20 authors in this list are given in Figure 14, and these authors are considered to be the main contributors of the project.

The legend indicates that there are a small amount of developers who performed the majority of the revisions, because the amount of revisions seems to develop exponentially towards the top developer. (Campbell Barton) We hypothesise that Pareto's principle applies, which in this context would claim that 20% of the developers performed 80% of the revisions. In Figure 15, we have selected the top 20% (around 45) developers based on the sorting of the legend performed by SolidTA. When no author is selected for a revision, it would appear in grey, and since there is no visible difference (no grey is visible) when comparing Figure 13 and Figure 15, we can conclude that the data indicates that Pareto's principle is indeed in effect for the repository.

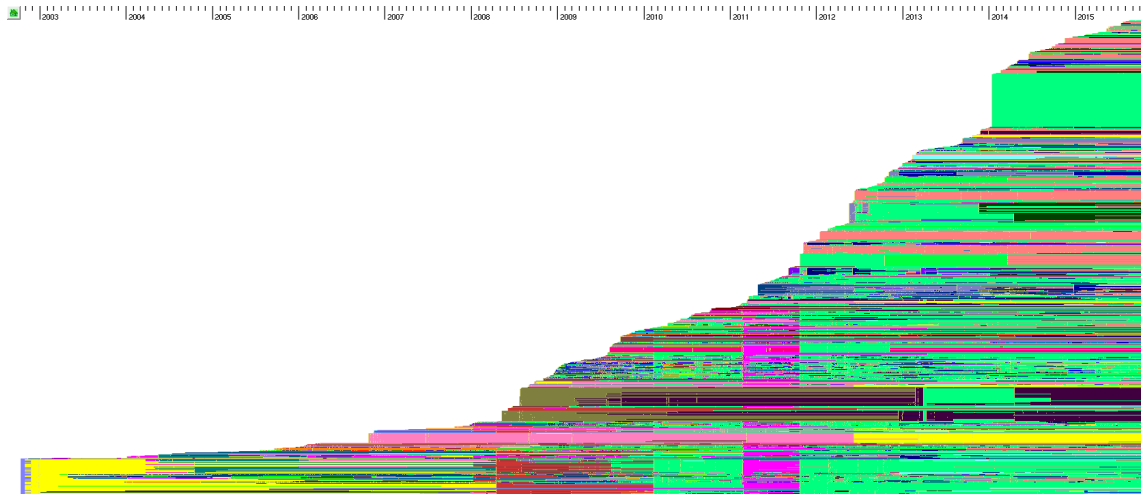


Figure 13: View of all files sorted on creation time including the authors metric in SolidTA. See Figure 14 for legend of authors.



Figure 14: Legend of top 20 authors in entire repository, the blue bars in the background indicate the amount of revisions for each author.

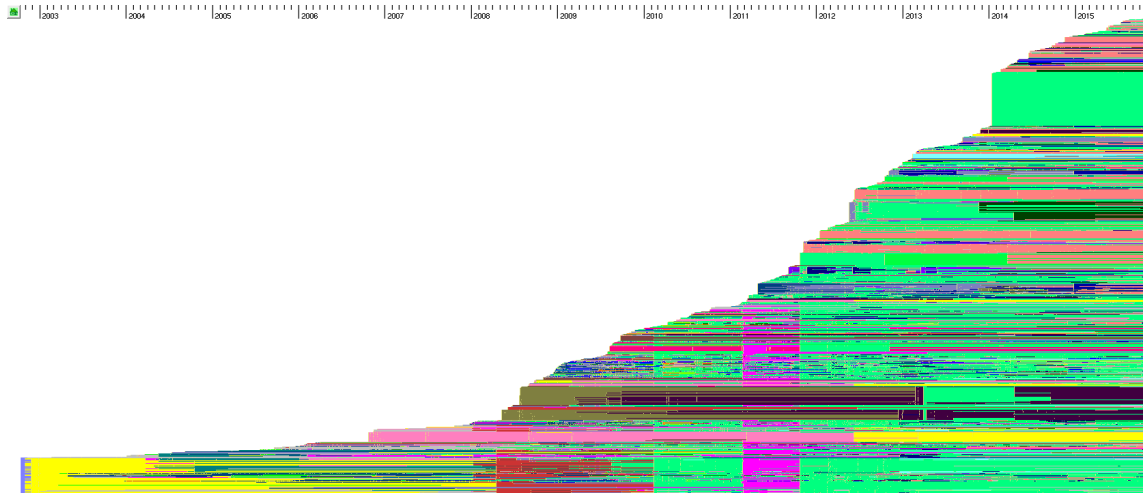


Figure 15: Viewing the contributions of the top 20% ( $\approx 45$ ) authors for the entire repository. See Figure 14 for legend of authors.

Figure 13 indicates that Campbell Barton could be the chief developer, since the figure shows a large surface area of his corresponding colour in the legend. However, this could also be an effect of certain files not being edited for a long time, which leads to a larger surface area even though there is not much activity being performed in these files. Therefore, we need to consult different views to come to a conclusion about who is the chief developer of the project. In Figure 16, Figure 17 and Figure 18, the trend view of the file count per author is given for respectively all authors, the top 20 authors (described in Figure 14) and the top 10 authors.

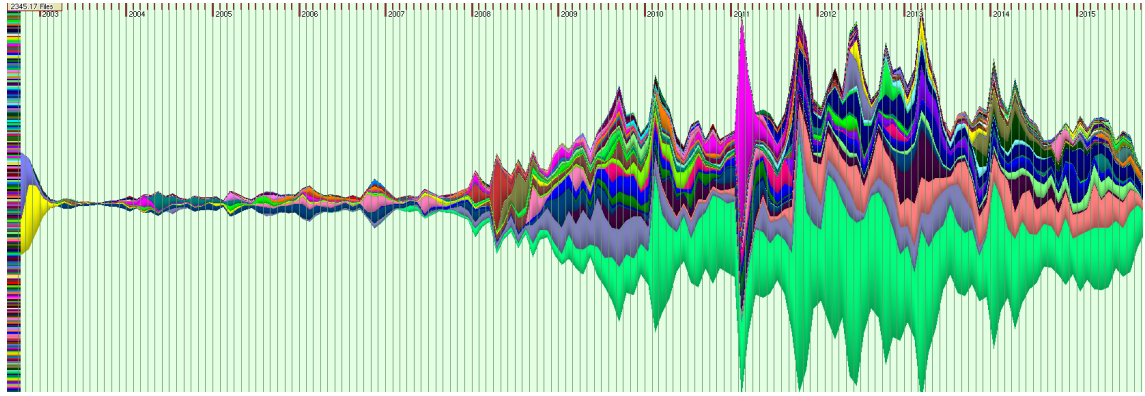


Figure 16: Evolution view of the file count of all authors over all files in SolidTA. See Figure 14 for legend of authors.

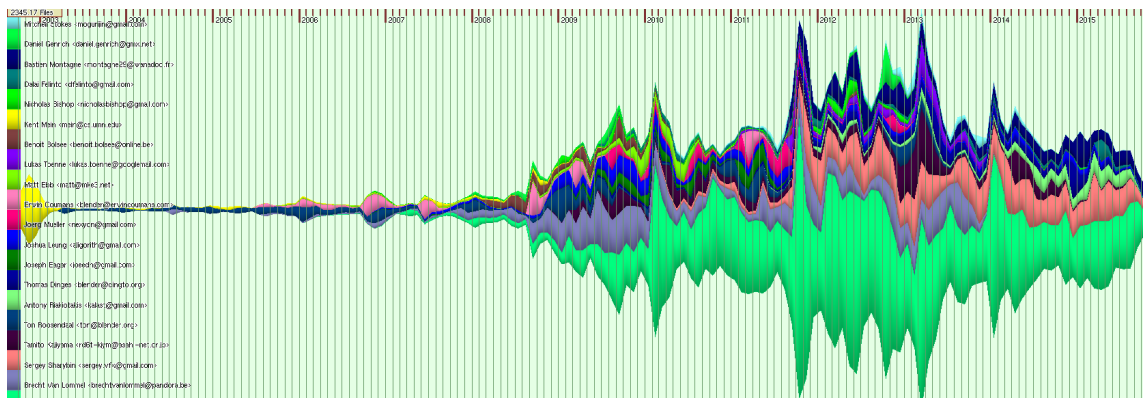


Figure 17: Evolution view of the file count of the top 20 authors over all files in SolidTA. See Figure 14 for legend of authors.

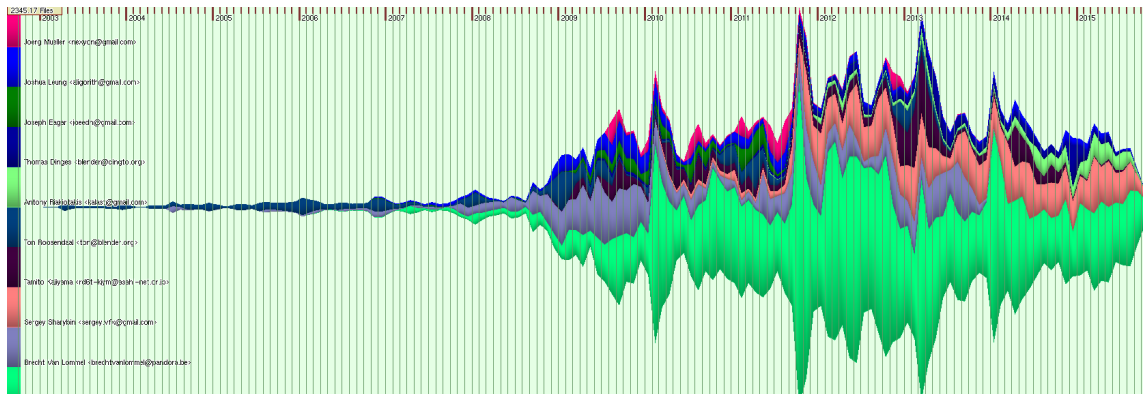


Figure 18: Evolution view of the file count of the top 10 authors over all files in SolidTA. See Figure 14 for legend of authors.



We define a developer as a chief developer for a certain time period, when they have performed a significant and consistent number of revisions relative to the other authors in that time period. The reason why we have provided three trend views of the file count per author, is the fact that chief developers at the start of the project may not be represented in the top authors anymore, so they would be obscured from a view that only listed the top authors. Therefore, a view of all authors was included in Figure 16. The view shows that Kent Mein, depicted in yellow, and Hans Lambermont, depicted in lavender, have performed the initial commits and development of the repository. Kent Mein can be described as the chief developer of the period from the start of the project to around April 2003. From this point onwards, Toon Roosendaal is the person with the most revisions and a consistent contribution to the repository. Ton Roosendaal can be appointed as chief developer of the period that ranges from April 2003 to February 2007, at which point Campbell Barton joins the project.

Campbell Barton subsequently has a very high and consistent activity based on the number of file revisions compared to other authors, often performing over half of the total amount of revisions per month. We also showed in the *Basic Repository Investigation* that Campbell Barton was by far the developer with the most commits in the repository, and this is also clear from Figure 14. Therefore, we consider Campbell Barton to be the chief developer of the Blender project from February 2007 up to this point. To further support our claim that Campbell Barton is indeed the current chief developer of the project, we excluded him from the evolution view in Figure 13, and obtained Figure 19, which clearly shows a large amount of grey revisions in a large amount of the older and newer files of the project.

It should be noted that in the list of developers, there seem to be some developer accounts who are actually the same person. (e.g. "Gaia Clary" vs "gaiaclary" listed under the same email address) In the worst case, namely for Brecht van Lommel (a top 20 developer), there were five accounts listed that belonged to the same person. The fact that the authors list contains duplicates could potentially throw off the process of determining the top contributors and the chief developer. However, we have investigated the accounts and define a single primary (the account with most revisions) and one or more secondary (less revisions than primary) accounts. In all cases, either the sum of secondary revisions did not weigh up against the primary revisions and would not have changed the corresponding developer's rank, or the total of primary and secondary was so low that it would not have made it even close to the top of the authors list, so there is no indication that this will affect the integrity of our analyses regarding the authors.

## 4.2 Replacing chief developer

Campbell Barton has been determined to be the current chief developer of the Blender project, as he consistently performs a majority of the revisions on the project which are located in virtually any location in the repository, as shown by Figure 13. It would be difficult for the project to continue its release cycle if Campbell Barton were to quit the project, as shown by Figure 19, since his presence in the project covers many old and new files. Furthermore, the amount of revisions would drastically decline, as presented in Figure 20. Nevertheless, if Campbell Barton were to leave the project, the files that this chief developer was most active in will have to be maintained by another developer, so a new chief developer or multiple chief developers will have to be chosen.

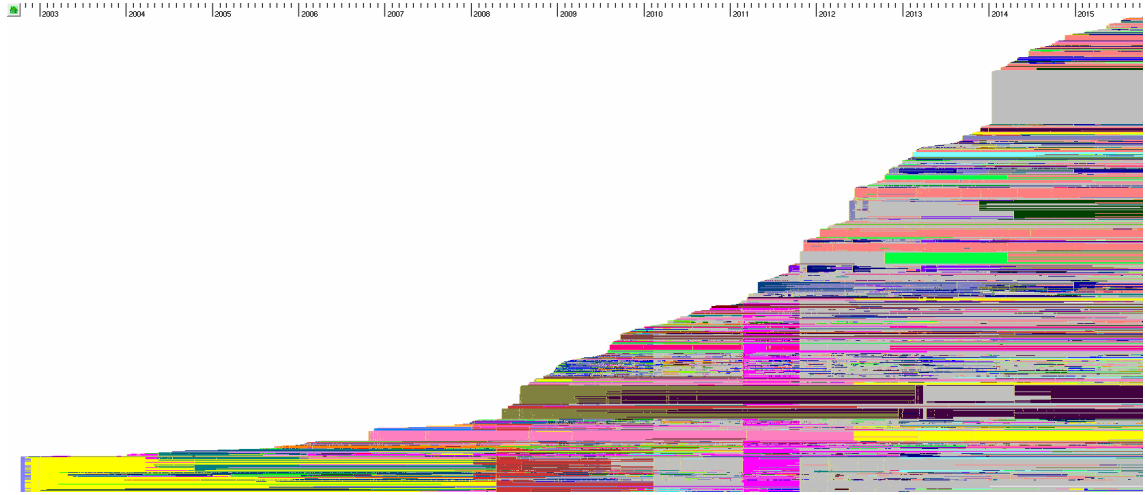


Figure 19: The same as Figure 13, excluding the revisions of Campbell Barton (displayed in grey).

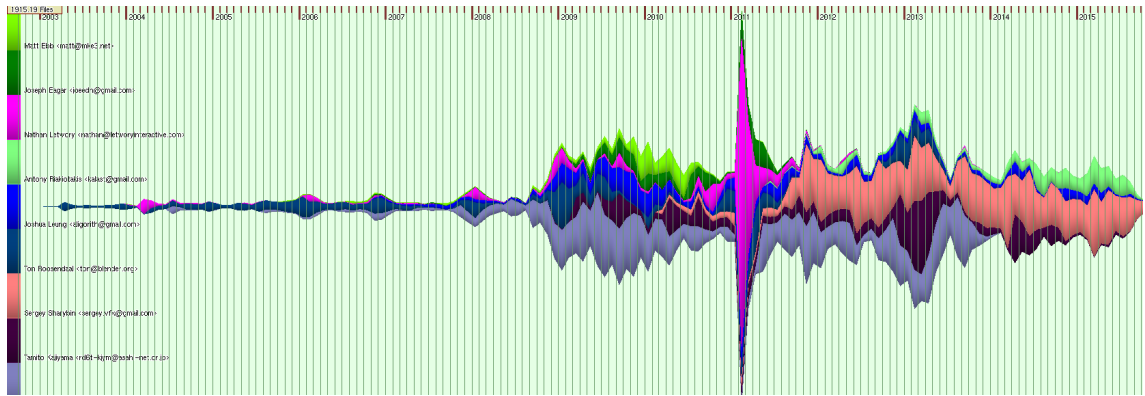


Figure 20: The same as Figure 18, excluding the revisions of Campbell Barton.

A developer is a suitable candidate for being the chief developer if they have performed revisions in the same files that Campbell Barton has edited and if they are currently active within the project (based on the amount of revisions). The SolidTA legend for authors is biased, in a sense that it will not show developers in the top of the list who have performed the most amount of commits in the last year, but rather in the entire project. Therefore, we will refer to [9], which accurately lists the top developers for 2015 so far, and the list is different from the list of top developers mentioned in Figure 14.

In Figure 21, the last two years of Figure 13 are presented, which will be used to make claims about the current activity of developers. Not all authors with a significant revision share in this period are displayed in previously mentioned legends, but through SolidTA we can see that Sergey Sharybin, Bastien Montagne, Lukas Toenne, Antony Riakiotakis and Joshua Leung have performed revisions on the same files as Campbell Barton in this time period. Furthermore, also based on 18, we consider Brecht van Lommel, Tamito Kajiyama and Ton Roosendaal as candidates. Section 11.3 in the Appendix of this document lists the revisions in the file view for all files sorted on creation time, for all these authors that were considered for taking over the work of Campbell Barton.

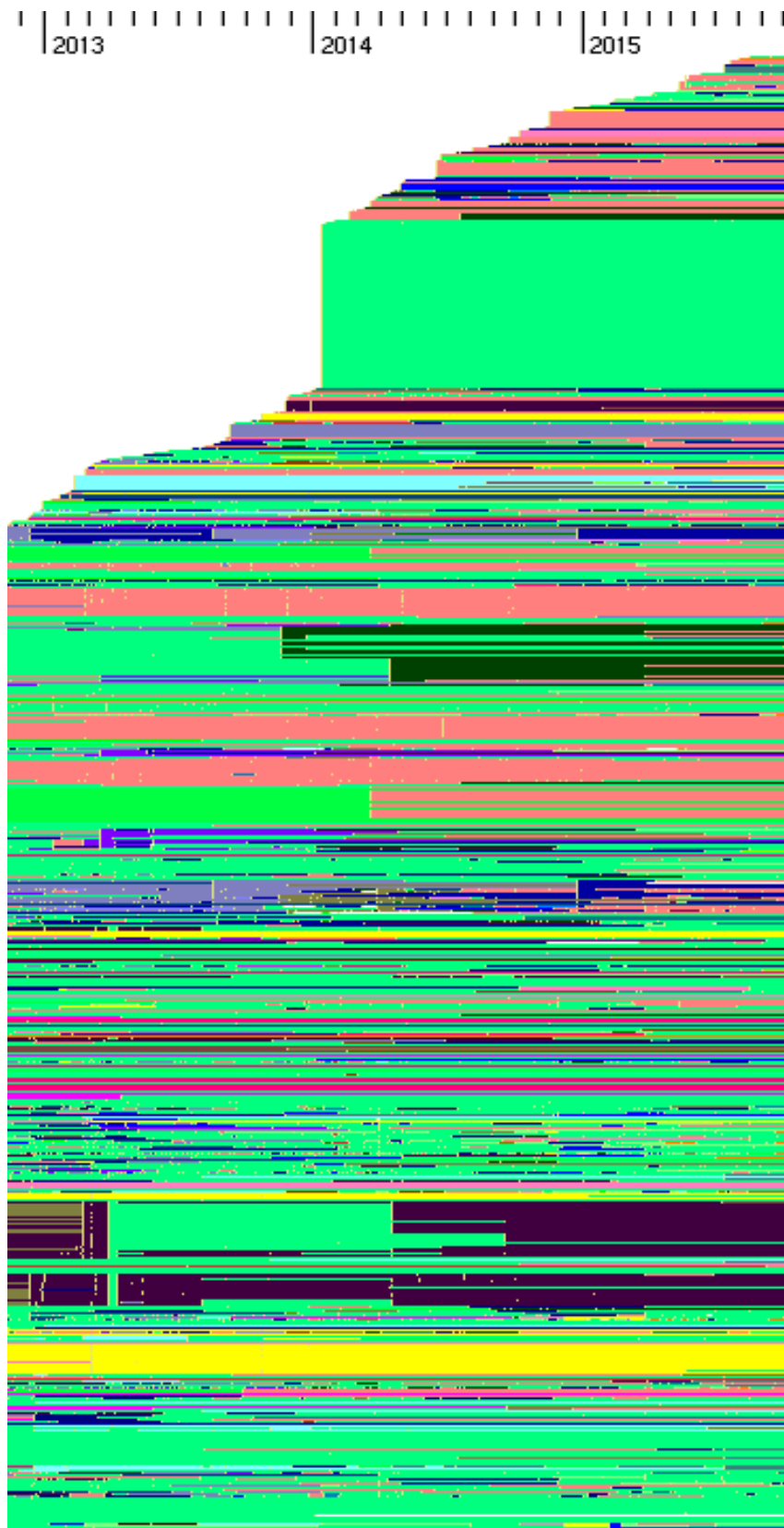


Figure 21: Detail of the last two years of Figure 13.

Ton Roosendaal was picked for his extensive revisions early on in the project on the older files. However, he has not been that active in the past two years. Brecht van Lommel has performed revisions in a large variety of locations in the repository which means they potentially have knowledge of large parts of the application, but she has also not been active as much recently. Tamito Kajiyama performed a large amount of revisions in earlier years on files that Campbell Barton also worked on, however these revisions are mostly focused on a single directory and recently his activity has dropped as well.

Therefore, Sergey Sharybin, Bastien Montagne, Lukas Toenne, Antony Riakiotakis and Joshua Leung remain as candidates to take over the work of Campbell Barton. Joshua Leung and Antony Riakiotakis have performed revisions on a limited amount of locations in the repository, so these two are not very suited for becoming chief developer. Lukas Toenne's (who has two accounts that have both been shown for the sake of completeness) revisions are also congregated mostly, but he has also performed revisions in other parts of the repository. Sergey Sharybin in particular and also Bastien Montagne are very suitable for becoming chief developer due to their sheer amount of file revisions to a large variety of file locations in the repository. Therefore, a subset of Lukas Toenne, Sergey Sharybin and Bastien Montagne would be most suited for taking over the work of Campbell Barton. Sergey Sharybin would be the most suitable of the three, due to the fact that he is still active, has a large fraction of the total revisions based on 18 (which are also performed on Campbell Barton's files), and has been a significant development member for many years.

### 4.3 Type correlation

We will now analyse if there is a correlation between the developers and the types of files that they worked on. To investigate this, we enable the file type metric in SolidTA, and show the amount of files per type in the legend. Figure 22 shows the contents of this legend after this action was performed. Subsequently, the file view of all project files was first sorted on creation time, and then sorted on file type while this metric was enabled, obtaining Figure 23.



Figure 22: Legend of file types

In this figure, the files are grouped together based on their type. Due to the fact that .cc, .c and .cpp files have seemingly the same color, we will note here that these groups are displayed from top to bottom as: .c, .cc, .cpp. The rest of the file groups can be related to the legend based on their colour. Now, we turn off the file type metric and turn on the authors metric, and we obtain Figure 24, in which we can overlay our knowledge of Figure 23 to find out which developers worked on which files. (See also Figure 14)

From the figures, we can tell that Campbell Barton is responsible for most of the revisions performed on .c files, and nearly the entirety of .dat, .cmake and .png files. (See Section 11.4 in the Appendix) This indicates that Campbell Barton is the chief of these files. Furthermore, Sergey Sharybin seems to be mostly responsible for the .cc files of the project and the majority of .hpp files. Ton Roosendaal performed all revisions on .jpg files, while Brecht van Lommel and Mitchel Stokes performed most of the revisions on respectively .spild and .rst files, and Brecht van Lommel and Thomas Dinges are together responsible for .osl files. (See Section 11.4 in the Appendix) For all other files, there is no indication of a clear correlation between developers and types of files that they worked on.

