Politecnico di Torino

Politecnico di Torino

Ingegneria Informatica

A.a. 2022/2023

Sessione di laurea July 2023

Understanding Creative Coding characteristics

a large-scale scraping and analysis of open-source projects

Relatori:

Luigi De Russis

Juan Pablo Saenz Moreno

Candidato:

Shantal Fabri

Table of Contents

List of Figures

Chapter 1

Introduction

Creative coding is an application of computer programming where the goal is to create something expressive or artistic. Through the use of software, code and computational processes, it aims to create results that are not necessarily predefined and rather based on discovery, variation and exploration that can sometimes produce unexpected results.[1, 2]

Even when the goal of creative coding is not really to create something strictly functional, that does not mean there aren't any applications or functions to it. Since the origin of the practice of creating art through coding during the 1960s, people have found uses to it, such as creation of live visuals for VJing [3] (VeeJay-ing, from Video Jockey), which includes the 'creation or manipulation of imagery in realtime through technological mediation and for an audience, in synchronization to music' [4], projections and projection mapping, art installations, entertainment such as video games, sound art, etc. Lately, its use is becoming increasingly common in fields like advertising, branding and the design industry in general, where even sometimes mass production of various designs for products is assisted by creative coding. [1]

The growth in its popularity and applicability makes creative coding a relevant and very interesting field to look at, analyze and learn more about. And, moreover, its existence also goes to show the capabilities of computers that go beyond functional purposes.

Nowadays, creators publish and share their creative coding projects online. Many artists have their own websites where they portray their projects that use and apply creative coding. But what a lot of creators are doing nowadays, is publish and share their creative coding projects on open source platforms. One of the main platforms where this is done is *OpenProcessing*, a website that hosts over one

million projects and invites creative coders, educators and designers to explore, experiment and play, and where creators can share their projects as open source and collaborate with the community. Each project can be shared, downloaded, liked and commented on by other users of the website, who can also fork the projects to further work on them or add their own twists to what other creators have done.

1.1 Goal

The goal of this thesis is to better understand the state of the art and the current way creative coding is being done. The objective is to understand and characterize what it is that creators are doing and how they are creating, what programming languages they are using, how they are structuring their code and projects, see if there are common patterns between different creative coding projects, what these patterns are and how many share them, etc. After this analysis, the goal is to evaluate how well these creators are programming based on measurements like lines of code (LOC), lines of comments, amount of files per project, complexity of projects as a whole and of functions, parameters used in functions, variation of functions, and other maintainability and complexity indexes.

To accomplish these goals, a quantitative analysis was conducted over a set of thirty thousand projects publicly available on *OpenProcessing*. Static code analyses were conducted on the projects and the whole data set to provide insights on the various source code metrics mentioned above.

Lastly, the discussion and understanding on how artists are working and creating in this field provides insights as to what new tools could be useful for them to further explore and create with the assistance of coding.

1.2 Thesis structure

This thesis is composed of three main parts:

- Scraping; why it was done; how it was done; results.
- Analysis; how it was done; factors analyzed; results.
- Discussion; discussion of the results; conclusions.

Chapter 2

Scraping

Web scraping, or scraping for short, is the practice of extracting or "scraping" data from the web. Even though web scraping can be done manually, the term usually refers to the use of automation tools to collect data from the web, be it a software script that simulates a human's interaction with the website, a browser extension or a different tool. It essentially is a form of gathering and copying data from the web, into a local or central database where it can later be retrieved from or analyzed. [5, 6, 7, 8]

Although to some people the idea of scraping the web might seem unethical or even illegal, the reality is that as long as the data to be retrieved from the web is of public access, they are free to be scraped. [5]

The applications of web scraping reach a big range of sectors with many different purposes. The different sectors that consume web data go from real estate, travel agencies, recruitment firms to research, e-commerce industry being the biggest consumer. The uses for web scraping include, news and other content scraping, contact scraping, research, price comparison, market study, brand monitoring, website change detection, among others. [6, 7]

The reason why scraping was necessary and how it was done for this thesis are discussed in the following sections, along with the results of the process.

2.1 Goal

In the case of this thesis, the use of web scraping that was applied is research, with the purpose of better understanding the state of the art of creative coding and mainly to understand how creating coding is being done and how artists are using

it to create.

In order to achieve a proper analysis, and to have a good representation of creative coding projects, a big enough set of data was needed. In this case, an amount of projects to be analyzed was expected to be in the order of thousands, and since this was a task that is essentially impossible to accomplish manually, another way had to be found.

The idea of getting the data needed directly from the database where the projects available in the website *OpenProcessing* are kept was the first option considered. After some research and contacting the people involved in the maintenance and creation of the website, the conclusion that doing such a thing was not possible was reached, given the lack of a public API that could provide the desired data. For this reason, the need of web scraping came to light as the indicated method to obtain the needed data.

2.2 Tools and procedure

The scraping, as mentioned before, was done over the website *OpenProcessing*, with a script that works directly on a web browser and simulates what a person would do while navigating the site. In this case, the script was written in the programming language python and the library Selenium [9] was used to perform the different actions on the website and everything related to the scraping itself, and it was programmed to run on the web browser Firefox.

In *OpenProcessing*, the projects are called sketches so this term will be used from now on interchangeably with creative coding projects, along with the term project.

Given how the website is organized and how the list of all sketches can be accessed, the whole process had to be run in two phases. The first phase consisted on collecting the links to the individual projects, and the second consisted in downloading the projects themselves among some other data related to them. Both these phases will be explained in detail in the following sections.

2.2.1 First phase: link collection

As previously stated, the first step to actually obtain the source code of the projects to analyze themselves was to obtain the links that refer to where the projects can be downloaded from.

The 'browse' tab of *OpenProcessing* displays a grid, initially of 10 rows of 12 squares, each showcasing a thumbnail of a project and when hovered over, its creator-given name and creator (see figure 2.1). In order to see an expanded list of sketches, the button "Show more" must be pressed, which loads an extra 10 rows of sketches. Each of these squares, when clicked, guides you to the page of an individual sketch, which is where the source code can be downloaded. These links are what needed to be collected in mass to then access and download the source code of the sketches.

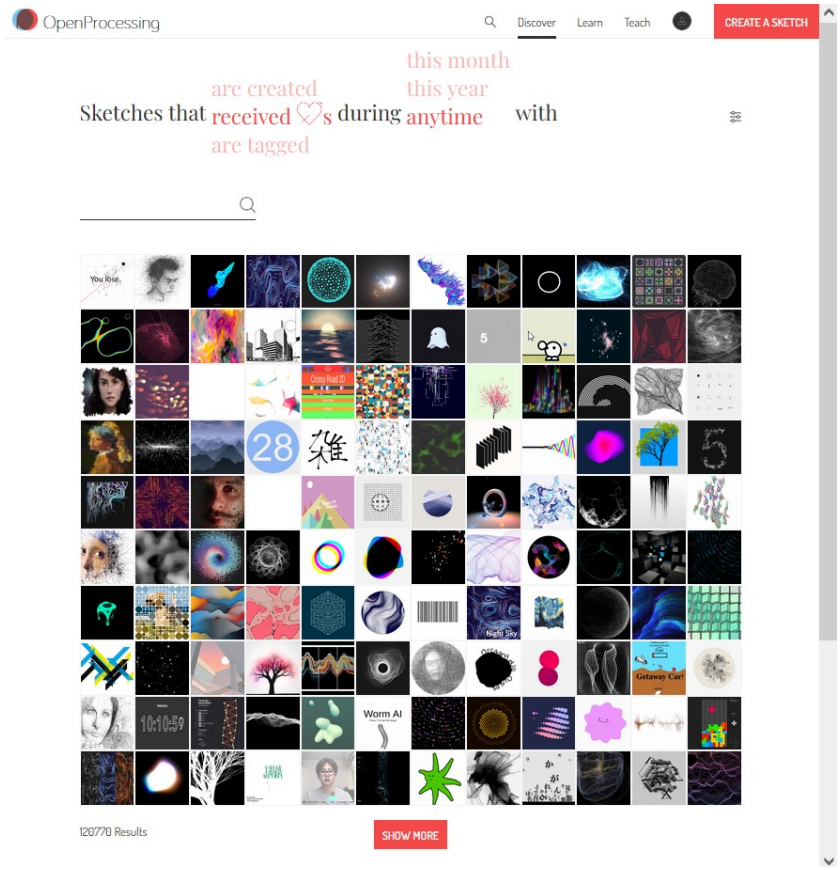

Figure 2.1: browse tab, showcasing grid of sketches

Since the squares representing the sketches are clickable and guide to a different part of the website, they have a reference link and therefore this information could be obtained by looking at the *html* attribute *href* of each sketch, which is represented in the *html* by the class 'sketchThumbContainer' as can be seen in figure 2.2.