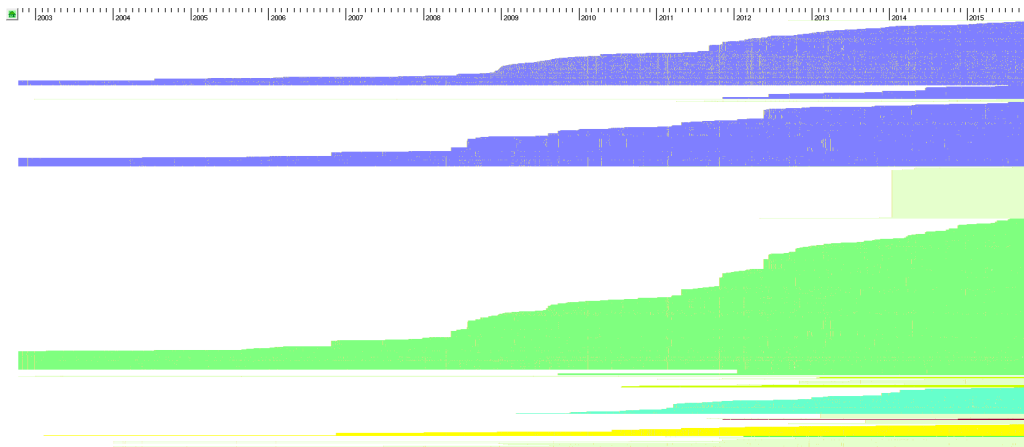


Figure 23: File view of all files, sorted by creation time and subsequently by type, with the file type metric enabled. See the legend in Figure 22.

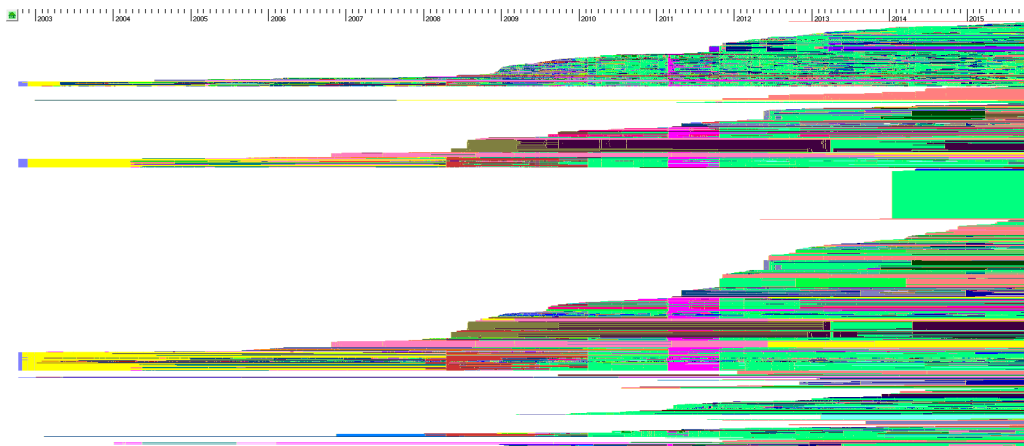


Figure 24: The same as Figure 23, replacing the file type metric for the authors metric.

## 4.4 Location correlation

Finally, we will analyse if there is a correlation between the developers and the locations of files that they worked on. We enable the folder metric in SolidTA, and put the configuration slider on level 2, so that only top-level directories are displayed. The legend for this metric is depicted in Figure 25. We sort the file view of all files first by creation time and then alphabetically, which generates Figure 26, in which we can see that the files are grouped together by the folder that they belong to. Similarly to the previous analysis, we now turn off the folder metric and turn on the authors metric, so that we can see the location in which the developers performed their revisions, depicted in Figure 27.

For the top-level directories, we see that Mitchel Stoke is indicated to be the chief developer of the `/doc` folder, in particular the `/doc/python_api/rst` folder. This makes sense, since this folder contains all `.rst` files, of which he was also the chief developer. Sergey Sharybin seems to have performed the most revisions in the `/extern` folder (particularly in `/extern/libmv` and `/extern/Eigen3` sub-directories), next to Erwin Coumans and Sergej Reich (in yellow) (particularly in the `/extern/bullet2` folder sub-directories). For `/intern`, there are two big sub-directories that are the responsibility of one or two developers. The `/intern/audaspace` folder is mostly managed by Joerg Mueller and `/intern/cycles` by Brecht van Lommel and Thomas Dinges. Campbell Barton seems very present in this directory, but his revisions are mostly related to cleanup and style of code. The release folder contains mostly image and data files which are per sub-directory indeed uploaded by a single author, but these are many smaller folders that are not worth discussing. Campbell Barton uploaded many `.dat` files, but these are data files as well. `/release/scripts` contains Python scripts that are managed by a large variety of developers.

Now, only the `/source` folder is left to analyse. Since the `/source` folder is the folder that contains most of the source code files of the project, we decided to highlight this folder by zooming in on `/source/blender` and performing the same approach as we did for the top-level directories. The (shortened) folder and author legend are given in Figure 28, the folder view in Figure 29, and the corresponding authors view in Figure 30. The configuration slider this time is set in such a way that the various sub-directories of `/blender/source` can be distinguished, so the fourth level.

From these figures, we can deduce some correlations between developers and folders. First of all, Campbell Barton seems to be present everywhere in this folder, and over the entire folder he could be seen as the chief developer, similarly as to the entire project. Tamito Kajiyama seems to be the chief developer of the `/source/blender/freestyle` folder, as he performed most of his revisions in this folder and has the majority of the revisions there. In the `/source/blender/nodes` folder, Lukas Toenne appears to be the chief developer, with Thomas Dinges as second chief developer. Finally, Lukas Toenne has performed the last revisions in the `/source/blender/compositor` folder, but these are mostly related to cleanup.

We can conclude that, for a limited amount of folders in the repository, it is indeed the case that a chief developer can be distinguished, so there is a correlation between developers and file locations.

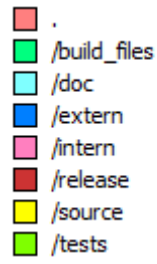


Figure 25: Legend for the folder metric (top-level directories)

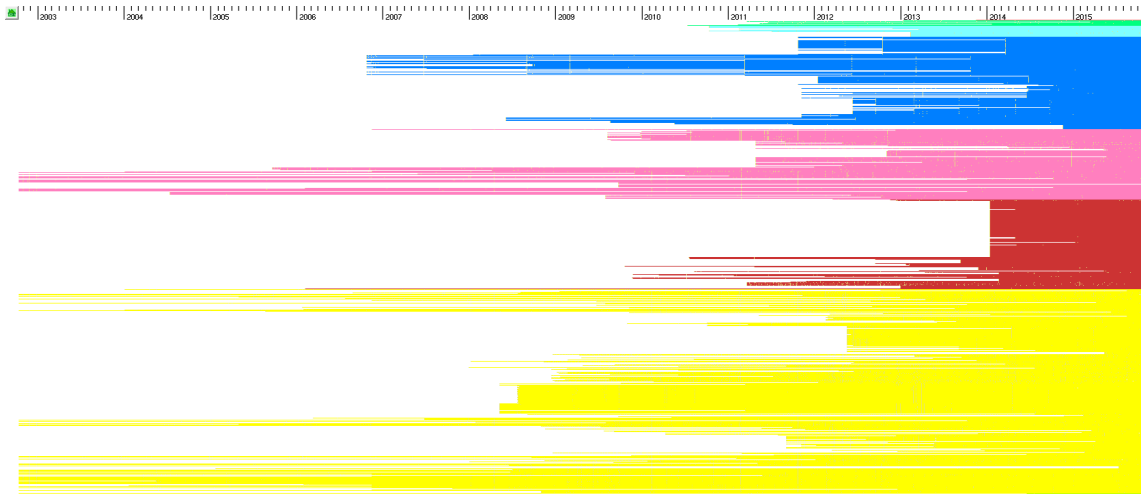


Figure 26: File view of all files sorted alphabetically, with the folder metric enabled.

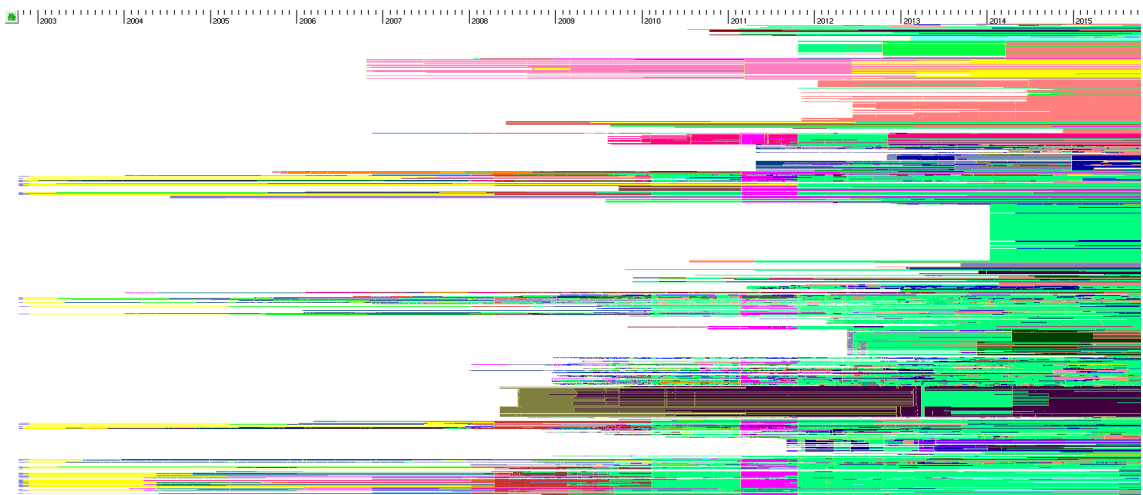


Figure 27: The same as Figure 26, replacing the folders metric for the authors metric.

	/source		Dalai Felinto <dfelinto@
	/source/blender		Janne Karhu <jhkarh@c
	/source/blender/avi		Thomas Dinges <blende
	/source/blender/blendfont		Jeroen Bakker <j.bakke
	/source/blender/blendkernel		Martin Poirier <theeth@
	/source/blender/blendlib		Lukas T##nne <lukas.it
	/source/blender/blendloader		Maxime Curioni <maxim
	/source/blender/blendtranslation		Bastien Montagne <mor
	/source/blender/bmesh		Nicholas Bishop <nichola
	/source/blender/collada		Lukas Toenne <lukas.to
	/source/blender/compositor		Matt Ebb <matt@mke3.
	/source/blender/datatoc		Joseph Eagar <joeedh@
	/source/blender/depsgraph		Nathan Letwory <nath
	/source/blender/editors		Antony Riakiotakis <kale
	/source/blender/freestyle		Joshua Leung <algorith
	/source/blender/gpu		Ton Roosendaal <ton@l
	/source/blender/ikplugin		Sergey Sharybin <serge
	/source/blender/imbuf		Tamito Kajiyama <rd6t+
	/source/blender/makesdna		Brecht Van Lommel <br
	/source/blender/makesrna		Campbell Barton <ideas
	/source/blender/modifiers		
	/source/blender/nodes		
	/source/blender/physics		
	/source/blender/python		
	/source/blender/quicktime		
	/source/blender/render		
	/source/blender/windowmanager		

Figure 28: Legend for the folder metric (/source/blender sub-directories)



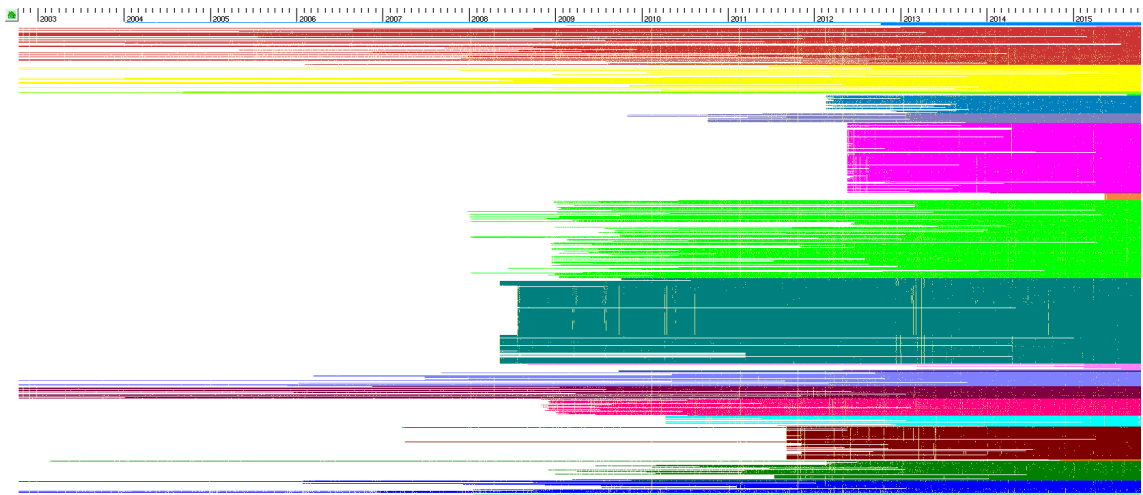


Figure 29: File view of all files in `/source/blender`, including the folders metric, sorted alphabetically. The configuration slider was adjusted so that the sub-directories of `/source/blender` can be identified.

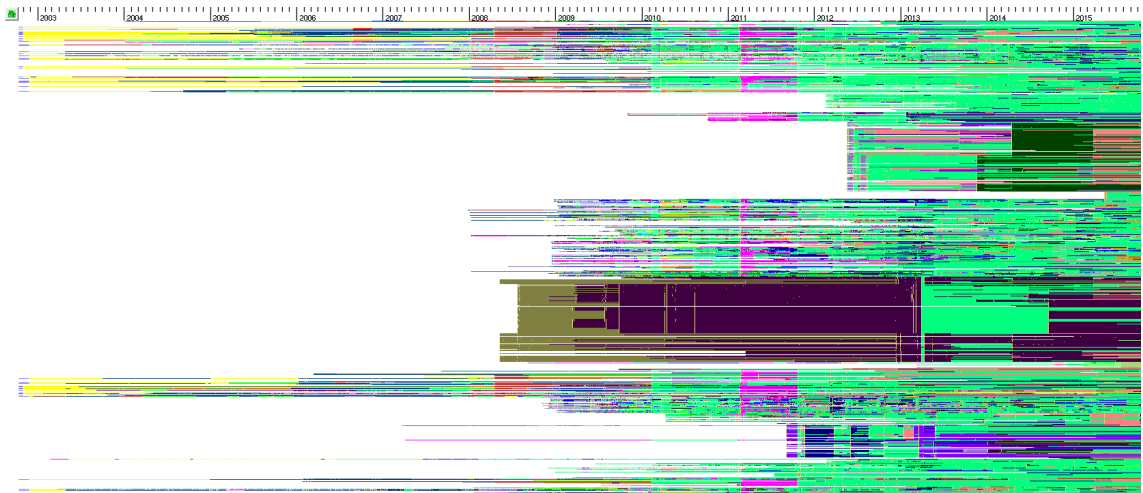


Figure 30: The same as Figure 29, replacing the folders metric for the authors metric.

## 5 Code Size Analysis

In this step, several analyses will be performed regarding the code size of the source code files in the Blender project. During this step, the Code Size metric in SolidTA will be utilised, which will be set to "Lines of Code". The legend in Figure 31 displays the values corresponding to each colour for this metric.

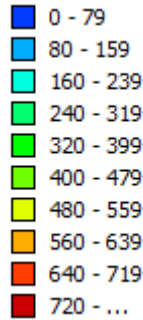


Figure 31: Legend for the colour-coded categories of the Code Size metric in SolidTA.

One aspect of the assignment that we would like to remark on is the fact that it is stated that the Lines of Text calculator must be used to obtain the Code Size metric. This is actually not the case, and the Lines of Text calculator has, in our case, not generated the Lines of Code sub-metric in the Code Size metric. To obtain this Lines of Code (and thus Code Size) metric, the CCCC calculator had to be used on all relevant files of the project, which calculates both the Code Size as well as the Complexity metric, which will be used in the next step.

### 5.1 Size evolution

First of all, the evolution of the file size, in terms of lines of code, throughout the project will be investigated. We have performed two approaches to give an indication of the evolution of file size: an initial, simple approach that turned out to be too naive for getting an indication about the size evolution; and a secondary, more involved approach which considered several subsets of files that were created around the same time.

#### 5.1.1 Initial approach

We will first discuss the initial approach for assessing the size evolution of the source code files in the project. In Figure 32, the file view for all files in the project is given, where the Code Size metric has been enabled and all categories have been selected, and the file is sorted first on creation time and subsequently on file type. We can see clearly in this figure that there are some non-source as well as source files for which the metric could apparently not be calculated or were skipped entirely. It is not sure what causes the CCCC calculator to fail determining the code size metric for source files, we will discuss this further in the *Evaluation* section. By sorting the view on creation time and then on the Code Size metric, we can select only the files for which the Code Size metric is calculated, and sort the resulting view again on creation time and file type, to create Figure 33.

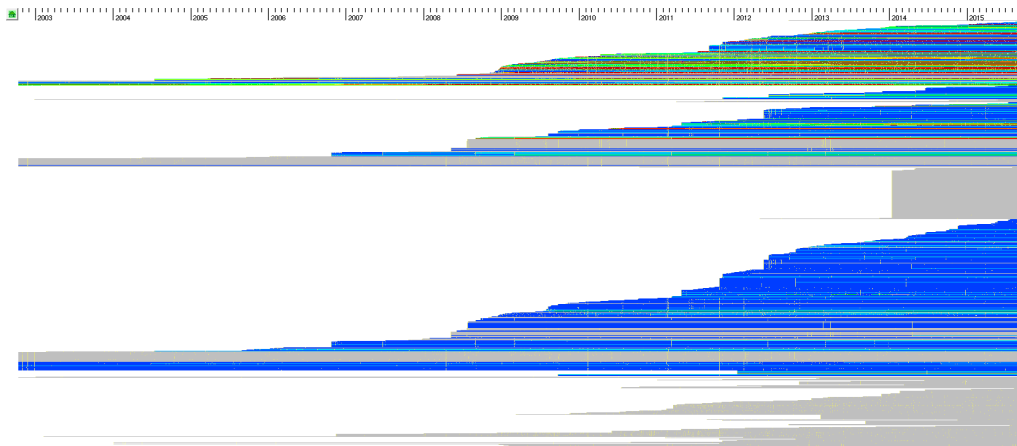


Figure 32: File view sorted on creation time and file type, with the Code Size metric enabled. Revisions displayed in grey are part of a file extension that could not be interpreted by the metric calculator, or failed due to other reasons.

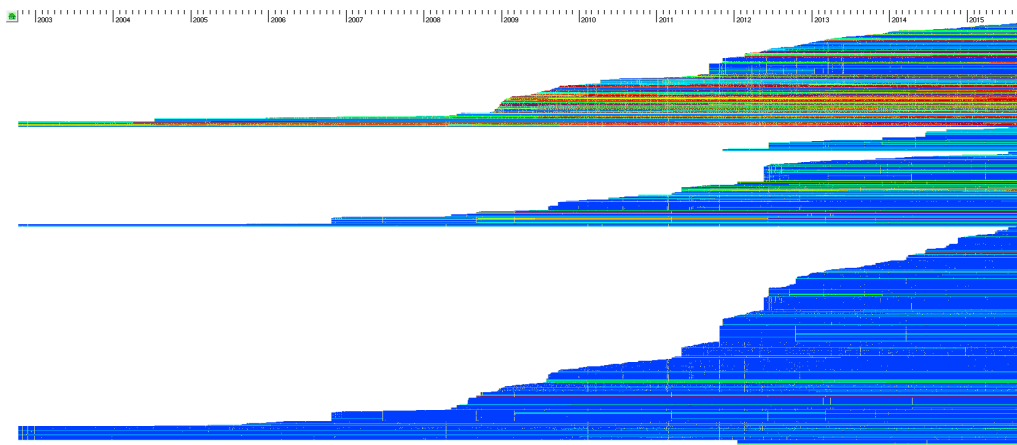


Figure 33: File view sorted on creation time and file type, for all files which have a value for the Code Size metric. (.c, .cc, .cpp, .h, .hpp and other extensions for C/C++)

In Figure 33, the only types of files that are still present are congregated in four chunks. The first large chunk consists of .c files, the second minor chunk consists of .cc files, the third large chunk of .cpp files, and the very large chunk at the bottom consists of header files. The figure indicates that for the header files, there is little to no change of code size over extended periods of time. Therefore, for these types of files the code size does not seem to evolve positively or negatively over time, but rather tends to stay the same over time. For the other file types, it is not clear from this figure what the evolution is, and requires further investigation. In Figure 34, the same file view is given as in Figure 33, apart from the fact that the header files are now excluded, so only the top three chunks are still present. The view is again sorted first on creation time, and then on type. In Figure 35, the same view is given sorted only on creation time.

These figures indicate that the code size is indeed very different per file and between revisions of files, as there are file revisions in each colour-coded code size category, but it is still not clear how the average code size of these files evolves over time.

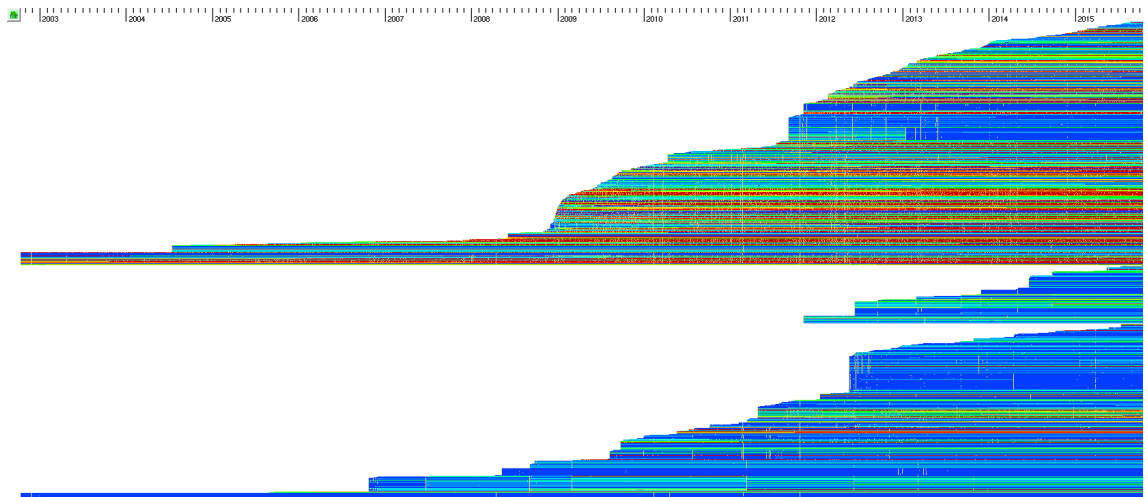


Figure 34: File view similar to Figure 33, not including the header files.