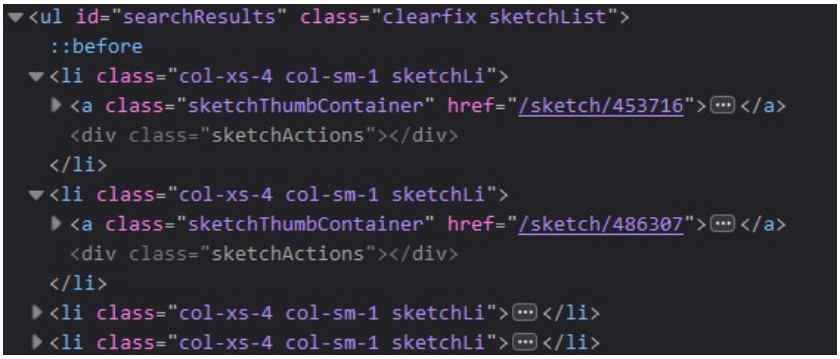

Figure 2.2: part of the *html* of the browse tab of *OpenProcessing*

Given all this information and structure, in order to obtain the sketches' links, the function on the script to collect links was programmed to open the url corresponding to the sketch grid and then click the "Show more" button to load more sketches into the grid until the amount of desired sketches is loaded. Once this point is reached, a list of all the *html* objects of the class 'sketchThumbContainer' is obtained, and then iterated to extract the information regarding the link itself. Finally, when all the links are extracted, they are exported into a csv file.

To have variation of the sample of projects to analyze, and also to have an extra factor to consider afterwards, the links collected were taken from 2 different sets within the same website. The first one, corresponding to the sketches that have been created, and the second one corresponding to the sketches that have received hearts. The goal was to get around 15 thousand sketches for each set, totaling 30 thousand. This meant running the script to collect the links twice.

The created sketches were accessed by the script through the following url `https://openprocessing.org/browse/?time=anytime&type=all&q=#`, that sets the filter terms to "are created" and "anytime" resulting in "Sketches that are created during anytime". In this version of the sketches grid, *OpenProcessing* orders the sketches in order of creation, from newest to oldest. For this reason, the sketches obtained from this set consisted of the 15 thousand most recently created at the moment of execution. In reality, 15216 links were collected for this set.

On the other hand, the sketches that have received hearts were accessed by the script through the following url `https://openprocessing.org/browse/?time=anytime&type=hearts`, that sets the filter terms to "receive hearts" and "anytime" resulting in "Sketches that receive hearts during anytime". In this case, the sketches are ordered from the one that has received the most hearts, or in other words, likes,

to the one that has received the least, with the minimum of 1 heart. Hence, the sketches expected to be obtained from this set correspond to the 15 thousand most liked sketches at the moment of execution. In practice, in this case, 14602 links were collected.

2.2.2 Second phase: projects downloading

The second part or phase of the scraping consisted of acquiring the creative coding projects themselves, this is accessing the links previously collected and from there, downloading the source code of the project.

Each link collected corresponds to one sketch page, where the name of the sketch, the author, the amounts of likes, comments and forks it has received and the sketch itself are displayed, as it can be seen in figure 2.3. Additionally, an extra tab can be opened from where, amongst other options, the possibility to download the link is shown. This, essentially, is the goal of this phase. For each sketch, the amount of likes and comments were collected and saved, and the sketch's source code was downloaded.

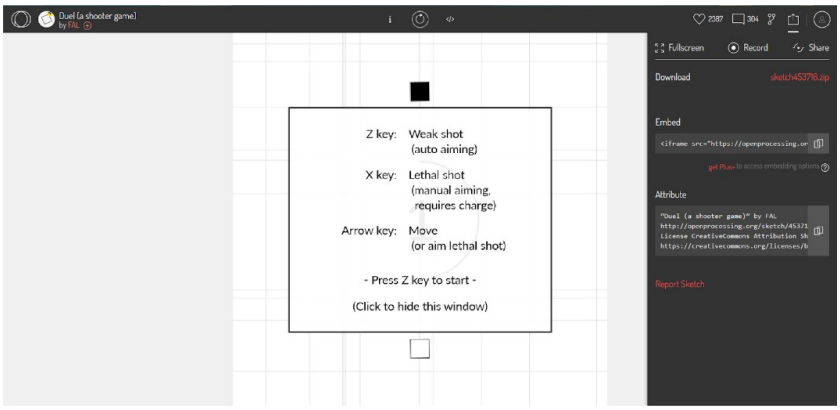

Figure 2.3: example of page of a sketch

To achieve this for all the links previously collected, they were split in groups of 2500 links each. These groups were run in parallel, firstly all the ones related to the set of links for created sketches, and then the groups of links from the hearted sketches. Each run taking approximately 5 hours.

Some of the projects failed to be recovered from the links, resulting in 14559 downloaded sketches from the created set, which means a loss of 657 sketches, equivalent to the 4.3% of links, and 14199 sketches downloaded from the hearted

set, which means a loss of 403 sketches, equivalent to the 2.8% of links. Given the small percentage they represent, and in the grand scheme of things there's still over 28 thousand sketches, these were neglectable and the script was not run again to try and recover those sketches.

The failure to recover some sketches can be explained by a few reasons: the sketch getting eliminated between link collection and attempt to download, hence giving a not found page; the source code of the sketch being hidden from the public and therefore not downloadable (see 2.4); or the sketch page taking to long to load, making the scraping script skip it altogether.