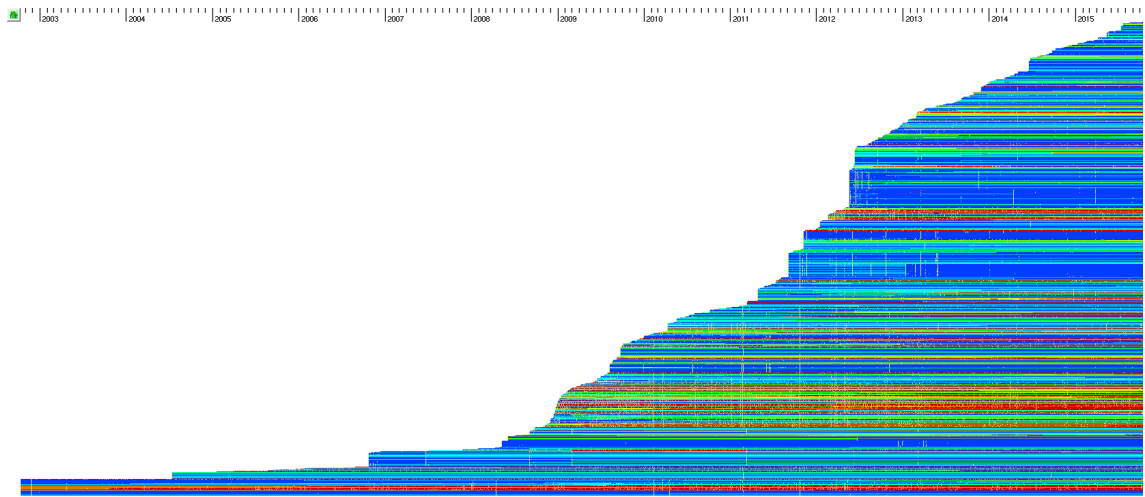


Figure 35: The same as Figure 34, only sorted by creation time.

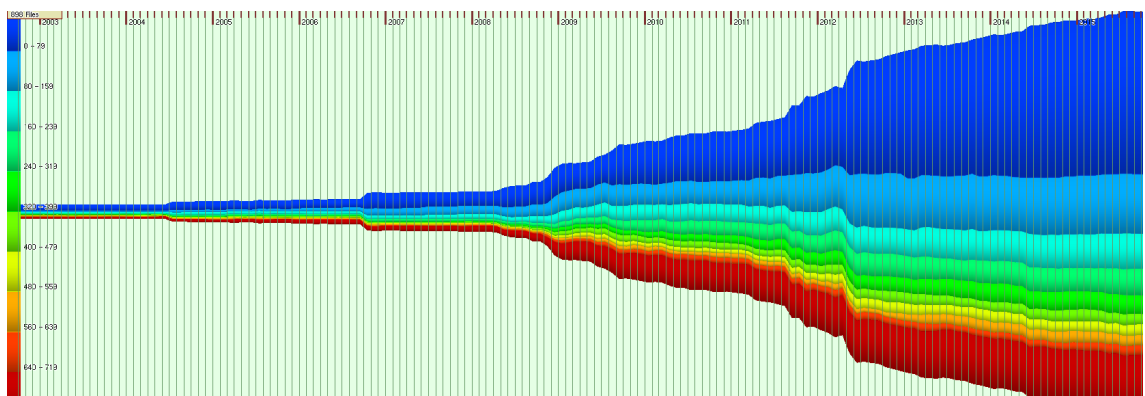


Figure 36: Trend view of the code size metric corresponding to the file views in Figure 34 and Figure 35, with display type Flow and set to display file count.

In an attempt to assess the evolution of the code size for this selection of files, we enabled the evolution view of the code size metric, which is depicted in Figure 36. However, we cannot distill any useful indications from this figure, due to the fact that this evolution view corresponds with files that have been created throughout the entire project. Therefore, an increase of the amount of files for a code size category is not just the result of an actual increase of the code size of files of a lower category, but also due to the fact that new files are continuously added which may fall in those higher categories from the start. We can conclude that this naive approach will not give us any information about the size evolution.

### 5.1.2 Secondary approach

A different approach is needed to give a clear indication about the size evolution of the Blender project. To do this, instead of showing the evolution of all files, we take the file views in Figure 34 and Figure 35, and choose a series of periods (seven, in this case) which consist of files that have been more or less created in the same period. By doing this, we can show evolution views that will not be subject to continuous influx of new files into the project. These periods and the underlying file view (same as previous figures) are given in Figure 37, where the black rectangles indicate periods, which are identified by S1, S2, ..., S7. These are also the names of the corresponding selections in SolidTA.

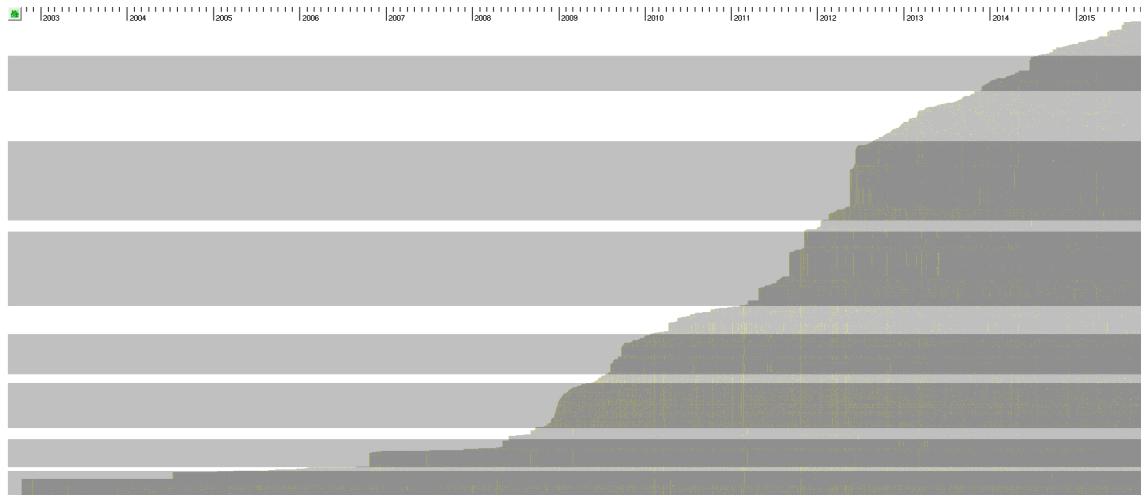


Figure 37: The seven periods, labelled from S1 to S7 from bottom to top, that will be used to give an indication about the evolution of code size.

For each of these seven periods, the corresponding evolution views of the code size have been depicted in this document. We can see that indications for the evolution of the code size are much easier distilled from these figures, due to the fact that the amount of files stays the same after a short stabilisation period (in which new files are still added, due to the fact that these files were not created at exactly the right times). We will briefly discuss each of these figures. From the first period, S1 depicted in Figure 38, we see that the lowest code size category slightly decreases in file count, and that in particular the highest category slightly increased in file count, indicating that the code size has increased on average over time, but the indication is not very strong. Therefore, we will formulate two hypotheses: one that suggests that source files are growing over time, and one that suggests that the size of source files remains more or less stable on average over time.

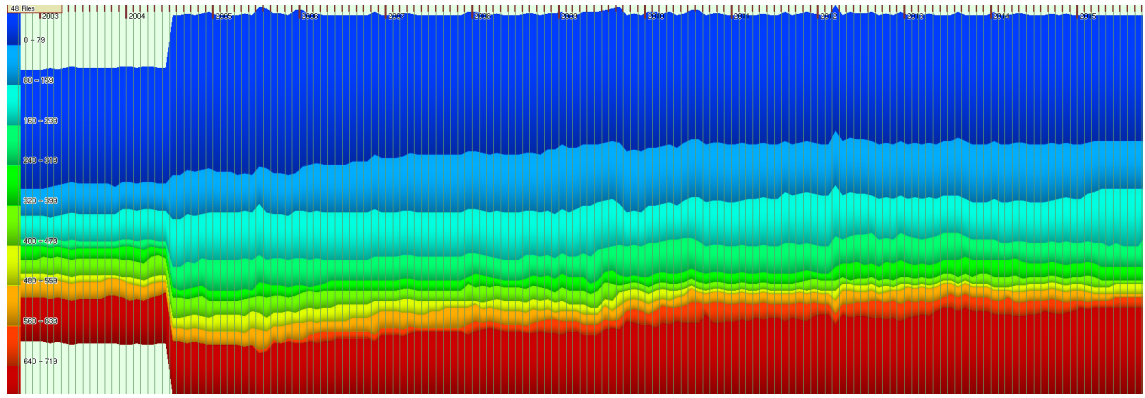


Figure 38: Evolution view of code size for S1.

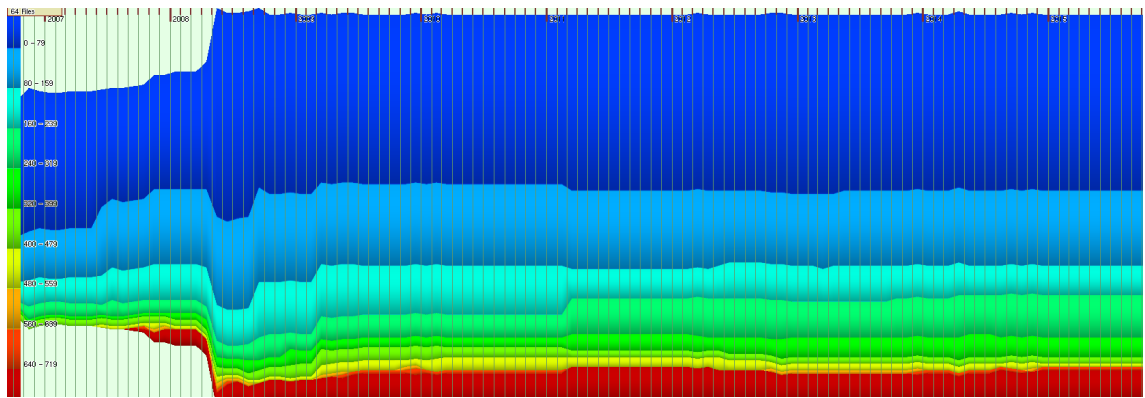


Figure 39: Evolution view of code size for S2.

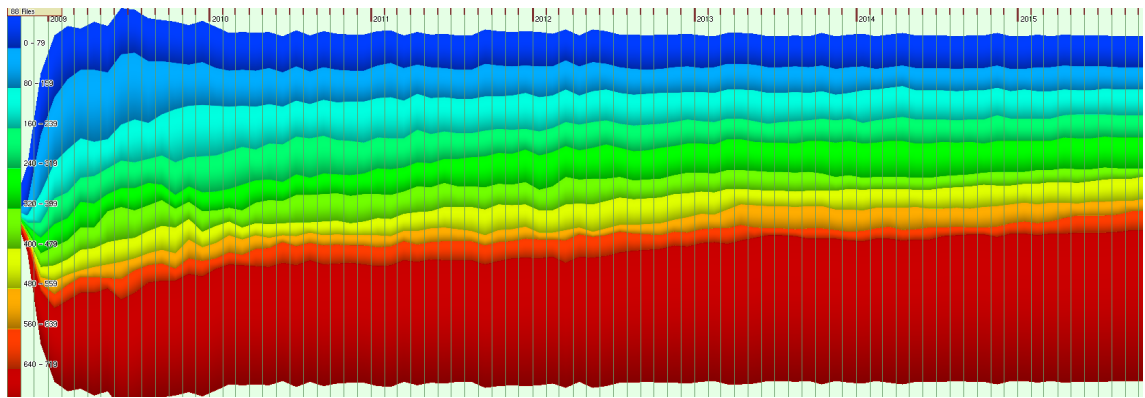


Figure 40: Evolution view of code size for S3.

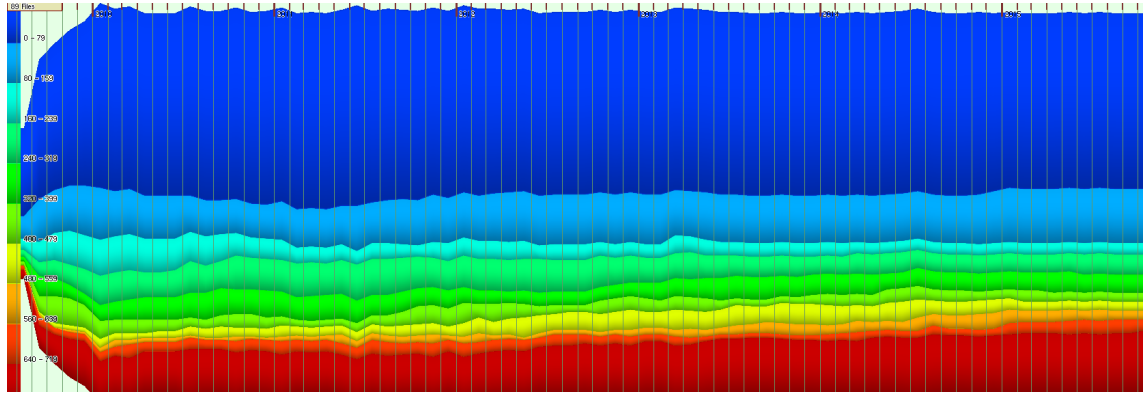


Figure 41: Evolution view of code size for S4.

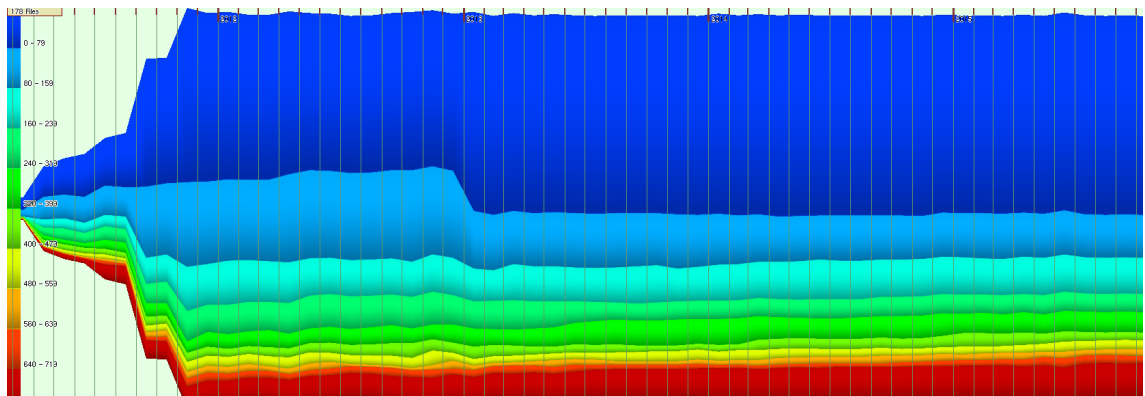


Figure 42: Evolution view of code size for S5.

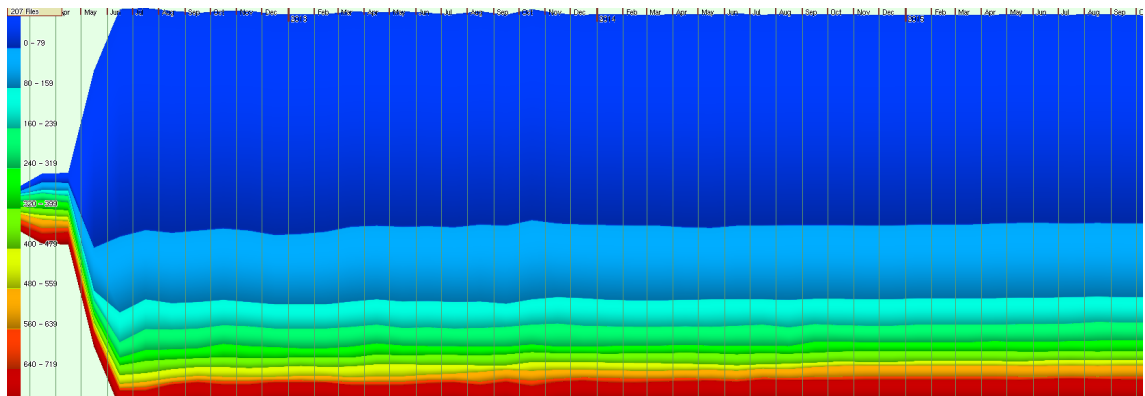


Figure 43: Evolution view of code size for S6.



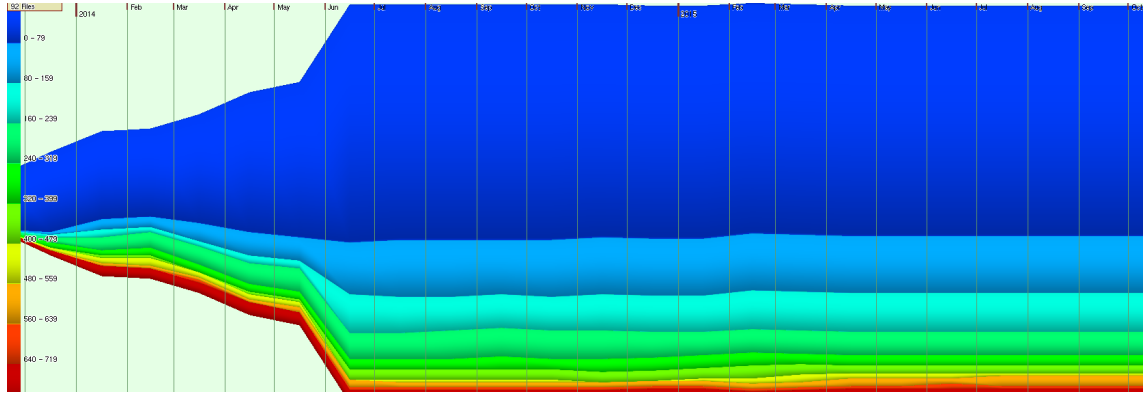


Figure 44: Evolution view of code size for S7.

To support or challenge these hypotheses, we look at the evolution views for the other periods. For Figure 39, the code size for the middle range categories seems to experience some bumps where lower categories lose some files in the benefit of higher categories. On average, however, the categories seem to remain stable in terms of file count. The same can be observed in Figure 41, Figure 42 (which even shows a bump in 2013 where the code size decreases on average, possibly due to refactoring activities) and Figure 43, where the code size also seems to remain stable and does not show any significant increases or decreases. In Figure 40 however, we can see clearly that the highest category gains in file count while the other categories become less prevalent. Combining the results of these seven periods, we can conclude that the most likely hypothesis is the one that argues that source files do not shrink or grow significantly on average throughout the project, since six of the seven evolution views seem to support this hypothesis, while only Figure 40 supports the hypothesis that files grow in code size on average throughout the project.

## 5.2 Most turbulent files

We will now investigate which source files grow and shrink the most on average. There is a strong indication that the header files are not among the files that shrink or grow the most, as determined in the last section, so these will again not be included in the analysis.

Files that have shrunk or grown the most are respectively defined to be the files that have once been in the highest/lowest category of code size, and have afterwards belonged respectively to the lowest/highest category, so they have been a part of each category at a point in their existence. To find the files that do shrink or grow the most, we first open the file selection displayed in Figure 35, sort the view based on creation time and then enable the code size metric. Subsequently, we select the highest code size category, and group the files in the view based on code size. This generates the view depicted in Figure 45. In this view, we select the top files that have been grouped together, which are semantically speaking the files that have ever been part of the biggest files of the entire project. Now, we sort these files again on creation time and re-enable the code size metric. We select the lowest category of code size, and we group again based on code size. The view that is now displayed is the one in Figure 46.



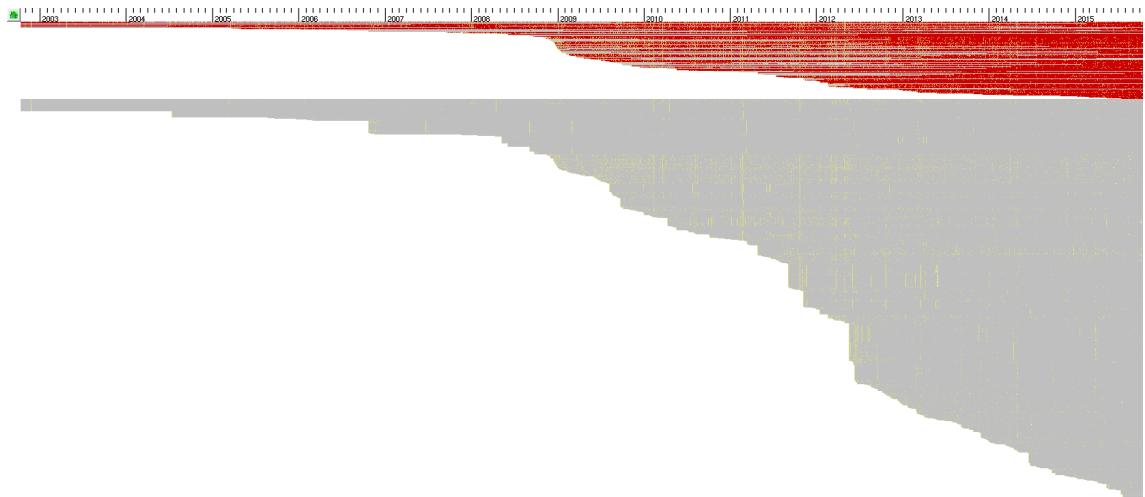


Figure 45: The same file view in Figure 35, sorted by creation time first and then grouped on code size while selecting the highest category.

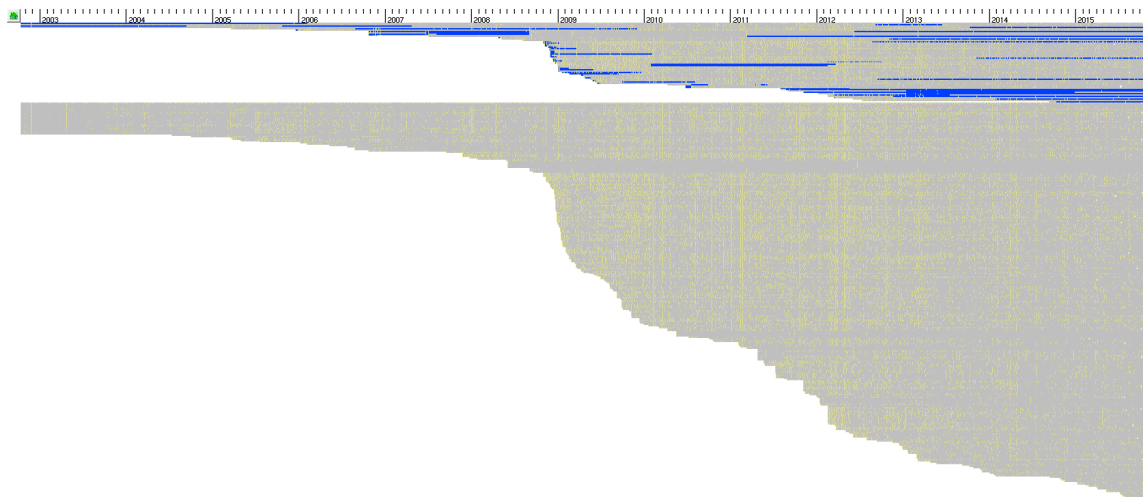


Figure 46: The files at the top of Figure 45, sorted by creation time and then grouped on code size while selecting the lowest category.

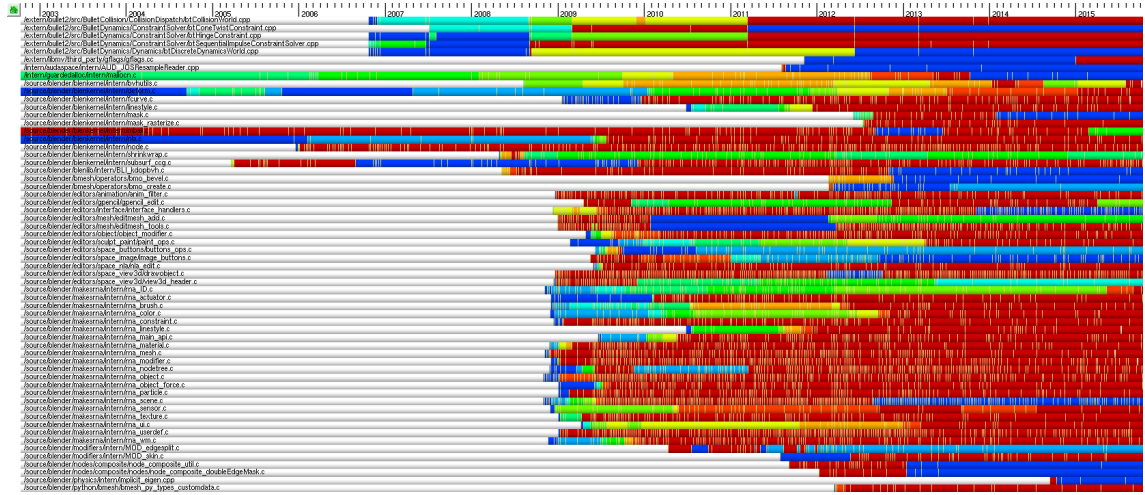


Figure 47: The files at the top of the view in Figure 46, sorted alphabetically, which are the most growing/shrinking files in the entire project.

Semantically, the files that are now in the top group are files that have ever belonged to the lowest code size category, as well as the highest code size category. Therefore, these are the files that will have shrunk or grown the most during the project, since that is the only process that could have affected the code size metric in such a way. We select these top files and sort them alphabetically, which generates the view in Figure 47. The same listing of files has also been given in *Section 11.6*. The majority of these files are a part of the /source/blender folder, and then the /source/blender/blenkernel, /source/blender/editors and /source/blender/makesrna in particular. As discussed before, the /source/blender folder contains the source code for the main functionality of the Blender application. From the fact that these kind of files are included in the list of most shrunk/grown files, this indicates indeed that /source/blender is one of the folders where the most changes in terms of file sizes are made, reflecting its purpose.

### 5.3 Source code fraction

Finally, we are interested in finding out the fraction of source code in the entire project in percentage. The assignment suggests that we should use the code size metric to determine the amount of source code files in the project. Therefore, we take the file view of all files in the project, sort this by creation time and subsequently enable the code size metric and select all categories. Then, we group the view by code size, the result of which is depicted in Figure 48. This clearly separates the files with have a value for the code size metric (which can be viewed as the source code files) opposed to the ones that do not, displayed in grey in the bottom group. By checking in the SolidTA window how many files are in this view, and in the view that is obtained by selecting the top files, we conclude that there are 4628 files that have a code size metric and that the total amount of files in Figure 48 is 8119. Therefore, we could say that the percentage of source code in the entire project is around:

$$4628/8119 \approx 57\%$$

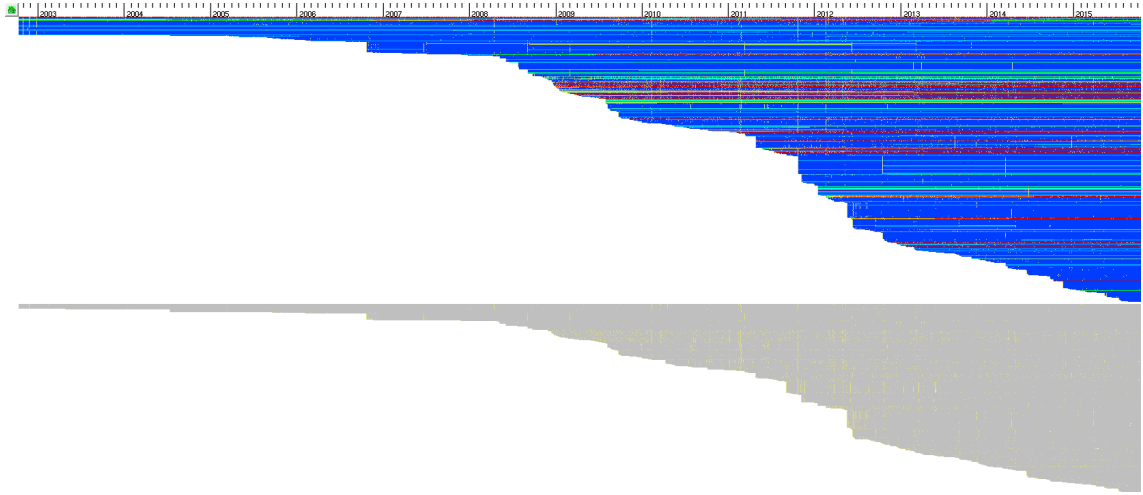


Figure 48: File view of all files, with the code size metric enabled, sorted first by creation time and then grouped by all code size categories, so that the files which have a code size metric are displayed in the top group.

However, the code size metric has only been calculated for files that have an extension related to C or C++, and for some of these files the calculation of the metrics failed. Therefore, we believe that there is reasonable doubt for the accurateness of this calculation, also because the Blender project contains an amount of Python code and code for shading languages, which we also consider to be part of the source code, even if no metrics can be calculated for it using the calculators available in SolidTA.

If we combine the data we found in the *Basic Repository Investigation* in the first step, particularly in Table 1, Table 2 and Table 4 (information on disk space and amount of files for all files and for only source code files, which also includes other types of source code than C/C++), we find that the entire project is about 8654 files accounting for 121.2MB, while the source code is around 6964 files and accounts for 85.1MB.

Based on the number of files that we have found, the percentage of source code is:

$$6964/8654 \approx 80.5\%$$

And based on disk space, the percentage would be:

$$85.1\text{MB}/121.2\text{MB} \approx 70.2\%$$

As is evident, these ways of determining the percentage of source code indicate a much larger fraction of the project and we argue that these hold more ground, given the in-depth details of the structure of the repository in *Basic Repository Investigation*.

## 6 Complexity Analysis

In this section, several analyses will be performed related to the complexity of the source files in the entire project. During this step, we will describe the complexity of the revisions of files using the Complexity metric, which will be set to "Total Complexity". The other sub-metrics of the Complexity metric did not seem suitable for answering the questions of this step, as for the last question the complexity has to be compared against the code size metric, which is not weighted, and all other complexity sub-metrics apart from Total Complexity are in fact weighted. Therefore, we decided to use the Total Complexity metric everywhere in this step when referring to complexity. The values that are used for the categories of this metric can be adjusted with the configuration slider.

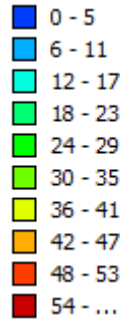


Figure 49: Legend for the Total Complexity metric (under Complexity)

We opted to set the configuration slider in such a way that the highest category is in the range of "54 - ...", because this led to a good distribution of the number of files between the different complexity categories, as can be seen in Figure 50, because the values of the central categories are spread relatively evenly between the highest and lowest category. We have also configured this view with higher and lower values for the configuration, which in general only had the effect of shifting the middle categories up and down in this view while increasing/decreasing the size of the highest/lowest category. This value was found to provide the most stable distribution.

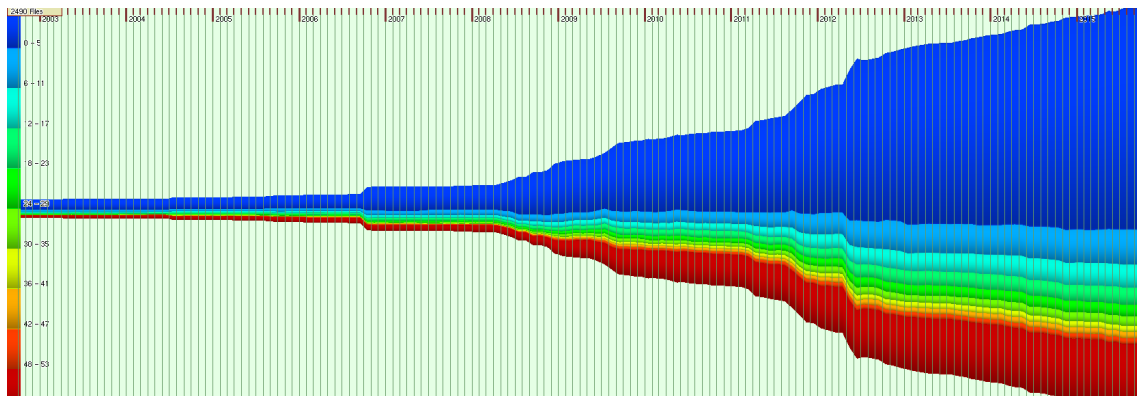


Figure 50: Evolution view of Total Complexity for all files

## 6.1 Most complex files

First of all, we are interested in finding the files in the project that are currently the most complex. We define files as currently the most complex, when their latest revision has a total complexity metric that falls in the highest complexity category. Since the complexity metric is only available for source code files, we only use these files in the initial view. In this initial view, which only contains source code files, we first select the highest complexity category, which gives the view depicted in Figure 51.

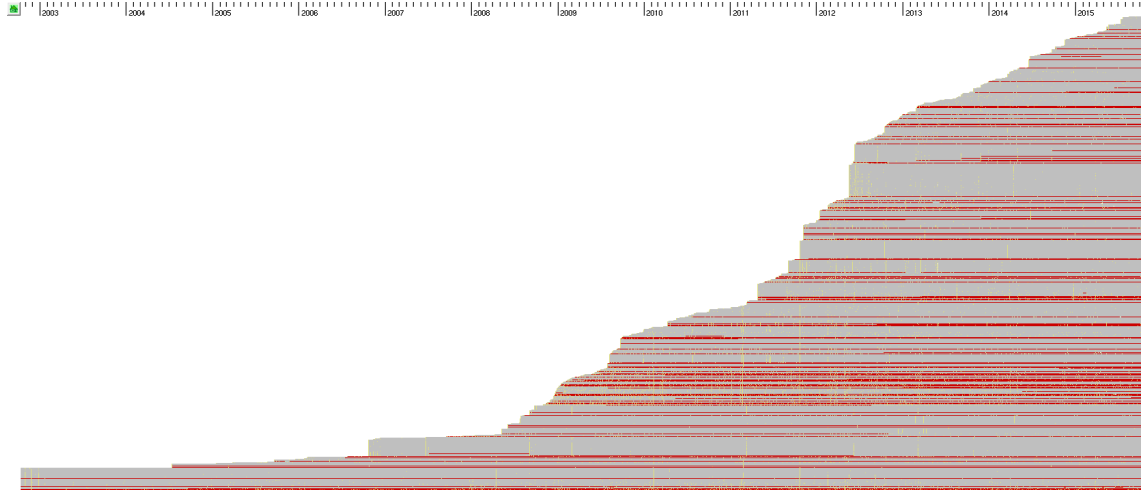


Figure 51: File view of all source files in the project, sorted on creation time, with the complexity metric enabled and the highest category selected.

In this view, we then group the files by the complexity metric, which will put the files with revisions that are highlighted in the top of the figure. These files are then selected and put in their own view, which is displayed in Figure 52, which contains all of the files that have ever had a revision in the highest complexity category.

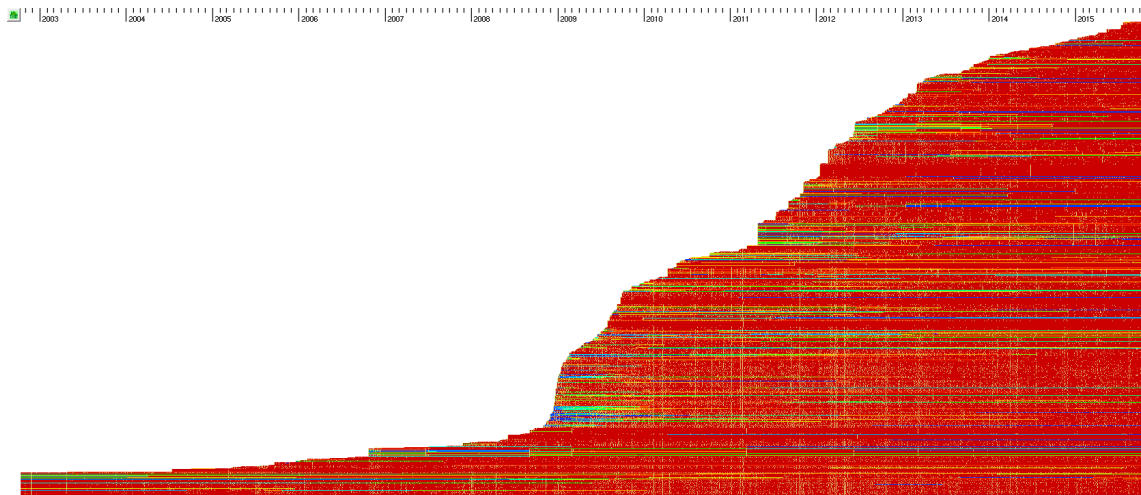


Figure 52: File view of the files that have a revision that is part of the highest complexity category, sorted on creation time.

Now, all we need to do is manually deselect (using the "Inverse Selection" setting in SolidTA) the files from this view of which the latest revision is not a part of the highest complexity category. After this task has been performed, we find the file view in Figure 53, which depicts the files that are currently in the highest complexity category, and are thus considered to be the currently most complex files in the project.

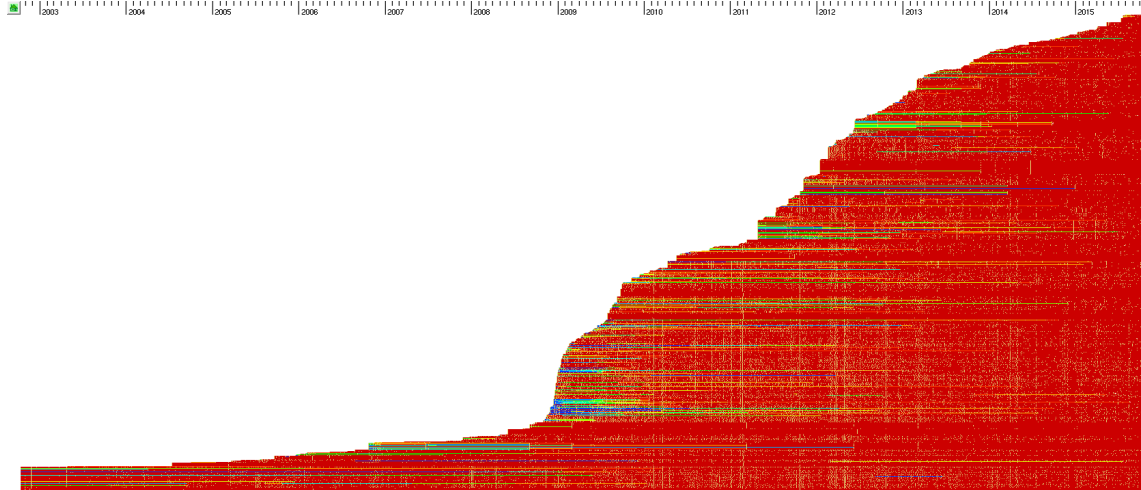


Figure 53: File view of all files in Figure 52 of which the current revision is in the highest complexity category, sorted on creation time.

## 6.2 Changes in complexity

Now that the most complex files have been determined, we are interested in the files that show the biggest increases or decreases in the total complexity metric. We define files with the largest increases in complexity to belong to the highest complexity category right now, but have once belonged to the lowest complexity category. Inversely, we define files with the largest decreases in complexity to belong to the lowest complexity category currently, while having once belonged to the highest complexity category.

To find these files, the view with the most complex files of the previous section will be utilised. Furthermore, we will also require a view with the currently least complex files in the project. To obtain this view, we perform a similar approach as above. First of all, we sort the view of all files based on complexity while selecting the lowest complexity category, and then we distill the top group of this figure into a new view, given in Figure 54. In this view, we subsequently deselect all of the files of which the latest revision does not fall in the lowest category, and the result of this action is Figure 55, which depicts the files that are currently in the lowest complexity category.

To obtain the list of files that increase the most in complexity, we use the view of the currently most complex files in Figure 53, enable the complexity metric, and then select the lowest complexity category. If the view is now grouped by complexity, the top group will contain all of the files that are currently in the highest complexity category, while having ever been in the lowest complexity category. These files are extracted into their own view in Figure 56, which is in fact the list of files that undergo the biggest complexity increase in the project.

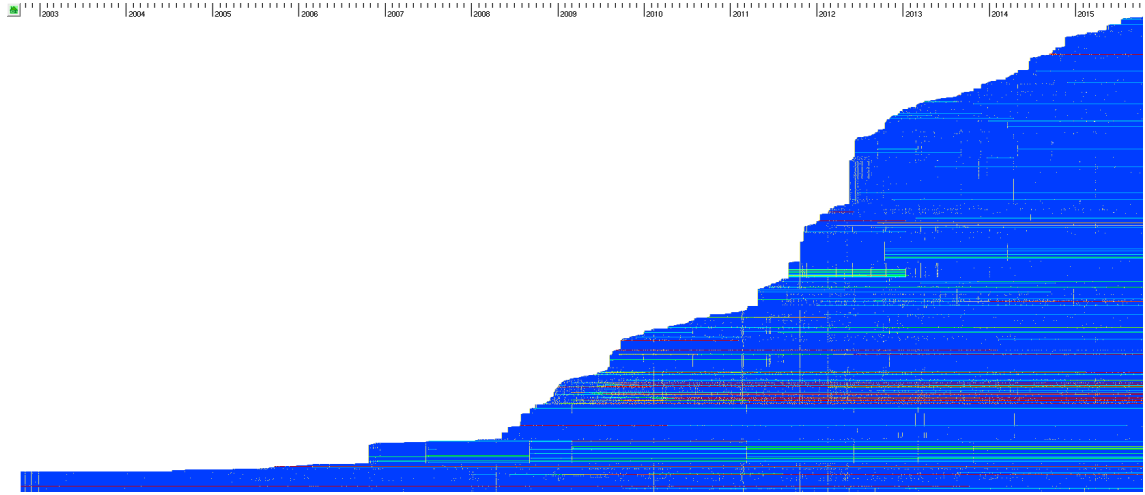


Figure 54: File view of the files that have a revision that is part of the lowest complexity category, sorted on creation time.

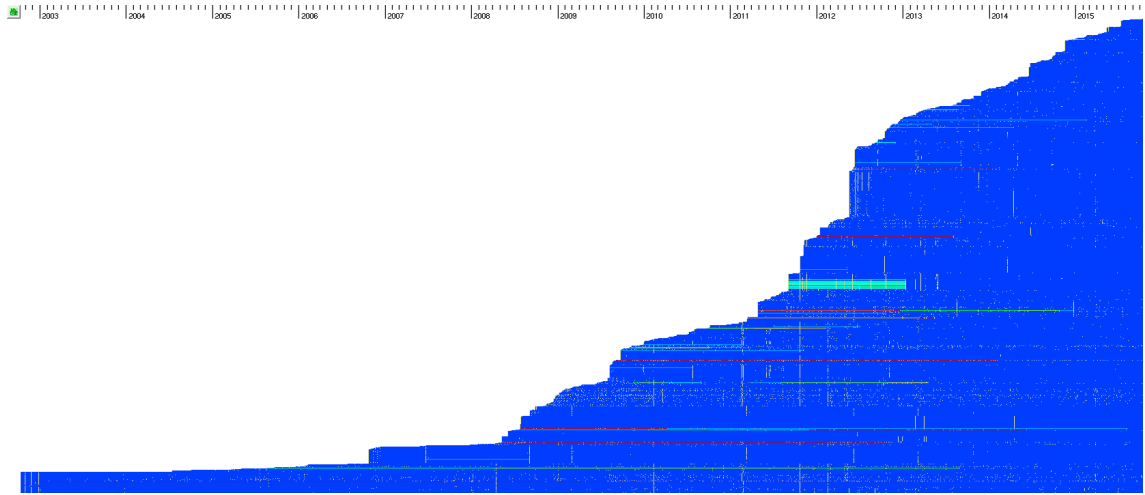


Figure 55: File view of all files in Figure 54 of which the current revision is in the lowest complexity category.

To obtain the list of files that decrease the most in complexity, we perform the same approach as for the list of files with increasing complexity, except that we start with the view of currently least complex files in Figure 55, and select the highest complexity category instead. If the view is then grouped by complexity, the top group will contain all of the files that are currently in the lowest complexity category, but have ever been in the highest complexity category. According to our definition earlier on in this section, this is indeed the list of files that have the highest decreases in complexity, and these files are extracted into their own view in Figure 57.

These two file views of most increases and decreases in complexity show that the most turbulent files (in terms of changes in complexity) are for a large majority .c files.



The files with increasing complexity are mostly related to serialisation of other object classes and extending the functionality of the editor windows, while the files with decreasing complexity are related to displaying nodes and mesh structures in the GUI of Blender and for graphical operations and device interfaces (with GPU in particular).

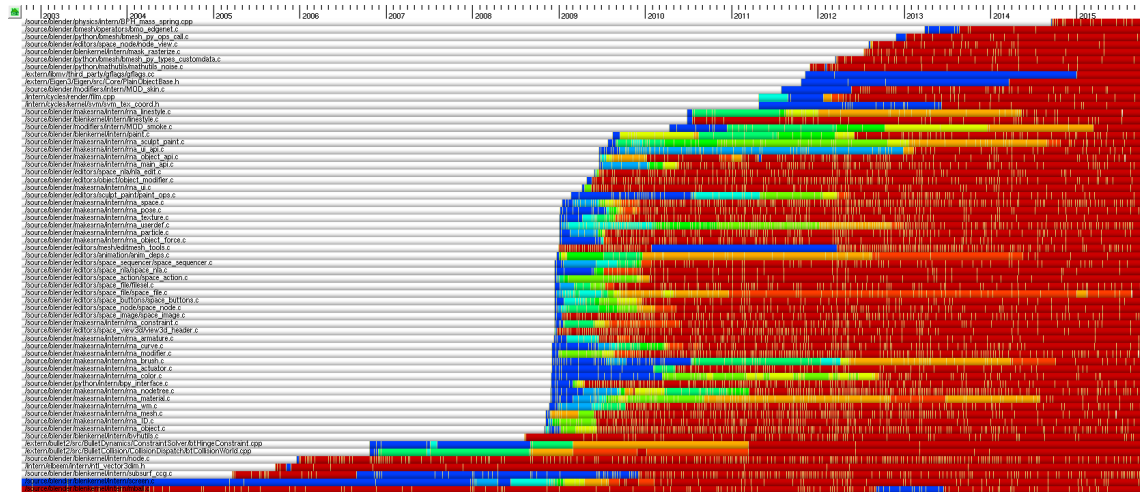


Figure 56: File view obtained by taking the top group after grouping Figure 52 using only the lowest complexity category, sorted on creation time, resulting in the files that have increased most in complexity.

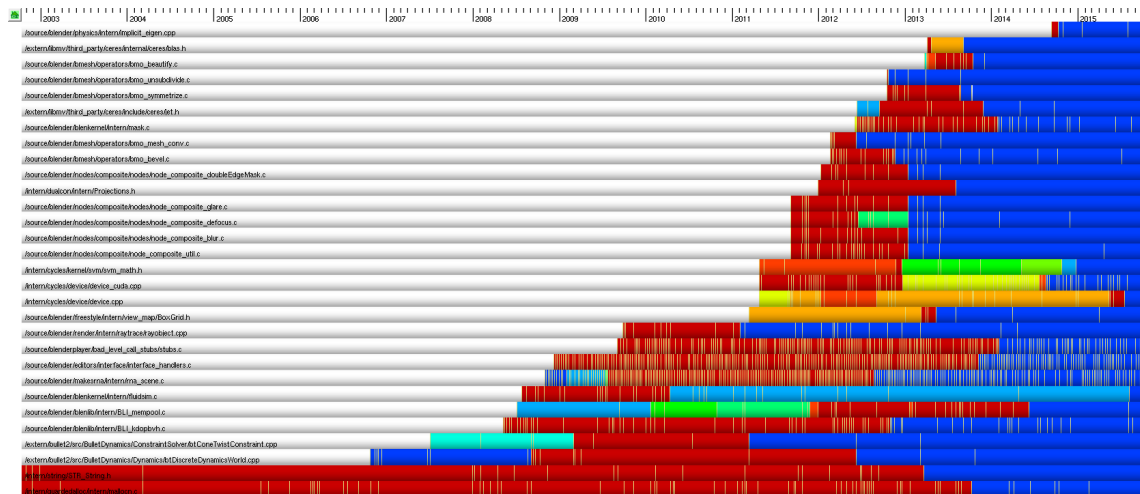


Figure 57: File view obtained by taking the top group after grouping Figure 54 using only the highest complexity category, sorted on creation time, resulting in the files that have decreased most in complexity.