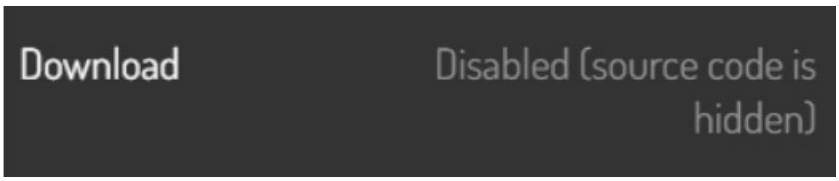

Figure 2.4: example of a sketch with hidden source code

2.3 Results

The result obtained after both phases of scraping was 28758 zip files, still separated into created and hearted sketches, each containing one full creative coding project, with the entire source code and everything else needed for the sketch to function. The sketches are identifiable by their id which is a number (in the form sketch<number>) generated by the site at the moment of creation of the project.

In order to analyze the source code, and proceed to the main part of this thesis, each project needed to be unzipped so that the tools used for the analysis could access the files and code of the sketch.

In the process of unzipping, from the 14559 total sketches of the created set, 14542 were successfully unzipped and, from the 14199 sketches corresponding to the hearted set, 14196 were successfully unzipped. That constitutes a total of 20 unsuccessfully unzipped projects, which is less than 1% of projects and a clearly neglectable amount.

To sum up, from the scraping executed, the final result is a total of 28738 creative coding projects, fully contained and including all files that compose it, ready to be analyzed by different tools. The analysis and its results will be explained in the following section.

Chapter 3

Analysis

After having the source code of a considerable amount of creative coding projects after completing the scraping step, comes the next step and the main part of this thesis, the analysis of the sketches.

This section will show a deep dive into the analysis done over these projects. A detailed explanation on what the goal of this research is, a description of what tools were implemented and how they were used to carry it out, an explanation on what factors were analyzed and why, and finally the results of the analysis will be presented.

3.1 Goal

The main goal is to get a feeling and general understanding of how creators are making and developing their creative coding projects. To achieve this, the aim is to look at two general main things, what creators use, and how creators develop.

Therefore, the objective is to firstly understand what tools and languages creators are coding with, and understand why these are popular or most used. Secondly, by looking into and with a deeper analysis of the source code, find general patterns that appear in the great scheme of things when looking at the creative coding projects in an aggregated way. In both cases, the aim is to also find what differences exist between the most liked sketches, denominated the hearted subgroup, and the newer ones, the created subgroup.

3.2 Tools and procedure

3.2.1 CLoC: Count Lines of Code

As mentioned before, the first thing that was to be looked at and analyzed was the wider level of information available. This is, what creators are using to develop their creative coding projects. The objective was to have a broad, aggregated description of the files used in these projects, that would give information on what languages are being used and which types of files are being used to assist on development and creation.

After some research, the tool cloc - Count Lines of Code [10] was found to categorize files in the way that was described above. Additionally, this tool also gives information about lines of code, that will also be talked about further in this document.

From the tool's github page [11], we can describe cloc as a command line program that takes files, directory and/or archive names as inputs, on which it then counts blank lines, comment lines and physical lines of course code. It recognizes and works on over three hundred languages, that are listed in detail with the corresponding file extensions in the section languages from the tool's page, which makes it a very versatile and reliable tool for the objective wanted at this stage of the analysis. Here, language is used to refer to both programming languages or types of file, such as text or css. Cloc also has some advanced features and uses that, as they were not used in this investigation, will not be discussed.

The tool in question also has many options that can be applied when running the command, like for example options regarding the output file or what things should be excluded or included when running the analysis. One option in particular presented itself to be very relevant in this case, `--skip-uniqueness`, which is used to, as the name implies, skip the uniqueness check on files and force all copies of files to be included in the report. After a quick look into a sample of the sketches, it was easy to notice that in a lot of them, the html file `index.html` was solely used to embed the contents of the javascript file `mySketch.js` hence there being many equal copies of this file. The comparison of results with and without this option will be presented later in this document (see section 3.4.1). But for the analysis done over the information, the results applying this option were preferred, since they give a truer overview of how many files exist.

```
<html>
    <head>
    <script src="mySketch.js" type="text/javascript"></script><script src="https://cdn.jsdelivr.net/npm/p5@0.5.2/lib/p5.min.js" type="text/javascript"></script>
    </head>
<body>
</body>

</html>
```

Figure 3.1: example of an index.html file, showing its simplicity and single purpose

Besides the many options and uses that this tool offers, it has some limitations. The ones relevant to us are mainly, taken from the tool's github page, in the section limitations [11]:

- Lines containing both source code and comments are counted as lines of code.
- Embedded languages are not recognized. For example, an HTML file containing JavaScript will be counted entirely as HTML.
- cloc treats compiler pragma's, for example `#if / #endif`, as code even if these are used to block lines of source from being compiled; the blocked lines still contribute to the code count.
- cloc's comment match code uses regular expressions which cannot properly account for nested comments using the same comment markers (such as `/* /* */ */`)

And some others that are not relevant to this research, are related to options that were not used or refer to very particular cases, and therefore won't be mentioned.

These limitations might make the results, mainly of line counts, a bit inaccurate, but on a neglectable level, given that they mostly describe very specific cases.

cloc was used to generate 4 files, in this case csv files that were further analyzed. It was run four times, twice for each group of sketches, hearted and created, and with and without the option mentioned above, `--skip-uniqueness`. The files generated have the following format: the columns describe amount of files of the language, language, amount of blank lines aggregated from all files of the language, amount of comment lines aggregated from all files of the language, amount of code lines aggregated from all files of the language. Each row corresponds to one language. As mentioned before, language is used to describe and refer to both programming languages or any type of file in general, such as text, css, svg, etc.

From this information, the factors more deeply analyzed, that will be further discussed and presented in a following section (see 3.4.1), include: types of files and languages used, most popular languages and file types, amounts of files and percentages, lines of code and lines of comments.

3.2.2 CR - Complexity Report

After having some data from the first level inspection of the files done with cloc (see 3.4.1), it was evident that the three most common languages that appeared were JavaScript, HTML and Arduino Sketch (Processing), together corresponding to about 97% of all files. That other 3%, corresponding mainly to media and stylesheet type files, was immediately deemed as not worth for analysis, given the types of files and neglectable quantity.

In the case of HTML files, as mentioned before, many of the files correspond to a copy of the same file shared among different sketches, and when not an exact copy, most of the apparitions of HTML files, they are used to embed code from JavaScript files. For this reason, and having a total of only 2198 unique HTML files from the over 19000 HTML files in total, it was also considered that these files were not worth analyzing further.

Being left with only the Arduino Sketch and JavaScript files, it was easy to see that a great number of the files correspond to JavaScript files. To be more precise, in the subgroup of hearted sketches there's almost twice as many JavaScript files as there's Arduino Sketch ones and almost three times as many in the subgroup of created sketches. In the case of all sketches collected, there's more than twice as many JavaScript files as Processing files.

The fact that JavaScript files outnumbered Processing files this much, and given that the newest sketches (subgroup created) have a higher difference in this sense, were a pretty good indicator that creators are leaning towards this as their preferred language for creative coding. Additionally, after some extensive research on available tools for source code analysis, that lead to almost exclusively finding tools designed for JavaScript, and none for Processing files, it was decided that a further analysis of metrics and other characterizations of creative coding projects was going to be done exclusively on JavaScript files.

The tool selected to carry out this analysis was cr - complexity-report [12] which gives a very exhaustive report of various metrics commonly used to determine complexity of software, such as cyclomatic complexity, or Halstead complexity measures. Additionally, information about file composition, mostly related to functions, is implicitly given.

The way cr works, from it's github page [12]:

> "complexity-report is just a node.js-based command-line wrapper around escomplex, which is the library that performs the actual analysis work.

> Code is passed to escomplex in the form of syntax trees that have been generated with esprima, the popular JavaScript parser."

In general terms, after receiving either the path of a single module or a folder containing many modules, "the tool will recursively read files from any directories that it encounters automatically"[12] and produce, from the `escomplex` function, the corresponding report in the desired output format. In this case, the output format chosen was json, the main reason being its easy for further analysis with a script.

This tool was chosen because, besides the extensiveness of the results it delivers, it was very easy to install, given that it works like a node module, simple to use, and manageable to read and further process the results, given the tree-like organization it produces.

Amongst the many command-line options that cr offers, for the case of this research, the option `--ignoreerrors` was used. This option ignores the parser errors and includes the files that raise these errors in the report.

Given the large number of sketches from each subgroup, hearted and created, they had to be subdivided in 2 smaller halves for cr to run in a reasonable time and for it to produce a json file of manageable size, totaling 4 individual runs. To get an idea of the size of these files, the smallest of these 4 files generated has more than 2.5 million lines, and the largest one has more than 10 million lines, so this split was needed.

The files generated have a dictionary-like format, organized as follows:

```
{
    reports: [
        {
            aggregate: {
                <report.aggregated.metrics>
            }
            functions: [
                {
                    name
                    <function.metrics>
                },
                ...
            ]
            <report.metrics>
            path
        },
        ...
    ],
```

```
19      <general.metrics>
20  }
```

where reports is a list of dictionaries where each represents a module or, in this case, a JavaScript file. Each report contains a dictionary `aggregate` with `<report.aggregated.metrics>`, a list of functions, each containing their name and `<function.metrics>`, and an extra set of `<report.metrics>`. There's also `<general.metrics>` associated with all of the modules analyzed.