We have also investigated whether there was some correlation between the location and/or functionality of these files and the fact that there is such an increase or decrease in complexity. If we sort both views alphabetically and turn on the folder metric for these views, and set the configuration to the fourth level, we obtain Figure 58 for the files with increasing complexity and Figure 59 for the files with decreasing complexity.



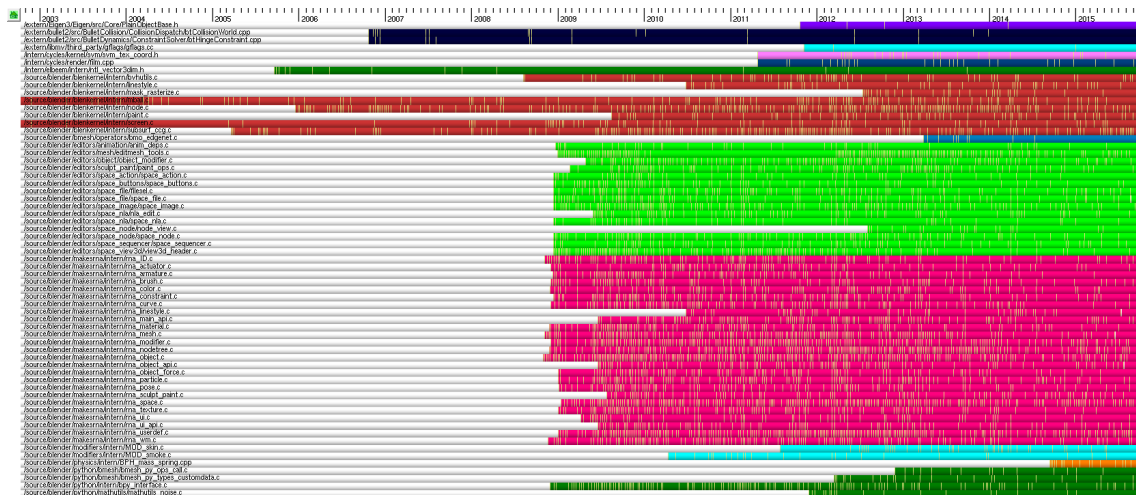


Figure 58: File view of all files increasing most in complexity, sorted alphabetically, with the folder metric enabled on the fourth configuration level.

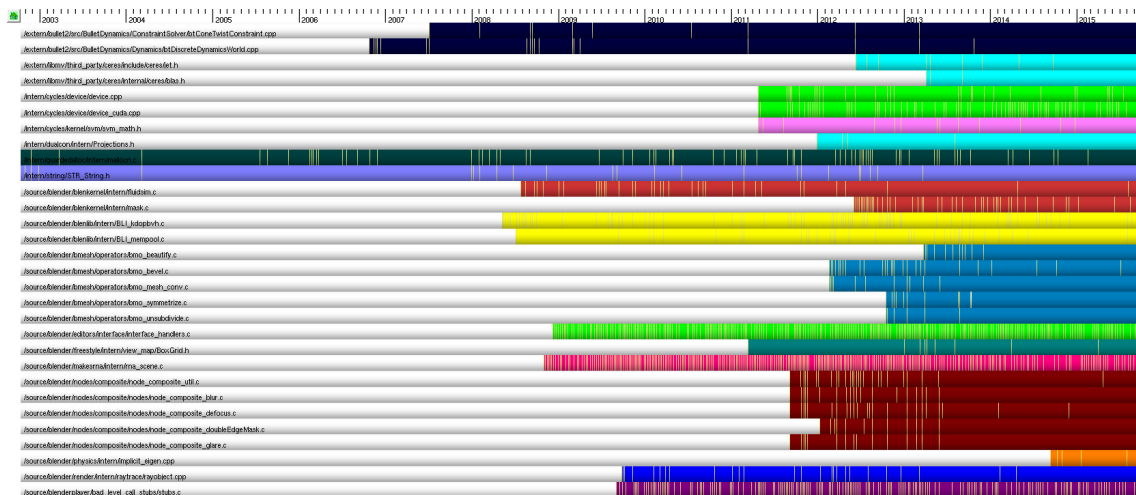


Figure 59: File view of all files decreasing most in complexity, sorted alphabetically, with the folder metric enabled on the fourth configuration level.

For the files that are decreasing in complexity, there are some files which originate from the same folders, but this is not a clear indication of a correlation with such a small sample size. For the files that are increasing in complexity, we actually do see a strong indication with the location, mainly in `/source/blender/blenkernel` (which contains the interfaces of the API available for advanced users, growing in functionality over time), `/source/blender/editors` (code for the graphical editors that have become more complicated due to a larger variety of operations available for users) and `/source/blender/makesrna` (regarding serialisation of data objects, which need to become more complex if their corresponding objects receive more methods or parameters).

### 6.3 Activity correlation

Next, having determined the files that are the most complex, most increasing and decreasing in complexity, we investigate if there is any correlation between these files and the level of activity. In SolidTA, activity is reflected by yellow vertical lines on the bars, where each vertical line indicates a revision being performed on that file (or a set of revisions if many revisions were performed at the same time).

For the files with increasing complexity, Figure 60, the files with decreasing complexity, Figure 61, the files which are currently the most complex, Figure 62, and the files which are currently the least complex, Figure 63, we have sorted the file views based on activity, with the complexity metric enabled for all categories, and we compare these figures.

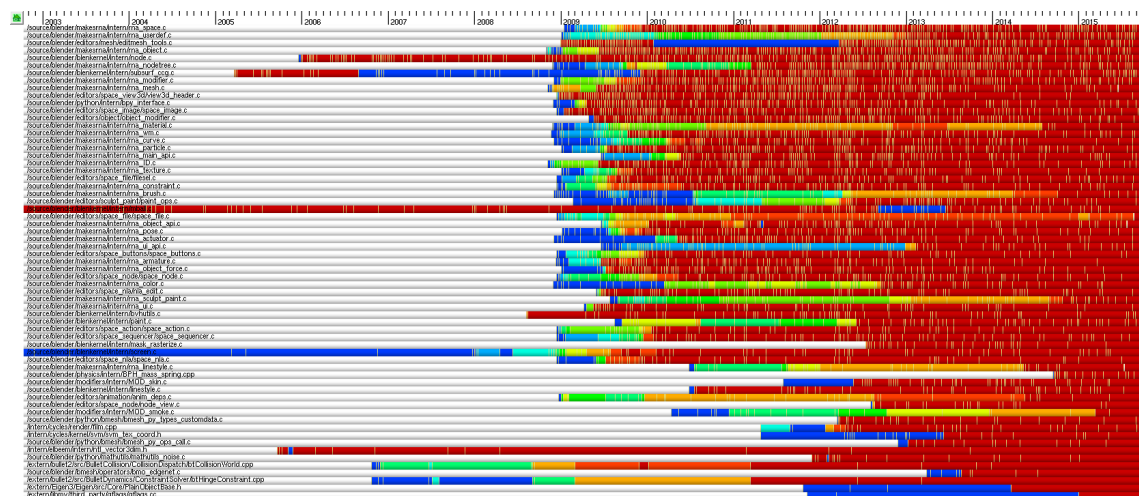


Figure 60: The same file view as in Figure 56, sorted on activity, with the complexity metric enabled for all categories.

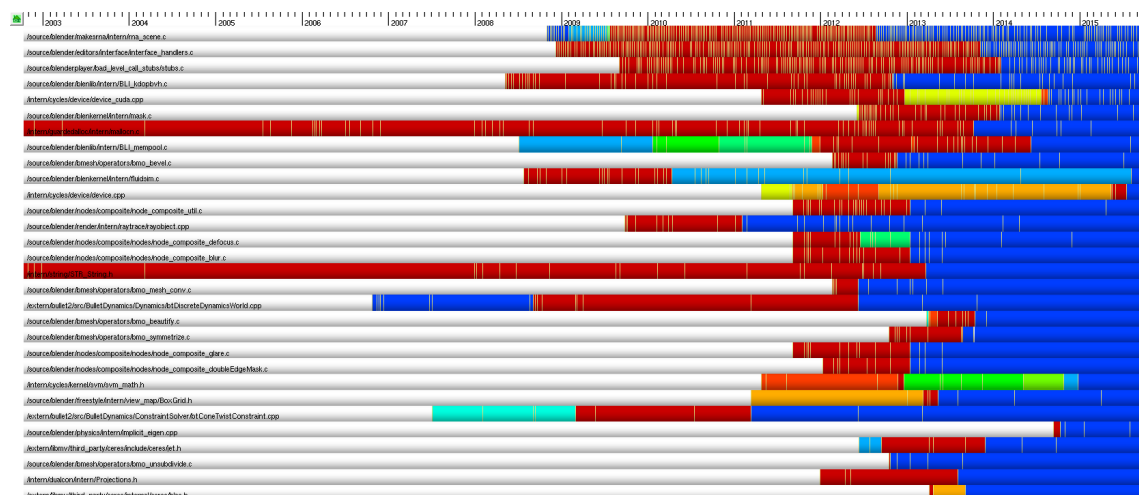


Figure 61: The same file view as in Figure 57, sorted on activity, with the complexity metric enabled for all categories.

Figure 60 and Figure 61 indicate that files with decreasing complexity have less activity than the files with increasing complexity, due to the difference in proportion of activity (presented in yellow vertical lines) between these figures. There are of course some files in the figure with decreasing files with high activity, but these do not weigh up against the other figure where almost all files are indicated to be moderately to highly active. We hypothesise therefore that high/increasing complexity may be directly correlated with highly activity. This hypothesis is also supported when looking at the currently most and least complex files in Figure 62 and Figure 63, where the former clearly indicates a much higher activity on average compared to the latter.

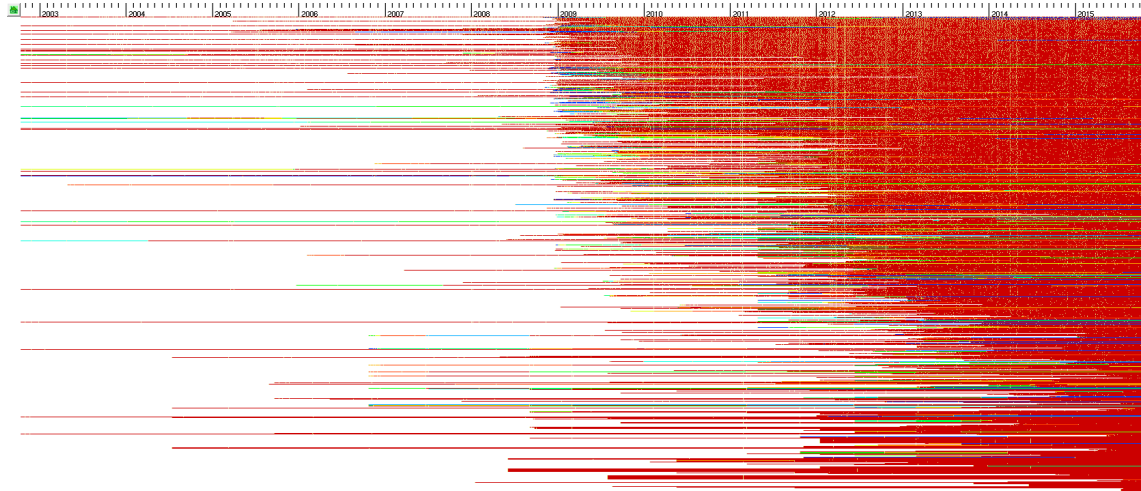


Figure 62: The same file view as in Figure 52, sorted on activity, with the complexity metric enabled for all categories.

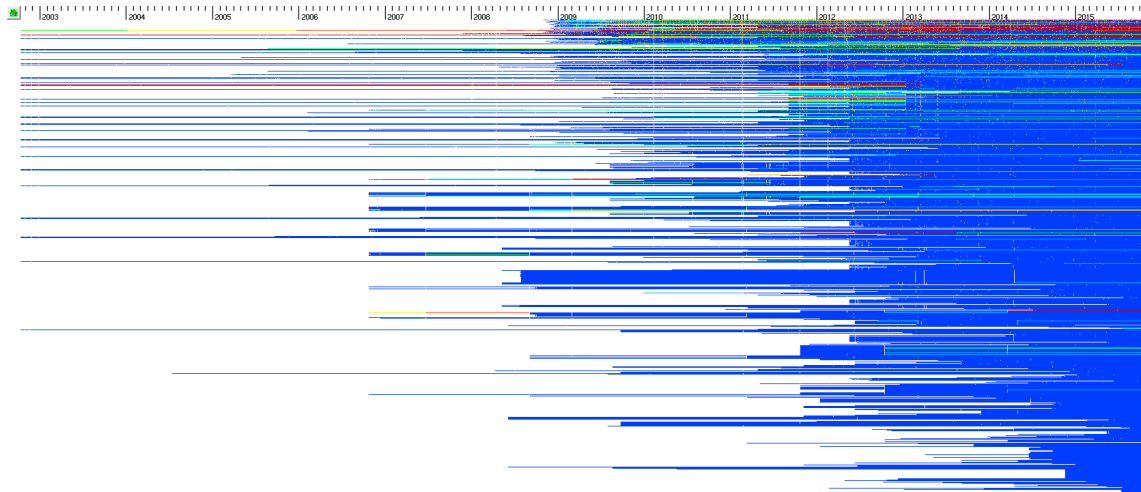


Figure 63: The same file view as in Figure 54, sorted on activity, with the complexity metric enabled for all categories.

To confirm our deny our hypothesis, we take the list of files of the entire project that have a value for the complexity metric. Then, we sort this list of files based on activity, as presented in Figure 64. Since the top of this view, with the highest activity, shows the majority of the files in the higher complexity categories, this is an indication that the most complex files are also the most active files. If we only select the highest or lowest category of complexity, represented respectively in Figure 65 and Figure 66, we can clearly see that the data indicates that the hypothesis that we described is supported.

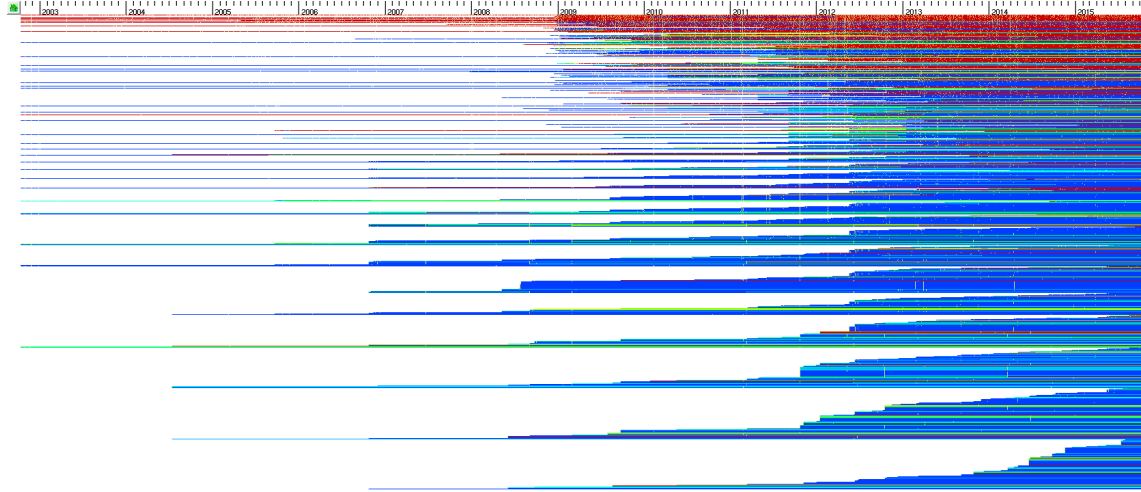


Figure 64: File view of all files that have a value for the complexity metric, sorted on activity, with the complexity metric enabled.

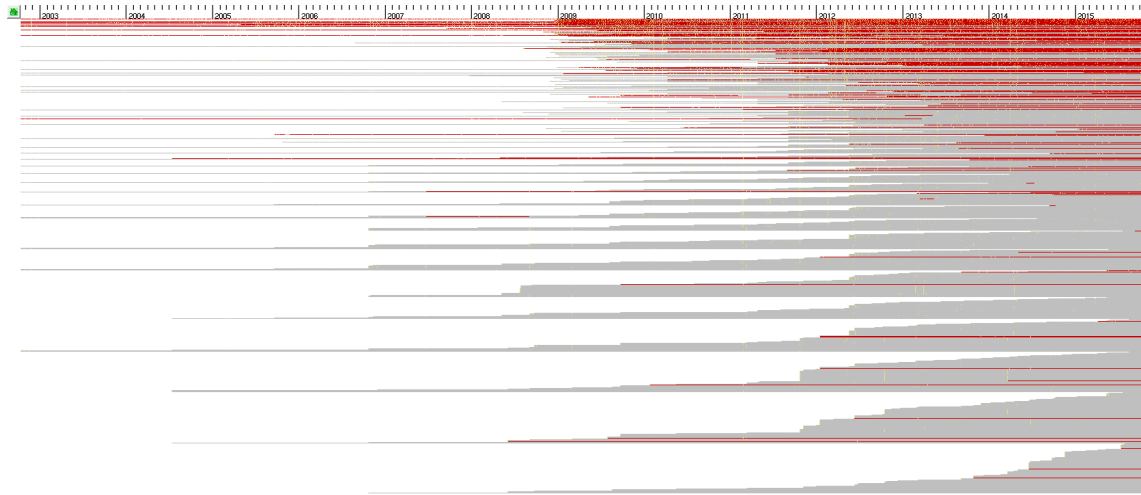


Figure 65: File view of all files that have a value for the complexity metric, sorted on activity, with the complexity metric enabled for the highest category only.

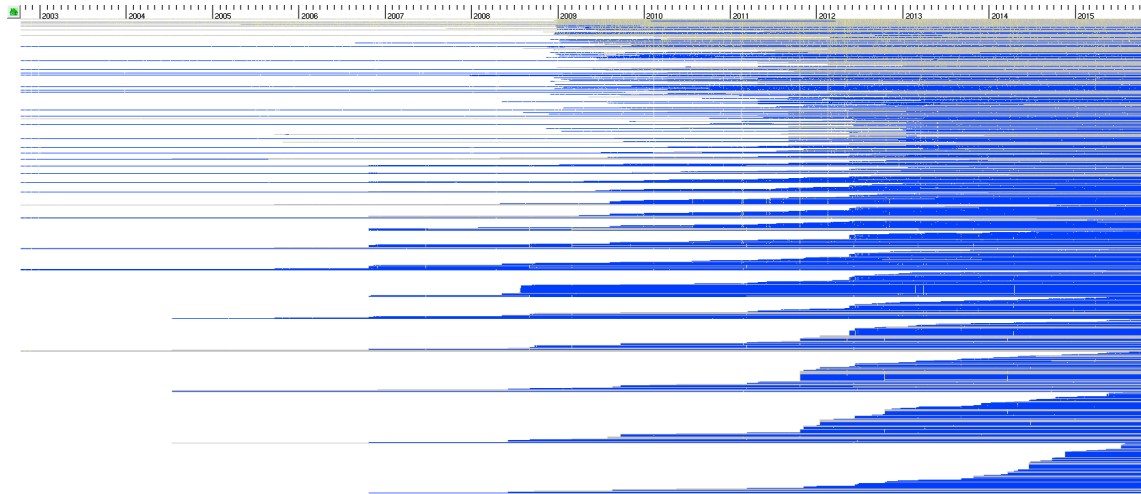


Figure 66: File view of all files that have a value for the complexity metric, sorted on activity, with the complexity metric enabled for the lowest category only.

## 6.4 Size correlation

Finally, we will establish if there is a correlation between complexity and code size, measured in Lines of Code as described in the *Code Size Analysis*. The same settings for the code size metric as described in that section will be used as well, please refer there for a legend for the different code size categories. For the complexity metric, the same settings as elsewhere in this section will be utilised.

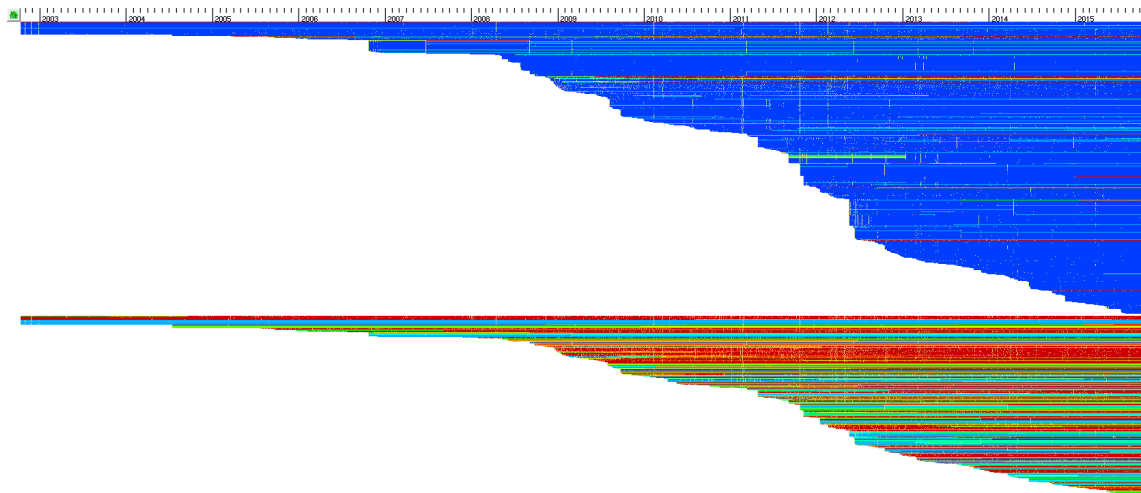


Figure 67: View of all source files, sorted on creation time and then grouped by the lowest complexity category.

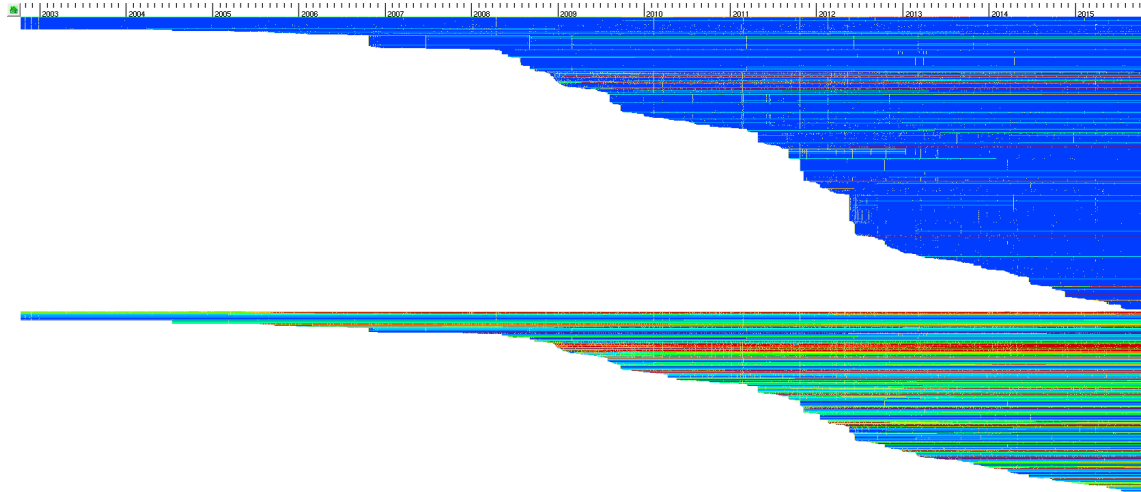


Figure 68: The same as Figure 67, with the complexity metric disabled and the code size metric enabled.