This format is further described and explained in detail in the result format section of the tool CR is built on [13] and it was a bit modified to exclude the adjacency and visibility matrixes, that were of no use given the disconnection between modules belonging to different sketches.

The various metrics this tool reports on, and that are further analyzed in this research are, from the file 'METRICS.md', found on escomplex's github page [13]:

- Lines of code: Both physical (the number of lines in a module or function) and logical (a count of the imperative statements). A crude measure.
- Number of parameters: Analysed statically from the function signature, so no accounting is made for functions that rely on the arguments object. Lower is better.
- Cyclomatic complexity: Defined by Thomas J. McCabe in 1976, this is a count of the number of cycles in the program flow control graph. Effectively the number of distinct paths through a block of code. Lower is better.
- Cyclomatic complexity density: Proposed as a modification to cyclomatic complexity by Geoffrey K. Gill and Chris F. Kemerer in 1991, this metric simply re-expresses it as a percentage of the logical lines of code. Lower is better.
- Halstead metrics: Defined by Maurice Halstead in 1977, these metrics are calculated from the numbers of operators and operands in each function. Lower is better.
- Maintainability index: Defined by Paul Oman & Jack Hagemeister in 1991, this is a logarithmic scale from negative infinity to 171, calculated from the logical lines of code, the cyclomatix complexity and the Halstead effort. Higher is better.

The reports and metrics generated with cr were further looked into, aggregated, and analyzed with the help of python and the use of jupyter notebook, an interactive web application that makes data analysis easier to visualize and organize.

The factors analyzed from the results given by complexity-report, that will be further discussed later include: patterns in file and function names, repetition of function names, lines of code of files and functions, number of parameters in files and functions, amount of files per sketch, cyclomatic complexity, Halstead metrics, among others.

3.3 Factors analyzed

3.3.1 Factors from Cloc

The first factor to be analyzed is the type of files present in the sketches. This is to understand what languages the creators are using and what other types of files they are including in the development of their sketches.

The amount for each type of file will be observed, in an quantitative and percentage manner. From this, the aim is to see what languages are more popular, if there's any differences in the different subsets of sketches, created and hearted, and why these differences exist.

For the most relevant languages, the lines of code and comments will be looked at. These are to get an idea of general patterns and judge the complexity and level of documentation, organization and manageability of the projects. [14]

3.3.2 Factors from CR

The factors analyzed with the help of CR are divided in three sub groups; general patterns, to get an idea of the general skeleton of sketches; parameters and lines of code, to observe the behavior of functions and files in relation to these metrics; and lastly metrics, where commonly known metrics will be described.

General patterns

By counting files per sketch, functions per file and functions per project, it is possible to get an approach as to how sketches are organized. With all these statistics, an idea of how modularized these sketches are can be obtained. Also,

by looking at differences of created and hearted sketches, more insights on what makes them different can be found.

Another factor that helps define the sketches skeleton is the function names and the occurrences of these among sketches. This information can give insight on which functions are common, and how common they are, and what functions are, in a way, the essential ones for sketches.

Lines of code and Parameter counts

One of the factors analyzed in this section is the amount of parameters, for both files and functions. This can give insight on how the functions used in sketches work and relate to each other.

The other factor that will be looked at is the lines of code. In this case, physical and logical lines of code, for both functions and files. The lines of code per function give information on how long and complex these are [14]. The combination of lines per function and per file give insights on how modularized functions are and how many functions are being used in files.

Lastly, a these factors mentioned above will be studied for the functions setup and draw, the two main functions in which the library `p5.js` and therefore JavaScript based projects are built upon. Also, a look into the count of these functions will be executed and analyzed to see how common they truly are.

Metrics

The first factor to be analyzed in this section is cyclomatic complexity. It is defined as a quantitative metric that counts the amount of linearly independent paths through a program's code. It is usually associated with complexity, readability, maintainability and portability of a program. In general, a lower value is better. [15, 16, 17, 18]

The second factor analyzed is cyclomatic complexity density. This is derived from cyclomatic complexity and it is defined simply as cyclomatic complexity/lines of code , resulting in a sort of rate or percentage way of viewing and interpreting cyclomatic complexity. For this reason, it is also associated with the same qualities as cyclomatic complexity, and once again, a lower value is generally seen as better.[19]

The next factor is the Halstead metrics. This once again is a metric created to measure code complexity, with emphasis on computational complexity. It is

conformed of different metrics that are defined each by a formula combining predefined parameters and other Halstead metrics [20, 21, 22].

The parameters used to calculate the metrics are defined as follows:

- n1 = Number of distinct operators
- n2 = Number of distinct operands
- N1 = Number of operator instances
- N2 = Number of operand instances

And the metrics are defined as:

Metric	**Meaning**	**Formula**
n	Vocabulary	n1 + n2
N	Size/Length	N1 + N2
V	Volume	N * log2 n
D	Difficulty	n1/2 * N2/n2
E	Effort	V * D
B	Bugs	V / 3000
T	Testing time	E / k

Table 3.1: halstead metrics, showing their symbol, name and formula used to calculate it

Lastly, the maintainability index. This index represents the relative ease of maintaining the code, where a higher number is better [17]. Originally, and in the definition that's going to be used for the purpose of the thesis, this metric ranges from minus infinity to 171. Newest definitions of this metric normalize it to be contained in a range from 0 to 100. The calculation of this index takes into account the Halstead volume, cyclomatic complexity and lines of code, and it's defined by the following formula [23]:

$$MaintainabilityIndex = 171 - 5.2 * ln(HalsteadVolume) - 0.23 * (CyclomaticComplexity) - 16.2 * ln(LinesofCode) \quad (3.1)$$

3.4 Results

In this section, the results of the analysis of the factors mentioned before are presented. They are divided in two main subsections, one per each tool used in

the initial generation of data. In the case of both subsections, the original data collected was further analyzed, grouped and worked on with the help of python, and then organized and presented with the help of `jupyter notebook`, both with the libraries pandas, specialized in data analysis and manipulation, and `matplotlib`, specialized in the creation of static, animated and interactive visualizations.

Before diving into the deeper analysis, it can be useful to have in mind the amount of sketches present in each subgroup and in total (see table 3.2). These are used for some calculations of averages per sketch in the following sections.

group	amount of sketches
created	14542
hearted	14196
total	28738

Table 3.2: Amount of sketches for each subgroup and total aggregated amount

3.4.1 Results from CLoC

The information presented here is related to languages, types of files, amount of files and finally line counts for code, comments and blank lines. The results are presented firstly on a general basis, for each subgroup, created and hearted sketches, and then in an aggregated manner, and then a further analysis is carried out for the languages that have the higher file count.

Types of files and languages

For each subgroup of the sketches, three tables are presented. The first two show the total counts of files and lines for each language present within the projects. The first one, shows the results generated by cloc, without using the option `--skip-uniqueness`, which makes it so that the tool performs a uniqueness check to ignore 'repeated' files thus counting only unique files. The second table shows the results generated using the option `--skip-uniqueness`, which makes it so that the tool skips the uniqueness check and includes all the existing files, hence giving a count of all files. Lastly, the third table shows a comparison of the count of files from the previous two tables, along with the difference and percentage difference for each language.

language	files	blank	comment	code
SUM	16219	188494	164785	1911725
JavaScript	10911	145423	146168	936677
Arduino Sketch	4213	38810	17951	275735
HTML	862	1991	200	57084
CSS	100	405	153	2715
Text	51	1308	0	593871
GLSL	32	426	169	1219
SVG	22	1	3	466
JSON	19	7	0	21152
CSV	4	0	0	21779
Java	1	81	123	941
Python	1	40	18	76
INI	1	1	0	4
Markdown	1	1	0	4
Properties	1	0	0	2

Table 3.3: Count of files and lines for languages present in created sketches. Results without the use of –skip-uniqueness option

language	files	blank	comment	code
SUM	28367	232606	196620	2353688
JavaScript	12860	174290	175887	1109659
HTML	10241	12166	453	126201
Arduino Sketch	4572	42917	19439	309369
CSS	402	601	209	5737
Text	112	1368	0	599420
GLSL	83	1123	486	2993
SVG	37	2	5	824
JSON	25	7	0	26973
CSV	21	0	0	171449
Markdown	10	10	0	40
Java	1	81	123	941
Python	1	40	18	76
INI	1	1	0	4
Properties	1	0	0	2

Table 3.4: Count of files and lines for languages present in created sketches. Results with the use of –skip-uniqueness option

language	files no skip check	files skip check	delta	percentage delta (%)
SUM	16219	28367	12148	42.82
JavaScript	10911	12860	1949	15.16
Arduino Sketch	4213	4572	359	7.85
HTML	862	10241	9379	91.58
CSS	100	402	302	75.12
Text	51	112	61	54.46
GLSL	32	83	51	61.45
SVG	22	37	15	40.54
JSON	19	25	6	24.00
CSV	4	21	17	80.95
Java	1	1	0	0.00
Python	1	1	0	0.00
INI	1	1	0	0.00
Markdown	1	10	9	90.00
Properties	1	1	0	0.00

Table 3.5: Comparison of count of files for languages present in created sketches from the results generated by not using and using the –skip-uniqueness option.