We hypothesise that there is a direct relationship between complexity and code size, due to the fact that in our experience, more source code gives room for an increase in total complexity of that source file. Therefore, to see if this hypothesis is true, we will take the file view of all files in the project for which the complexity metric is available (and thus also the code size metric, since these are both calculated through the CCCC calculator in SolidTA) and sort these first on creation time, then on the lowest complexity category, obtaining the view in Figure 67, where the top group indicates files with the lowest complexity adversely to the bottom group, which indicates the higher categories of complexity.

Now, we turn off the complexity metric and turn on the code size metric, as depicted in Figure 68. Indeed, it is clearly indicated that the source code with the highest code size is located in the bottom group, which is the group with highest complexity. Furthermore, we have also performed the opposite operation, which consist of sorting the view of all source files based on the highest complexity category, as depicted in Figure 69, in which the top group contains the most complex files, and the bottom group contains the lower categories of complexity.

If we now switch the complexity metric for the code size metric, we see again that the files which fall in the higher code size categories also fall in the highest complexity categories, in Figure 70. Therefore, we conclude that there is a high indication that the hypothesis of complexity being directly linked to code size is correct.

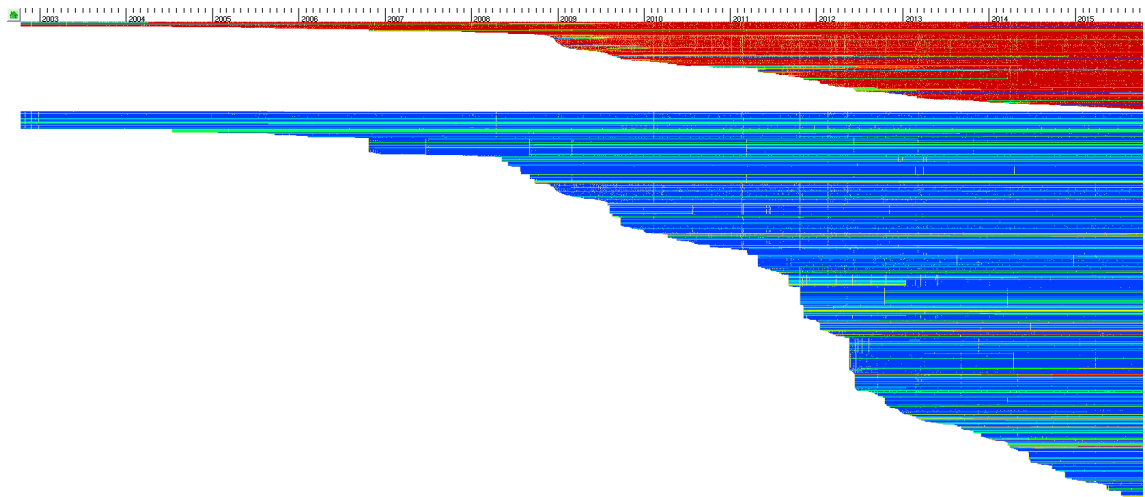


Figure 69: View of all source files, sorted on creation time and then grouped by the highest complexity category.

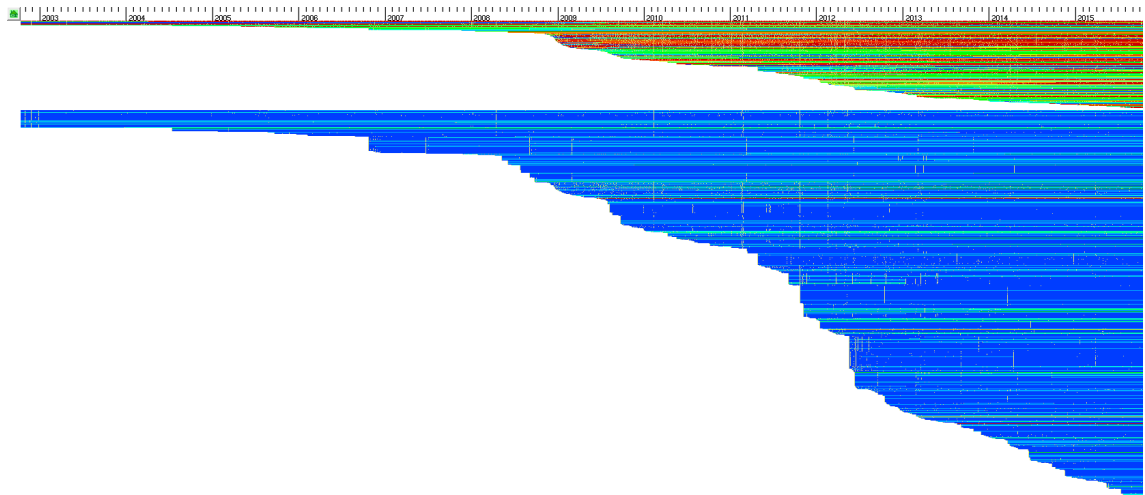


Figure 70: The same as Figure 69, with the complexity metric disabled and the code size metric enabled.

## 7 Dependency Analysis

In the elaboration of this assignment, we have used the Solid Trend Analyzer to draw conclusions related to the absolute values of software metrics, the relation between software metrics, and the correlation between these metrics and the types, authors and locations of files. While viewing these properties of files in itself is a very powerful technique for analysing a software repository, the source code of these repositories often refers to other files by means of calls, inheritance, containment or build dependencies, which is a type of analysis that is missing from tools such as the Solid Trend Analyzer. In this short essay, we will devise a data model and visualisation strategy for displaying these kind of dependency graphs and describe these in detail.

### 7.1 Data modelling

Several different dependency graphs exist in a software source code repository. To visualise the dependency evolution of each of these graphs, a suitable data model is required first and foremost. The first part of this essay proposes such a data model, discusses some implementation details for efficient implementation of the model and describes the steps necessary to include a new version of the software source code into the model. The different types of dependency graphs that are considered here are: the call graph, the inheritance graph, the containment graph, and the build dependency graph.

- **Call graph:** a graph that displays the calls that are being made between source code components. The nodes are the function definitions, while the edges are the function calls being made.
- **Inheritance graph:** a graph that shows the inheritance between one object to another. The nodes on these graph are classes or objects, while the edges point out the different inheritance relations between those classes or objects.
- **Containment graph:** a graph which focuses on the physical containment of one object in another. The nodes could be a package, a folder, a file, a namespace, classes or functions, and the edges are the containment relationships between these objects/concepts.
- **Build dependency graph:** a graph that shows how the build time of files affects the build time of other files that depend on the former file and vice versa. The nodes are the files, the edges indicate the dependencies between those files related to compilation.

#### 7.1.1 Data model proposal

Now that the scope of the visualisation tool has been clearly defined, namely the effective visualisation of evolution in different types of dependency graphs, we can begin constructing a data model for these graphs. The data model is required where all the dependency graphs can be stored in. Not only should the model store the nodes and edges of the graph itself, it should also contain the correspondence information for all the nodes and edges which indicates how an element of version  $i$  corresponds to an element of version  $i + 1$ .



The different types of dependency graphs that are being analysed can be combined into one large graph that shows all of the information that would otherwise be captured in each of the separate graphs. Because of this property, the data model can be simplified, since it does not have to store each of the graphs separately. The data model that is proposed to store all of the information for all types of dependency graphs is shown in Figure 71.

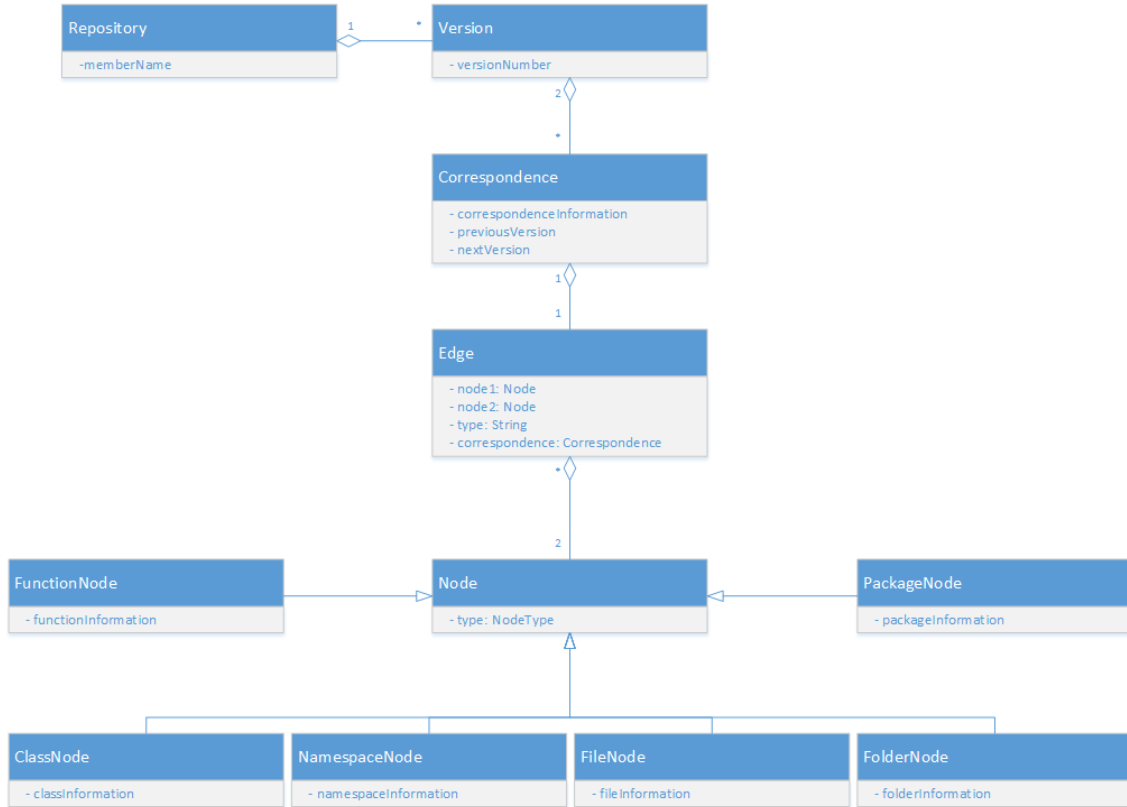


Figure 71: Proposed data model for the visualisation tool

Since it is possible to combine all of the dependency graphs into a single graph, the data model does not have to record each of the graphs separately and represents one large graph. Several types of nodes are used in the model such as a node for a function, class, namespace, file, folder and package. These types of nodes correspond to the types of nodes required by the different dependency graphs. All of the dependency graphs show a relationship between two nodes with an edge between the two nodes, which is also captured by the model. However, because the relationships of several dependency graphs are combined, the edge has a certain type that indicates from which dependency graph the relationship is taken.

The parts of the model discussed so far capture all of the dependency graphs for a certain version of a software repository. However, the model also has to capture the correspondence information that connects an element from version  $i$  to the next version  $i + 1$ . To include this information in the model, every relationship between two nodes (indicated by an edge) is provided with the so-called correspondence information. This indicates how the relationship between two nodes in version  $i$  relate to the next version,  $i + 1$ . This, in turn, is linked to some information regarding the version of the software, which all belongs to a certain repository.

The proposed data model captures the information of all of the dependency graphs into a single graph. By doing this it is slightly more difficult to display the evolution of the software source code based on a single dependency graph. When only the information from a single graph is required, the entire graph has to be traversed, but only the nodes that are connected by an edge of a specific type and the edge itself should be displayed. Since the graph also contains the relationships from all of the other dependency graphs, this is computationally more expensive than when the graphs would be recorded separately.

When the information of a single dependency graph must be shown, it is also more difficult to extract the information from the next version, since it is not possible to simply follow the correspondence information included in the edges. This is due to the fact that not every part of the graph is linked by the correspondence information. For example, when a new function is inserted in the source code, none of functions in the previous version have any correspondence to that new function. Therefore, it is also required to traverse the entire graph of the next version and filter out any information that is not required.

However, while it is slightly more difficult to display the information of a single dependency graph, the model gives great flexibility when the information of two or more dependency graphs have to be combined and displayed. It is also much more efficient with respect to the data storage required, since some dependency graphs share certain information. The call graph and containment graph, for example, both include function nodes in the graph. When these graphs would be constructed separately, the same function nodes would have to be duplicated across the different graphs, and when the correspondence information must be determined, it must be determined once for each graph. By combining the information into a single graph, the function node only has to be recorded once and the correspondence information for that node can also be used for the other dependency graphs that also include that node.

### 7.1.2 Implementation decisions

Software repositories can contain a very large number of files and each of the files can potentially contain many lines of code. This can also result in very large dependency graphs. Therefore, it is very important that the data model is implemented in an efficient way.

To make the implementation efficient, it can be built on top of a graph database. Because of the inherent feature of maintaining direct links to properties and the corresponding nodes, it is very efficient to search for the relationships of a certain node. This can even be used for the correspondence information, which makes it possible to link a node directly to the node of the next version.

Another way in which the implementation of the data model can be made more efficient is by parallelising the extraction of information from the source code per version. When all of the graphs have been obtained, the graphs of successive version of the source code can be analysed to find the correspondence information for the different data elements. It might also be possible to extract the different dependency graphs from a single version, however, because all of the information is captured into a single graph this might prove more difficult.

### **7.1.3 Updating the model**

Whenever a new version is committed, that new version must also be included in the data model. Adding the new version to the data model happens in a two-step process. First, all of the different types of dependency graphs must be extracted from the source code and combined into a single graph. Once these graphs have been obtained, the correspondences between the last version in the data model and the newly obtained graph must be determined.

Determining the correspondences between the last version and the new version is not that straightforward for all of the dependency graphs. For example, to determine the correspondence between two functions, it is possible to look at the name of the function, the class in which it is located, the file, etc. However, if the function's name has been changed, or when the function has been moved to another class, this method cannot be applied. In those cases, a more complex approach must be used. One such approach is described in [11].

It is also possible that an element that was present in the last version of the source code is no longer present in the new version or that a new element has been introduced that was not present in the last version. In both cases it is not possible to record any correspondence information for those elements.

## **7.2 Data visualisation**

Now that a data model has been proposed for the dependency analysis tool, an effective way to perform the analysis on this model is required. One of the essential parts of the analysis method is the ability to visualise the evolution of the dependency data set. This is a difficult problem since the data set is very complex and can also be quite large.

### **7.2.1 Visualisation technique**

For the visualisation of the dependency evolution, the call graph is chosen as an example. The visualisation is built on top of the already existing 2D layout of the Solid Trend Analyzer. In this layout, the x-axis represents the time, and the y-axis represents the files and every file is drawn as a rectangle.

The layout that is used in the Solid Trend Analyzer is not very well suited to display the call graph on, because one file can contain multiple function definitions and function calls can also be made to functions that reside in another file. For this reason, the call graph will not be displayed directly in the same layout as provided by the Solid Trend Analyzer, but in a separate view. A simple, yet effective example of how the visualisation might look like is shown in Figure 72.

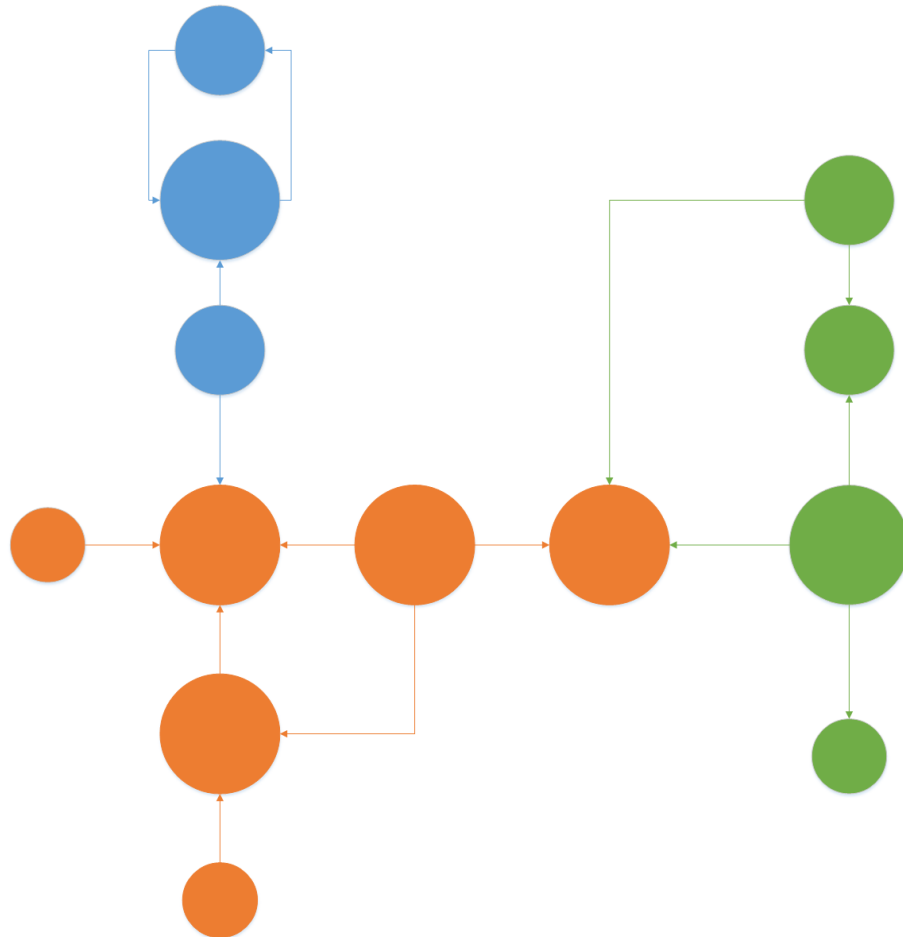


Figure 72: Visualisation of a call graph for the proposed visualisation technique

The technique that is used shows all of the functions that belong to the same class or file grouped together and also gives a colour coding per file or class. The directed edges between the nodes show which functions call what other functions. The size of the nodes also differ, nodes are drawn larger when many edges begin or end in that node. This way, important functions in the software are easily identified.

The scalability of the technique is reasonable. When the graph contains a very large amount of functions and function calls, the graph will contain many nodes and edges. While this will decrease the readability of the graph, the difference in node size helps the user with identifying what are probably the most important functions and the grouping of the nodes gives the user a better understanding of how the functions relate to each other.

The technique is only scalable if it is indeed possible to visualise the call graph in the way that was previously described. If the source code is structured in such a way that only a relatively small part of the functions is called by many of the other functions and that functions that reside within the same class or file mostly call each other, then it is believed that it is indeed possible to obtain a reasonable visualisation using this technique. However, when many functions in different files or classes call many other functions in other files or classes then there might be too many edges between the nodes to obtain a reasonable visualisation.

With a very large number of functions, the visualisation becomes somewhat cluttered. The functions are grouped per file or class, but when there are too many, the nodes are drawn quite close to the other nodes. This can make it somewhat hard to understand the graph. However, this problem can be somewhat mitigated by dynamically scaling the node size. When the number of nodes is very high, the size of the nodes can be decreased. This will then trade some of the readability for a decrease in the amount of cluttering, since a smaller node size results in more space that is available for spreading the nodes. The readability problem that arises from this can be solved by giving the user the possibility to zoom in on a certain part of the graph.

One improvement that could be implemented on top of this approach is the one shown in [10] where the nodes and edges are colour-coded based on the number of versions that a certain function remains untouched. This could also be used for the shown technique, to draw the attention of the user away from parts of the code that were not modified for a larger amount of time. Instead, the attention is drawn to the functions and calls that were recently introduced or modified.

### 7.2.2 Graph types

Several different types of dependency graphs are considered for the analysis of a source code repository. These graphs can be grouped into the following three categories: a cyclic graph, an acyclic graph and a tree. For the given dependency graphs, the following categorisation is made:

- The **call graph** shows which functions a particular function calls. Because a function can call other functions, can call itself and a function A that is called by a function B can also call that function A, the call graph is a cyclic graph.
- The **class inheritance graph** is an acyclic graph. The graph cannot contain cycles, since a class cannot inherit properties from a class that already inherits from that class. However, any class can inherit from any other class, therefore it is not a tree.
- The **containment graph** is a tree, because when an element A is contained in another element B, it is automatically contained into the elements that contain the element B. Also, element A cannot also be contained in another element that does not contain B.
- The **build dependency graph** is an acyclic graph. A file A can depend on a file B, but then B cannot depend on A. A file can also not depend on itself. And, as was the case with the inheritance graph, any file can depend on any other file.

When the type of the dependency graph is known, it is possible to use this knowledge for the visualisation of the software evolution. It is also possible to create a data model that takes advantage of the properties of that type of graph.

Knowing that the dependency graph that needs to be visualised is a tree has some advantages. A tree has a much clearer structure than the other graph types that were mentioned above. A tree has no cycles and a child node can have only one parent node. It is also possible to group nodes together when they have the same parent node.

### **7.3 Conclusion**

In this short essay, we have described the importance of the analysis of evolutions in dependency graphs next to the file-based software quality (and other) metrics provided by tools such as Solid Trend Analyzer. It is clear that both kind of analysis methods complement each other very well for the analysis of the maintainability, modularity, complexity and quality of software and development processes.

A data model was presented that attempts to capture the four types of dependency graphs, call graphs, inheritance graphs, containment graphs, and build dependency graphs, as well as their evolution between versions. Subsequently, we have described the advantages and disadvantages of this model and the design decisions that were taken.

Finally, we suggested a method for the visualisation of our proposed data model. This method displays nodes as dots which are coloured based on their membership to a class, file or package, and are varied in size based on their importance. Edges between these nodes indicate directed function calls. Although, it was difficult to visualise this type of graph on top of the view in Solid Trend Analyzer.

## 8 Evaluation

For this assignment, we have analysed the repository of the Blender project using several analyses in order to assess the maintainability, modularity, complexity and quality of the software, as well as the development process. We will now conclude this report.

### 8.1 Summary

In the first step, we have looked into the structure of the repository, defined the locations where most source code was located, showed the main contributors to the project for two definitions for midpoints of the project, and hypothesised that the amount of commits for the project develops erratically, with the majority lying in the second half of the project, rather than being evenly spread.

This hypothesis was confirmed in the second step, where we provided an initial visual overview of the project. The stable and unstable development periods were identified based on the hypothesis that these occurred when the sorted file view of the project shows respectively a low slope and a steeper slope. Other views seemed to indicate that this hypothesis was right, based on our definition of stable and unstable development periods. Furthermore, the data indicated that the code is currently in a reasonably stable state.

In the third step, analyses were performed regarding the authors/developers of the Blender project. We identified that there was indeed a chief developer for the project, but that in the past there have been different chief developers for limited periods of time. Several potential candidates for replacing the chief developer if need be have been determined through the evolution views of the commit count and the file views with the authors metric enabled. We have also shown that for some file types and a limited amount of folders, there seems to be a correlation between respectively file types and authors, and folders and authors.

The code size of the project was discussed in the fourth step. In particular, the evolution of the code size was thoroughly discussed. A naive approach was performed that did not give any valid insights in the size evolution, however, a secondary approach did. We hypothesised that the code size of files, on average, stays approximately the same over time in the project, which was highly supported by the obtained evolution views of groups of files. The files which shrink and grow the most have been identified in this section as well, and several methods were given for determining the fraction of source code in the entire project, for different definitions of both source code and what property defines that fraction.

In the fifth and last step regarding the Blender project, we analysed the complexity metrics. The currently most complex files, the files with the most increase and decrease in complexity, and the currently least complex files were determined in this section. Two hypotheses were formed in this section, namely that complexity and code size are directly related, and that activity and complexity are also largely directly related to each other. A number of figures were constructed that show support for these hypotheses, which indicates a high acceptance of these hypotheses.

In the last part of the document, we have described in an essay how we would propose to model various types of dependency graphs and how to visualise so that they may be used in the software analysis procedures. Advantages and disadvantages of our design decisions have been described, and we have argued why the visualisation method is suitable and applicable for displaying these kind of graphs.

## 8.2 Tools

In general, the tools that were provided and have been obtained and installed by ourselves proved to be easy to use, and we did not encounter any major problems. However, the CCCC calculator in SolidTA would often fail to complete within a reasonable time for certain revisions for certain files or just refuse to calculate the complexity/code size at all, seemingly at random. This could be due to the fact that the CCCC calculator uses a slightly older specification of the C and C++ language, which means that newer constructs may not be recognised and may cause failures or hangs of the calculator.

Furthermore, SolidTA lacks proper functionality for exporting views into images, which had to be done through screenshots. At the start of the course, in some occasions we also noticed that SolidTA would shutdown at random moments which led to a loss of data, since SolidTA is apparently not capable of retrieving calculated metrics that it has performed during execution runs after which the application is not properly closed. This did not seem to happen at all later on in the course, we are unsure what the reason for this behaviour was.

For a new version, pulling the file and revision list in SolidTA is fairly quick. However, when trying to obtain the contents of revisions in the new version, SolidTA may attempt to loop over all files to check if it has fetched the content of those files. This leads to a longer execution time of simply fetching the new files than is necessary.

Finally, SolidTA has some bugs related to the user interface. When the application is left in a minimised state for extended periods of time, errors will be displayed routinely in the log window, and functionality of the interface will fail or lag. When the configuration slider for a metric is adjusted while the trend view for that metric is open, errors will be displayed and the trend view will seize updating, which can be fixed by disabling the trend view and reopening it, but is still rather annoying. Scrolling through the file list can be very laggy in the application as well.



### 8.3 Analysis

The analysis of the software took more time than we initially planned it would, due to the fact that we wanted to give a very detailed analysis in all of the different steps. We hope that this effort is observed and appreciated by the reader of this document.

During the lectures, Alexandru Telea remarked on the fact that it would be infeasible to fetch all of the contents and calculate all of the metrics for a specific repository. However, now that we have performed this assignment, we can say that this was not a problem for us personally, since (apart from the issues described above) we could let SolidTA do the calculation and file fetching over extended periods of time in the background while performing other tasks.

Regarding the analysis of metrics itself, it was relatively easy to use the metrics in SolidTA and the corresponding file, evolution and trend views to come to conclusions on the questions that were given for the assignment.

### 8.4 Conclusion

We experienced this assignment as very insightful and enjoyable, due to the fact that it is very practical, that we can pick our own, real-life software repository to investigate, and to actually draw conclusions on this real-life data based on the various quality metrics, developers and other metrics such as activity and file types/locations. It was interesting to play around with the different settings in SolidTA, and to see how SolidTA has indeed widened and deepened our understanding of the repository while using it for this assignment.

All in all, we hope that the reader appreciates the effort put in this assignment and this document, and appreciates the level of detail that we have tried to maintain throughout the assignment.

## 9 Acknowledgements

We would like to extend our gratitude to Alexandru Telea for his teaching efforts in the course of Software Maintenance and Evolution that we attended, and for reviewing this report.

## 10 References

- [1] Blender Foundation and the Blender Project: <http://www.blender.org/>
- [2] "Developers: Ask Us Anything!", an article on the Blender wiki:  
<http://wiki.blender.org/index.php/Dev:Doc/AskUsAnything>
- [3] Index of /source/, on Blender Project sitemap:  
<http://download.blender.org/source/>
- [4] "Blender Dokumentation: Die Geschichte von Blender" on WikiBooks:  
[https://de.wikibooks.org/wiki/Blender\\_Dokumentation:\\_Die\\_Geschichte\\_von\\_Blender](https://de.wikibooks.org/wiki/Blender_Dokumentation:_Die_Geschichte_von_Blender)
- [5] "Blender 2.76.2 8d9c7e6 - API documentation", available on:  
[http://www.blender.org/api/blender\\_python\\_api\\_2\\_76\\_2](http://www.blender.org/api/blender_python_api_2_76_2)
- [6] "Blender (software)", an article on Wikipedia:  
[https://en.wikipedia.org/wiki/Blender\\_\(software\)](https://en.wikipedia.org/wiki/Blender_(software))
- [7] Blender 2.76b Release Notes: <http://www.blender.org/features/2-76/>
- [8] Release Cycle of Blender project: [http://wiki.blender.org/index.php/Dev:Doc/Process/Release\\_Cycle](http://wiki.blender.org/index.php/Dev:Doc/Process/Release_Cycle)
- [9] Top developers of 2015 of the Blender project:  
<https://www.miikahweb.com/en/blender/git-statistics/year/2015>
- [10] Collberg C., Kobourov S., Nagra J., Pitts J., Wampler K. (2003), *A System for Graph-Based Visualization of the Evolution of Software*, Proceedings of the 2003 ACM symposium on Software visualization, ACM, p. 77-ff.
- [11] Chevalier F., Auber D., Telea A. (2007), *Structural analysis and visualization of C++ code evolution using syntax trees*, Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting, ACM, 2007. p. 90-97.

## 11 Appendices

The following is a set of listings and images that were provided as further clarification on the content of this document and the steps that were performed. Please refer to the text of the document for references and explanation on these items.

### 11.1 Time Tracking

We each spent around the following amount of hours on each activity item: (excluding passive items)

Activity	Time spent
Installing SolidTA	30m
Picking repository	4h
Installing git command line tool	5m
Installing Filelight	10m
Getting familiar with SolidTA	2h
Getting familiar with Filelight	15m
Refreshing git command knowledge	1h
Downloading file revisions and contents	10h (passive)
Calculating Lines of Text for all revisions	2h (passive)
Performing CCCC calculator on all revisions	20h (mostly passive)
Performing step 1	8h
Performing step 2	5h
Performing step 3	9h
Performing step 4	7h
Performing step 5	8h
Performing step 6	10h
Documenting	25h
<b>Total:</b>	80h

Table 3: Distribution of files and sizes among top-level directories, including only source code files.

## 11.2 File extensions

Extension	Files	Size (B)	Extension	Files	Size (B)
<b>.h</b>	<b>2866</b>	<b>20121023</b>	<b>.cpp</b>	<b>1224</b>	<b>20283612</b>
<b>.c</b>	<b>1203</b>	<b>34664636</b>	.dat	962	2485808
<b>.py</b>	<b>500</b>	<b>3309310</b>	<b>.cc</b>	<b>272</b>	<b>2595935</b>
.txt	212	975956	<b>.hpp</b>	<b>116</b>	<b>725040</b>
<b>.rst</b>	<b>95</b>	<b>563089</b>	<b>.osl</b>	<b>86</b>	<b>121949</b>
.spild	73	8640162	.png	45	973353
.cmake	35	133868	.jpg	24	768388
<b>.gls</b>	<b>23</b>	<b>108853</b>	.patch	20	35031
.sh	19	181997	<b>.cl</b>	<b>13</b>	<b>61496</b>
<b>.inl</b>	<b>12</b>	<b>86516</b>	.xml	10	495966
<b>.mm</b>	<b>7</b>	<b>127153</b>	.spimtx	6	589
.spi3d	5	4510443	.html	5	48589
.svg	4	5361844	.icns	4	669866
.ico	4	347184	.blend	3	2271140
.bmp	3	182970	.libmv	3	978
.map	3	615	.gz	2	5109341
.ttf	2	501120	.cfg	2	41641
<b>.m</b>	<b>2</b>	<b>40452</b>	.css	2	5588
.plist	2	3058	.conf	2	873
.xcf	1	1274820	.bz2	1	182524
.3dl	1	78736	.GPL3	1	35147
.ocio	1	27109	.pfb	1	25181
.GPL2	1	19049	.nsi	1	8599
<b>.hh</b>	<b>1</b>	<b>6299</b>	<b>.cu</b>	<b>1</b>	<b>6181</b>
.desktop	1	5589	.LZO	1	4003
.in	1	3106	.rc	1	1057
.mac	1	1003	<b>.js</b>	<b>1</b>	<b>918</b>
.linux	1	625	.cygwin	1	596
.FFT	1	509	.gitmodules	1	435
.gitignore	1	406	.manifest	1	395
.org	1	391	.install	1	306
.arcconfig	1	170			

Table 4: List of file extensions in the entire repository, sorted primarily on Files and secondarily on Size, read from left to right from top to bottom. Files without an extension are not included in this list. Source code extensions are depicted in bold.

Extension	Files	Size (B)	Extension	Files	Size (B)
<b>.h</b>	<b>1396</b>	<b>5982124</b>	<b>.c</b>	<b>1111</b>	<b>31130875</b>
<b>.cpp</b>	<b>729</b>	<b>6006631</b>	.txt	98	288715
<b>.gsl</b>	<b>21</b>	<b>96015</b>	<b>.cc</b>	<b>16</b>	<b>240596</b>
<b>.py</b>	<b>7</b>	<b>34012</b>	.ico	2	320820
<b>.m</b>	<b>2</b>	<b>40452</b>	.map	2	582
<b>.cl</b>	<b>1</b>	<b>11530</b>	.rc	1	1057
.sh	1	672	.manifest	1	395
.conf	1	41			

Table 5: List of file extensions in the source folder, sorted primarily on Files and secondarily on Size, read from left to right from top to bottom. Files without an extension are not included in this list. Source code extensions are depicted in bold.

Extension	Files	Size (B)	Extension	Files	Size (B)
<b>.h</b>	<b>888</b>	<b>10769847</b>	<b>.cc</b>	<b>230</b>	<b>2139536</b>
<b>.cpp</b>	<b>185</b>	<b>2636876</b>	<b>.hpp</b>	<b>78</b>	<b>550238</b>
<b>.c</b>	<b>42</b>	<b>3060367</b>	.txt	38	182301
.patch	18	34082	.sh	11	35514
<b>.py</b>	<b>5</b>	<b>38925</b>	.libmv	3	978
.GPL3	1	35147	.GPL2	1	19049
<b>.hh</b>	<b>1</b>	<b>6299</b>	.LZO	1	4003
.org	1	391	.map	1	33

Table 6: List of file extensions in the extern folder, sorted primarily on Files and secondarily on Size, read from left to right from top to bottom. Files without an extension are not included in this list. Source code extensions are depicted in bold.

Extension	Files	Size (B)	Extension	Files	Size (B)
<b>.h</b>	<b>576</b>	<b>3288276</b>	<b>.cpp</b>	<b>310</b>	<b>11640105</b>
<b>.osl</b>	<b>80</b>	<b>114496</b>	<b>.c</b>	<b>49</b>	<b>464781</b>
.txt	49	180935	<b>.hpp</b>	<b>38</b>	<b>174802</b>
<b>.cc</b>	<b>12</b>	<b>149420</b>	<b>.inl</b>	<b>12</b>	<b>86516</b>
<b>.cl</b>	<b>12</b>	<b>49966</b>	<b>.py</b>	<b>8</b>	<b>115201</b>
<b>.mm</b>	<b>7</b>	<b>127153</b>	<b>.gsl</b>	<b>2</b>	<b>12838</b>
.cfg	1	28173	<b>.cu</b>	<b>1</b>	<b>6181</b>
.cmake	1	4189	.mac	1	1003
.linux	1	625	.cygwin	1	596
.FFT	1	509			

Table 7: List of file extensions in the intern folder, sorted primarily on Files and secondarily on Size, read from left to right from top to bottom. Files without an extension are not included in this list. Source code extensions are depicted in bold.

Extension	Files	Size (B)	Extension	Files	Size (B)
.dat	962	2485808	<b>.py</b>	<b>354</b>	<b>2499242</b>
.spild	73	8640162	.png	41	916962
.jpg	24	768388	.xml	10	495966
.txt	10	75626	<b>.osl</b>	<b>6</b>	<b>7453</b>
.spimtx	6	589	.spi3d	5	4510443
.svg	4	5361844	.icns	4	669866
.blend	3	2271140	.bmp	3	182970
.gz	2	5109341	.ttf	2	501120
.plist	2	3058	.xcf	1	1274820
.bz2	1	182524	.3dl	1	78736
.ocio	1	27109	.ico	1	25214
.pfb	1	25181	<b>.c</b>	<b>1</b>	<b>8613</b>
.nsi	1	8599	.html	1	6223
.desktop	1	5589			

Table 8: List of file extensions in the release folder, sorted primarily on Files and secondarily on Size, read from left to right from top to bottom. Files without an extension are not included in this list. Source code extensions are depicted in bold.

11.3 Revisions of potential replacements for chief developer

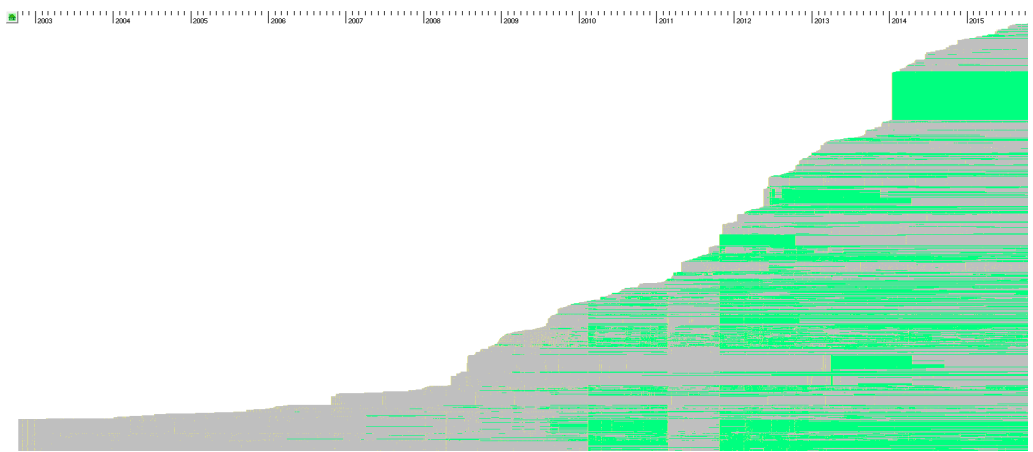


Figure 73: Revisions of Campbell Barton (current chief developer)

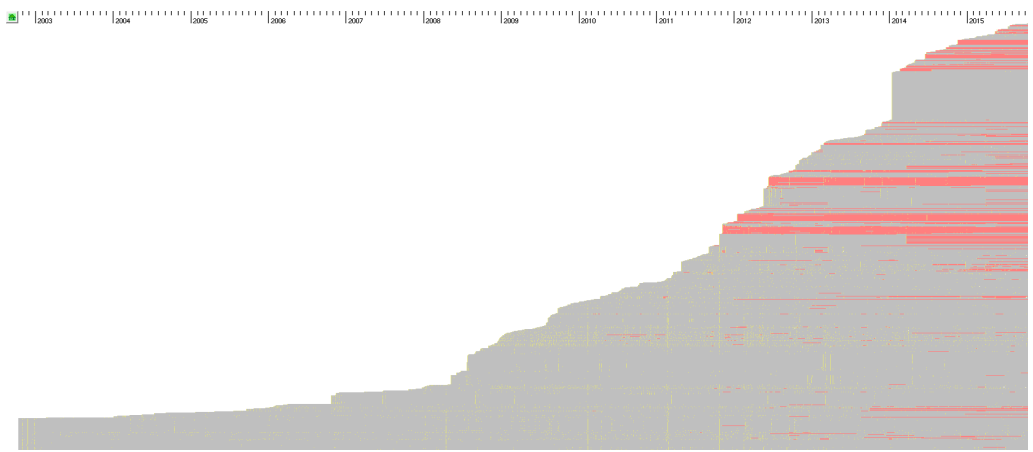


Figure 74: Revisions of Sergey Sharybin

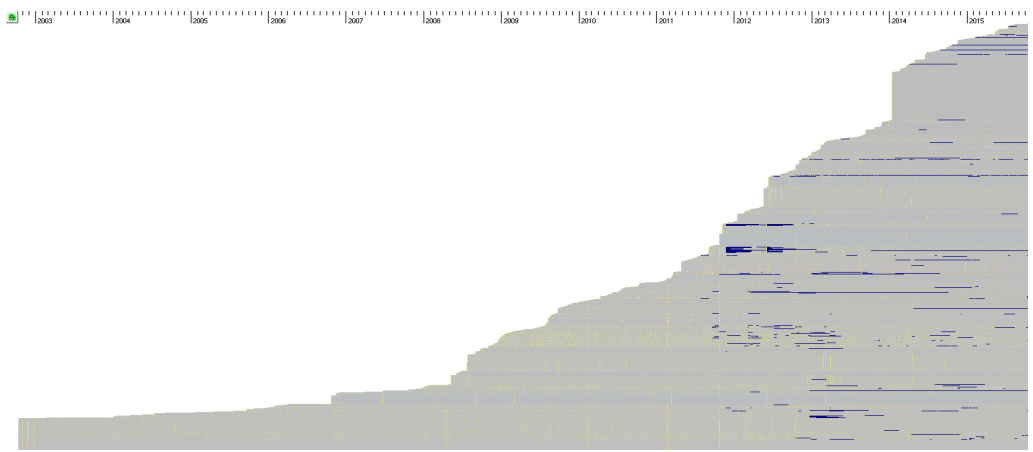


Figure 75: Revisions of Bastien Montagne

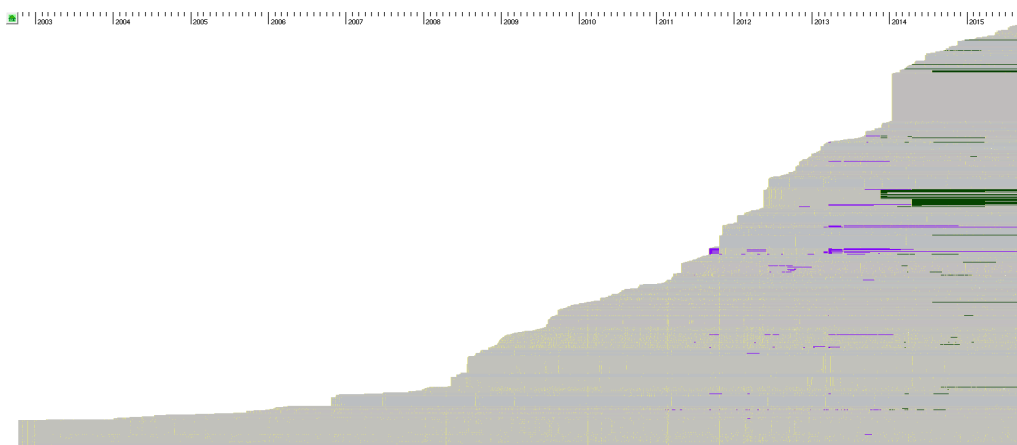


Figure 76: Revisions of Lukas Toenne

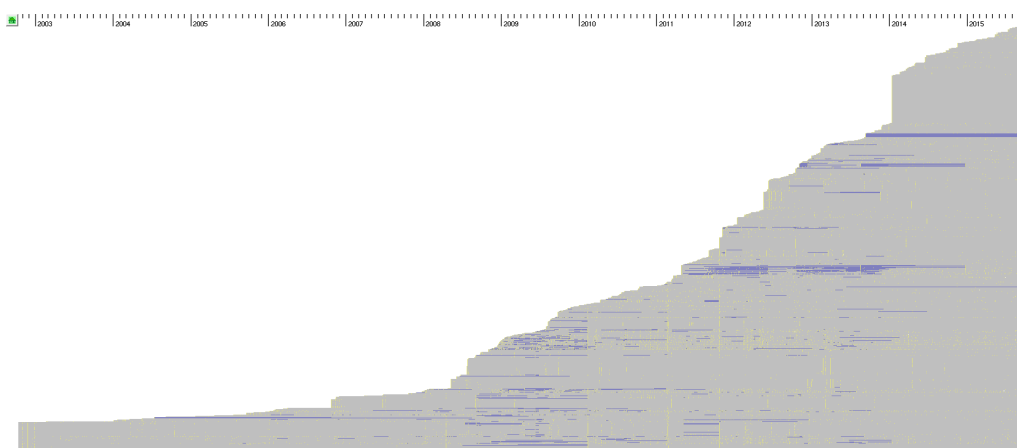


Figure 77: Revisions of Brecht van Lommel

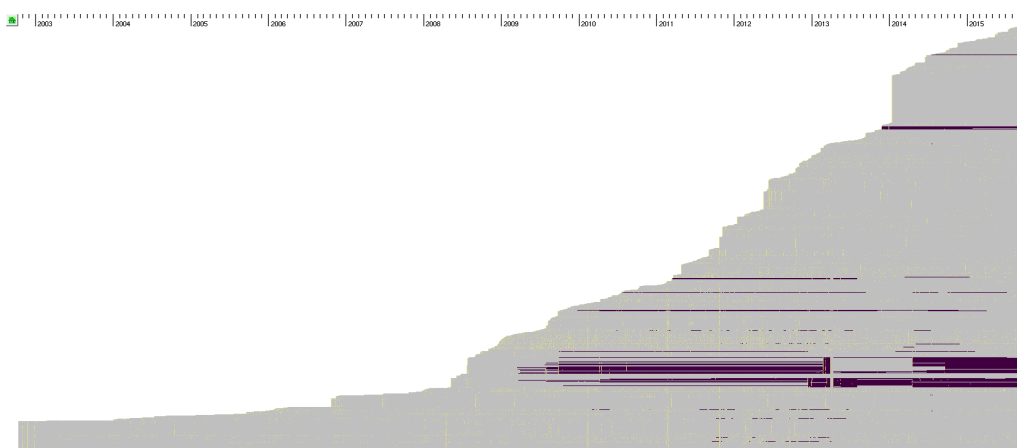


Figure 78: Revisions of Tamito Kajiya



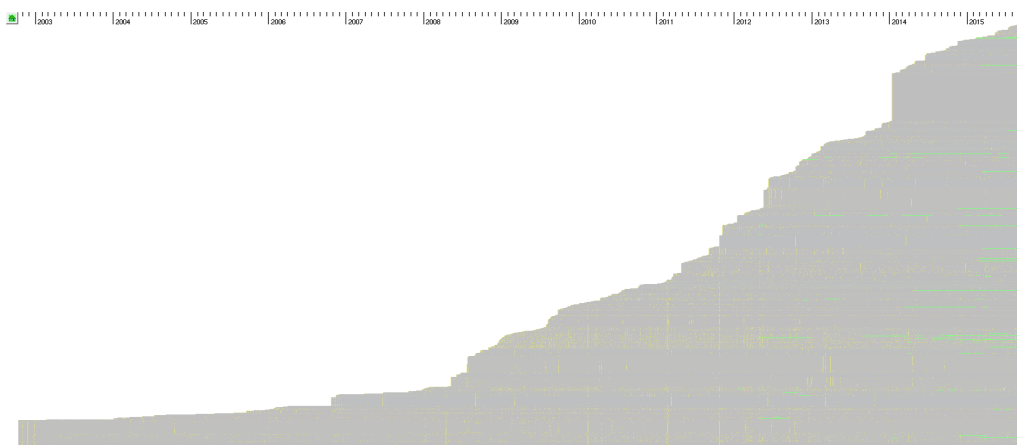


Figure 79: Revisions of Antony Riakiotakis

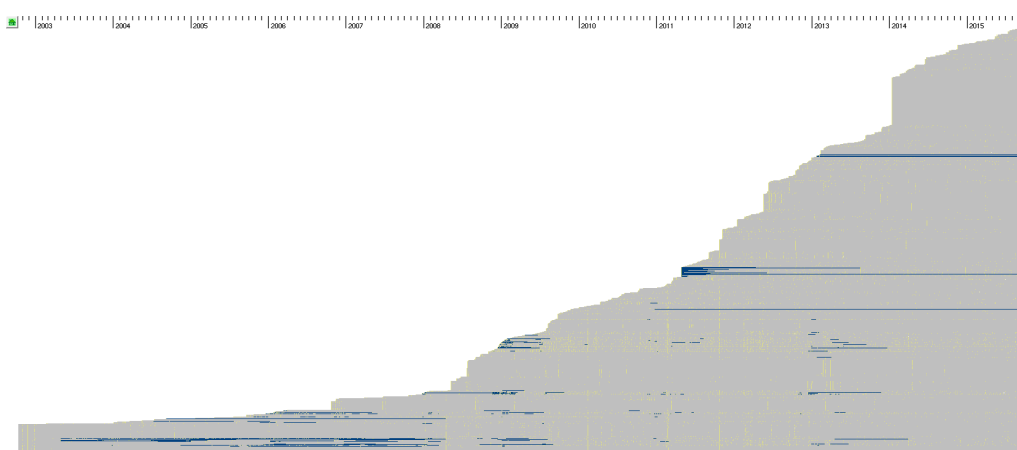


Figure 80: Revisions of Ton Roosendaal

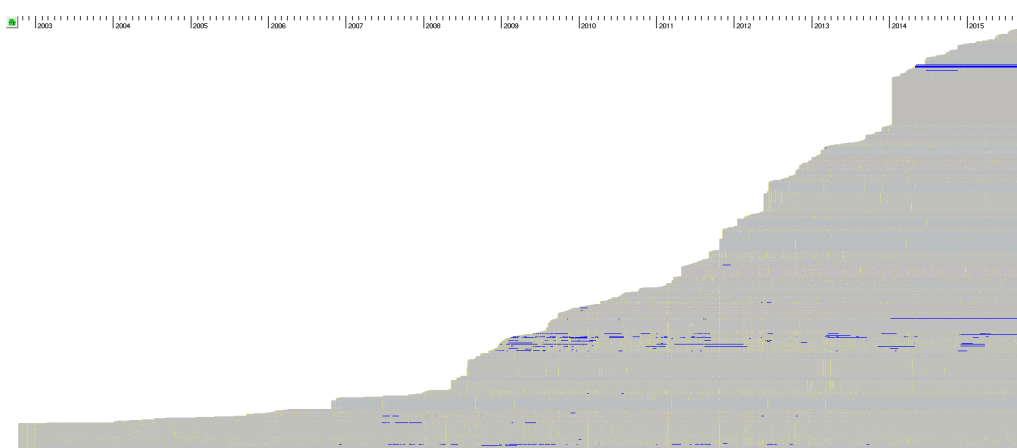


Figure 81: Revisions of Joshua Leung

# 11.4 Details of authors metric per file type

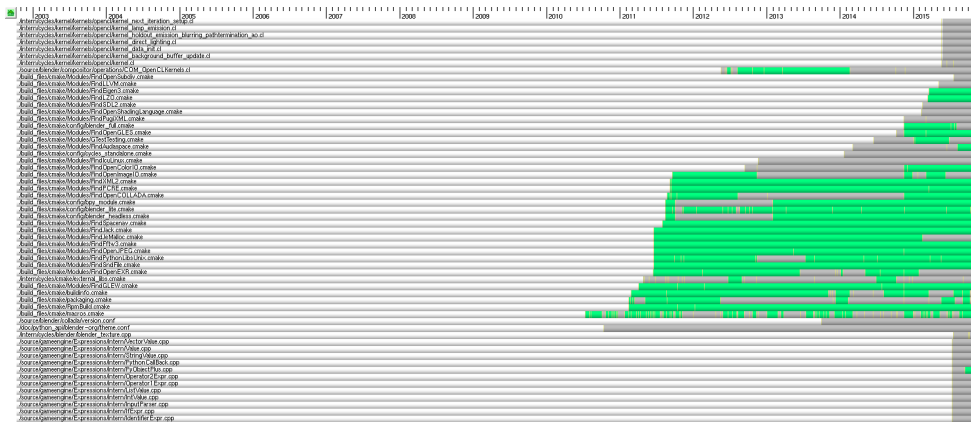


Figure 82: Detail of authors metric for .cmake files, with Campbell Barton highlighted as the only author.



Figure 83: Detail of authors metric for .jpg files, with Ton Roosendaal highlighted as the only author.

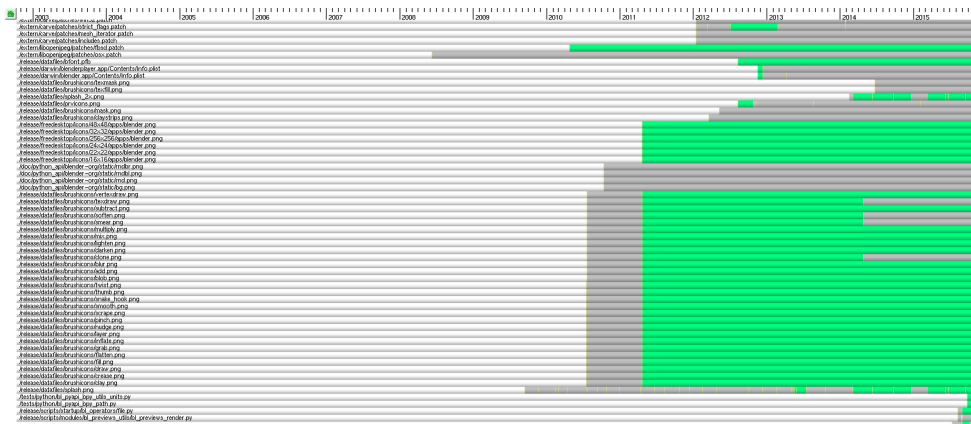


Figure 84: Detail of authors metric for .png files, with Campbell Barton highlighted as the only author.

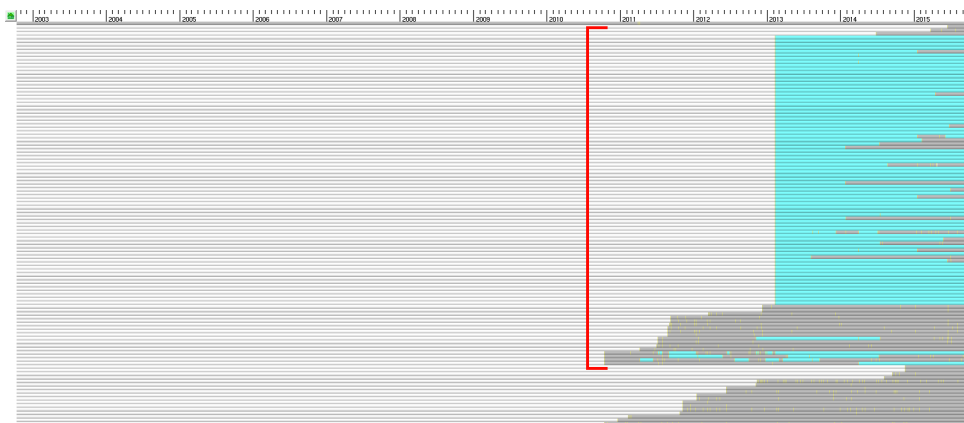


Figure 85: Detail of authors metric for .rst files, with Mitchel Stokes highlighted as the only author.

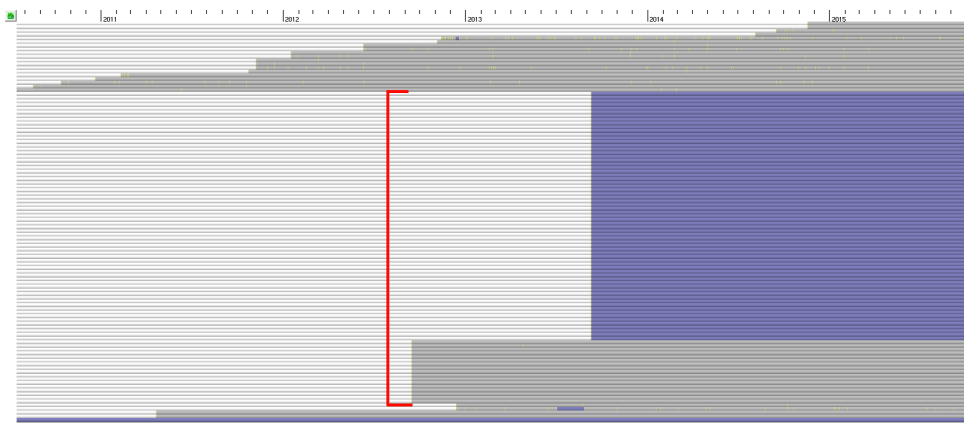


Figure 86: Detail of authors metric for .spild files, with Brecht van Lommel highlighted as the only author.



Figure 87: Detail of authors metric for .osl files, with Brecht van Lommel and Thomas Dinges highlighted as authors.

11.5 Details of authors metric per (sub-)directory



Figure 88: Detail of authors metric for /doc folder.

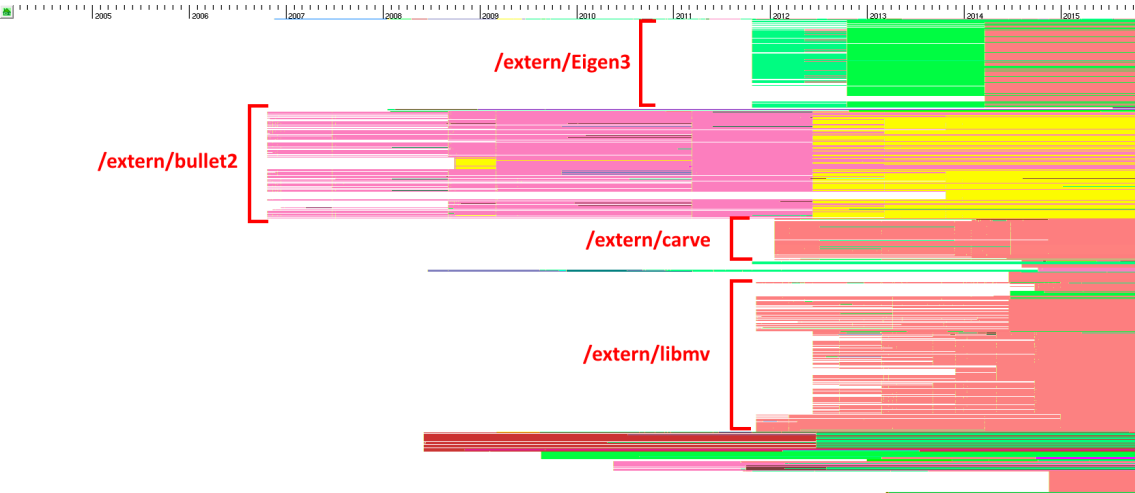


Figure 89: Detail of authors metric for /extern folder.

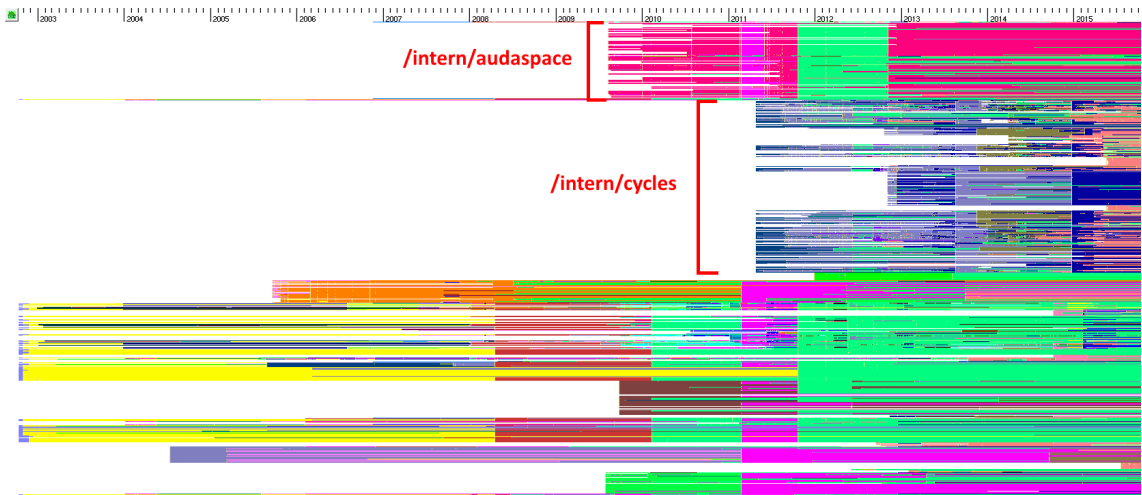


Figure 90: Detail of authors metric for /intern folder.

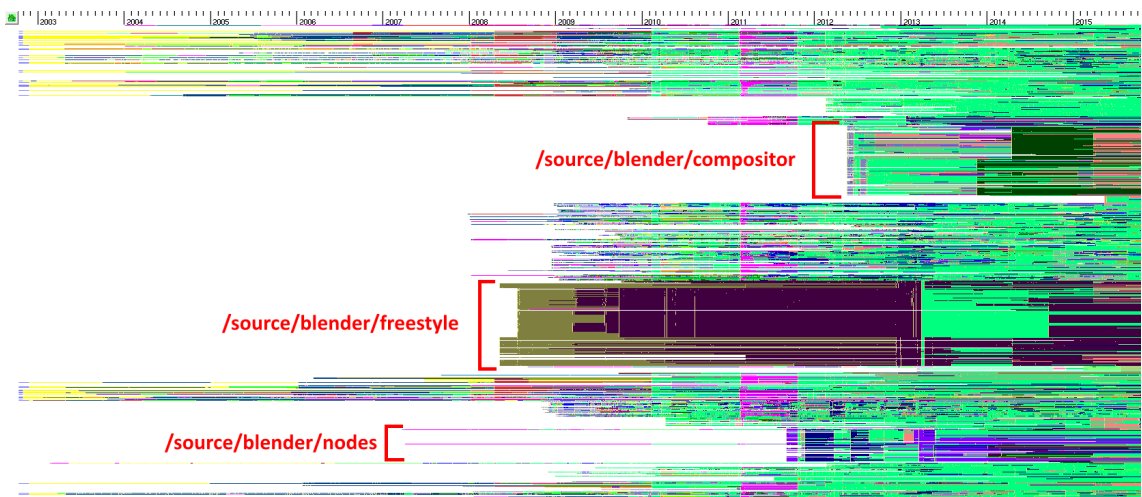


Figure 91: Detail of authors metric for the /source/blender folder.

## 11.6 Relevant file views

Listing 13: Files that grow/shrink the most in terms of LOC (60 files)

```
/intern/guardedalloc/intern/mallocn.c
/source/blender/blenkernel/intern/deform.c
/source/blender/blenkernel/intern/mball.c
/source/blender/blenkernel/intern/nla.c
/source/blender/blenkernel/intern/subsurf_ccg.c
/source/blender/blenkernel/intern/node.c
/extern/bullet2/src/BulletCollision/CollisionDispatch/btCollisionWorld.
cpp
/extern/bullet2/src/BulletDynamics/ConstraintSolver/btHingeConstraint.
cpp
/extern/bullet2/src/BulletDynamics/ConstraintSolver/
btSequentialImpulseConstraintSolver.cpp
/extern/bullet2/src/BulletDynamics/Dynamics/btDiscreteDynamicsWorld.cpp
/extern/bullet2/src/BulletDynamics/ConstraintSolver/
btConeTwistConstraint.cpp
/source/blender/blenkernel/intern/shrinkwrap.c
/source/blender/blenlib/intern/BLI_kdopbv.c
/source/blender/blenkernel/intern/bvhutils.c
/source/blender/makesrna/intern/rna_object.c
/source/blender/makesrna/intern/rna_scene.c
/source/blender/makesrna/intern/rna_ID.c
/source/blender/makesrna/intern/rna_mesh.c
/source/blender/makesrna/intern/rna_wm.c
/source/blender/makesrna/intern/rna_material.c
/source/blender/makesrna/intern/rna_nodetree.c
/source/blender/makesrna/intern/rna_sensor.c
/source/blender/makesrna/intern/rna_color.c
/source/blender/makesrna/intern/rna_actuator.c
/source/blender/makesrna/intern/rna_brush.c
/source/blender/makesrna/intern/rna_modifier.c
/source/blender/editors/interface/interface_handlers.c
/source/blender/editors/space_view3d/view3d_header.c
/source/blender/makesrna/intern/rna_constraint.c
/source/blender/editors/space_view3d/drawobject.c
/source/blender/editors/animation/anim_filter.c
/source/blender/editors/mesh/editmesh_add.c
/source/blender/editors/mesh/editmesh_tools.c
/source/blender/makesrna/intern/rna_object_force.c
/source/blender/makesrna/intern/rna_particle.c
/source/blender/makesrna/intern/rna_userdef.c
/source/blender/makesrna/intern/rna_texture.c
/source/blender/blenkernel/intern/fcurve.c
/source/blender/editors/sculpt_paint/paint_ops.c
/source/blender/makesrna/intern/rna_ui.c
/source/blender/editors/gpencil/gpencil_edit.c
/source/blender/editors/object/object_modifier.c
/source/blender/editors/space_image/image_buttons.c
/source/blender/editors/space_nla/nla_edit.c
/source/blender/editors/space_buttons/buttons_ops.c
/source/blender/makesrna/intern/rna_main_api.c
```

```

/source/blender/modifiers/intern/MOD_edgesplit.c
/source/blender/blenkernel/intern/linestyle.c
/source/blender/makesrna/intern/rna_linestyle.c
/source/blender/modifiers/intern/MOD_skin.c
/intern/audaspace/intern/AUD_JOSResampleReader.cpp
/source/blender/nodes/composite/node_composite_util.c
/extern/libmv/third_party/gflags/gflags.cc
/source/blender/nodes/composite/nodes/node_composite_doubleEdgeMask.c
/source/blender/bmesh/operators/bmo_bevel.c
/source/blender/bmesh/operators/bmo_create.c
/source/blender/python/bmesh/bmesh_py_types_customdata.c
/source/blender/blenkernel/intern/mask.c
/source/blender/blenkernel/intern/mask_rasterize.c
/source/blender/physics/intern/implicit_eigen.cpp

```

Listing 14: Files that increase most in complexity (63 files)

```

/source/blender/blenkernel/intern/mball.c
/source/blender/blenkernel/intern/screen.c
/source/blender/blenkernel/intern/subsurf_ccg.c
/intern/elbeem/intern/ntl_vector3dim.h
/source/blender/blenkernel/intern/node.c
/extern/bullet2/src/BulletCollision/CollisionDispatch/btCollisionWorld.
cpp
/extern/bullet2/src/BulletDynamics/ConstraintSolver/btHingeConstraint.
cpp
/source/blender/blenkernel/intern/bvhutils.c
/source/blender/makesrna/intern/rna_object.c
/source/blender/makesrna/intern/rna_ID.c
/source/blender/makesrna/intern/rna_mesh.c
/source/blender/makesrna/intern/rna_wm.c
/source/blender/makesrna/intern/rna_material.c
/source/blender/makesrna/intern/rna_nodetree.c
/source/blender/python/intern/bpy_interface.c
/source/blender/makesrna/intern/rna_color.c
/source/blender/makesrna/intern/rna_actuator.c
/source/blender/makesrna/intern/rna_brush.c
/source/blender/makesrna/intern/rna_modifier.c
/source/blender/makesrna/intern/rna_curve.c
/source/blender/makesrna/intern/rna_armature.c
/source/blender/editors/space_view3d/view3d_header.c
/source/blender/makesrna/intern/rna_constraint.c
/source/blender/editors/space_image/space_image.c
/source/blender/editors/space_node/space_node.c
/source/blender/editors/space_buttons/space_buttons.c
/source/blender/editors/space_file/space_file.c
/source/blender/editors/space_file/filesel.c
/source/blender/editors/space_action/space_action.c
/source/blender/editors/space_nla/space_nla.c
/source/blender/editors/space_sequencer/space_sequencer.c
/source/blender/editors/animation/anim_deps.c
/source/blender/editors/mesh/editmesh_tools.c
/source/blender/makesrna/intern/rna_object_force.c
/source/blender/makesrna/intern/rna_particle.c

```

```

/source/blender/makesrna/intern/rna_userdef.c
/source/blender/makesrna/intern/rna_texture.c
/source/blender/makesrna/intern/rna_pose.c
/source/blender/makesrna/intern/rna_space.c
/source/blender/editors/ sculpt_paint/paint_ops.c
/source/blender/makesrna/intern/rna_ui.c
/source/blender/editors/object/object_modifier.c
/source/blender/editors/space_nla/nla_edit.c
/source/blender/makesrna/intern/rna_main_api.c
/source/blender/makesrna/intern/rna_object_api.c
/source/blender/makesrna/intern/rna_ui_api.c
/source/blender/makesrna/intern/rna_sculpt_paint.c
/source/blender/blenkernel/intern/paint.c
/source/blender/modifiers/intern/MOD_smoke.c
/source/blender/blenkernel/intern/linestyle.c
/source/blender/makesrna/intern/rna_linestyle.c
/intern/cycles/kernel/svm/svm_tex_coord.h
/intern/cycles/render/film.cpp
/source/blender/modifiers/intern/MOD_skin.c
/extern/Eigen3/Eigen/src/Core/PlainObjectBase.h
/extern/libmv/third_party/gflags/gflags.cc
/source/blender/python/mathutils/mathutils_noise.c
/source/blender/python/bmesh/bmesh_py_types_customdata.c
/source/blender/blenkernel/intern/mask_rasterize.c
/source/blender/editors/space_node/node_view.c
/source/blender/python/bmesh/bmesh_py_ops_call.c
/source/blender/bmesh/operators/bmo_edgenet.c
/source/blender/physics/intern/BPH_mass_spring.cpp

```

Listing 15: Files that decrease most in complexity (30 files)

```

/intern/guardedalloc/intern/mallocn.c
/intern/string/STR_String.h
/extern/bullet2/src/BulletDynamics/Dynamics/btDiscreteDynamicsWorld.cpp
/extern/bullet2/src/BulletDynamics/ConstraintSolver/
    btConeTwistConstraint.cpp
/source/blender/blenlib/intern/BLI_kdopbv.c
/source/blender/blenlib/intern/BLI_mempool.c
/source/blender/blenkernel/intern/fluidsim.c
/source/blender/makesrna/intern/rna_scene.c
/source/blender/editors/interface/interface_handlers.c
/source/blenderplayer/bad_level_call_stubs/stubs.c
/source/blender/render/intern/raytrace/rayobject.cpp
/source/blender/freestyle/intern/view_map/BoxGrid.h
/intern/cycles/device/device.cpp
/intern/cycles/device/device_cuda.cpp
/intern/cycles/kernel/svm/svm_math.h
/source/blender/nodes/composite/node_composite_util.c
/source/blender/nodes/composite/nodes/node_composite_blur.c
/source/blender/nodes/composite/nodes/node_composite_defocus.c
/source/blender/nodes/composite/nodes/node_composite_glare.c
/intern/dualcon/intern/Projections.h
/source/blender/nodes/composite/nodes/node_composite_doubleEdgeMask.c
/source/blender/bmesh/operators/bmo_bevel.c

```



```
/source/blender/bmesh/operators/bmo_mesh_conv.c
/source/blender/blenkernel/intern/mask.c
/extern/libmv/third_party/ceres/include/ceres/jet.h
/source/blender/bmesh/operators/bmo_symmetrize.c
/source/blender/bmesh/operators/bmo_unsubdivide.c
/source/blender/bmesh/operators/bmo_beautify.c
/extern/libmv/third_party/ceres/internal/ceres/blas.h
/source/blender/physics/intern/implicit_eigen.cpp
```