language	files	blank	comment	code
SUM	19035	359313	277420	2585817
JavaScript	11119	206546	210616	1333197
Arduino Sketch	5784	119444	64690	702280
HTML	1556	3219	586	16498
GLSL	118	1680	634	4854
Text	114	26681	0	271245
SVG	102	2	49	8030
CSS	85	235	78	2049
JSON	57	19	0	49356
Java	53	1486	767	7955
CSV	38	0	0	190324
Properties	6	0	0	11
XML	1	1	0	13
Markdown	1	0	0	4
PHP	1	0	0	1

Table 3.6: Count of files and lines for languages present in hearted sketches. Results without the use of –skip-uniqueness option

language	files	blank	comment	code
SUM	28213	410405	327733	3022307
JavaScript	12220	235742	254824	1530422
HTML	8873	10793	681	68436
Arduino Sketch	6295	133002	70553	785111
GLSL	197	2023	758	6006
CSS	178	266	80	3272
Text	154	27068	0	290111
SVG	123	2	70	8367
JSON	62	21	0	86908
Java	53	1486	767	7955
CSV	46	0	0	235673
Properties	8	0	0	15
XML	2	2	0	26
Markdown	1	0	0	4
PHP	1	0	0	1

Table 3.7: Count of files and lines for languages present in hearted sketches. Results with the use of –skip-uniqueness option

language	files no skip check	files skip check	delta	percentage delta (%)
SUM	19035	28213	9178	32.53
JavaScript	11119	12220	1101	9.01
Arduino Sketch	5784	6295	511	8.12
HTML	1556	8873	7317	82.46
GLSL	118	197	79	40.10
Text	114	154	40	25.97
SVG	102	123	21	17.07
CSS	85	178	93	52.25
JSON	57	62	5	8.06
Java	53	53	0	0.00
CSV	38	46	8	17.39
Properties	6	8	2	25.00
XML	1	2	1	50.00
Markdown	1	1	0	0.00
PHP	1	1	0	0.00

Table 3.8: Comparison of count of files for languages present in hearted sketches from the results generated by not using and using the –skip-uniqueness option.

language	files	blank	comment	code
SUM	34353	535755	434201	4413283
JavaScript	21461	342046	350000	2201278
Arduino Sketch	9937	156764	81560	966442
HTML	2198	4900	754	71704
Text	161	27985	0	864609
CSS	161	586	223	4361
GLSL	129	1835	704	5263
SVG	124	3	52	8496
JSON	75	26	0	70070
Java	54	1567	890	8896
CSV	41	0	0	212051
Properties	6	0	0	11
Markdown	2	1	0	8
Python	1	40	18	76
XML	1	1	0	13
INI	1	1	0	4
PHP	1	0	0	1

Table 3.9: Count of files and lines for languages present in all sketches. Results without the use of –skip-uniqueness option

language	files	blank	comment	code
SUM	56580	643011	524353	5375995
JavaScript	25080	410032	430711	2640081
HTML	19114	22959	1134	194637
Arduino Sketch	10867	175919	89992	1094480
CSS	580	867	289	9009
GLSL	280	3146	1244	8999
Text	266	28436	0	889531
SVG	160	4	75	9191
JSON	87	28	0	113881
CSV	67	0	0	407122
Java	54	1567	890	8896
Markdown	11	10	0	44
Properties	9	0	0	17
XML	2	2	0	26
Python	1	40	18	76
INI	1	1	0	4
PHP	1	0	0	1

Table 3.10: Count of files and lines for languages present in all sketches. Results with the use of –skip-uniqueness option

language	files no skip check	files skip check	delta	percentage delta (%)
SUM	34353	56580	22227	39.28
JavaScript	21461	25080	3619	14.43
Arduino Sketch	9937	10867	930	8.56
HTML	2198	19114	16916	88.50
Text	161	266	105	39.47
CSS	161	580	419	72.24
GLSL	129	280	151	53.93
SVG	124	160	36	22.50
JSON	75	87	12	13.79
Java	54	54	0	0.00
CSV	41	67	26	38.81
Properties	6	9	3	33.33
Markdown	2	11	9	81.82
Python	1	1	0	0.00
XML	1	2	1	50.00
INI	1	1	0	0.00
PHP	1	1	0	0.00

Table 3.11: Comparison of count of files for languages present in all sketches from the results generated by not using and using the –skip-uniqueness option

After a quick look at these tables, it is easy to see that the results generated with the uniqueness check include a significantly less amount of files than the ones generated without uniqueness check, meaning that a considerable quantity of files constitutes non unique files. The biggest difference in quantity can be seen in HTML files, meaning that most of these files are not unique (see 3.5, 3.8 and 3.11). From a total of 19114 HTML files, only 2198 are unique.

Given that we are interested in the total number of files, the results of only the second table for each subset (tables 3.4, 3.7 and 3.10) will be further looked into. From this, we are interested in seeing how many different types of files are used, which are the most popular, how many files not directly related to coding the sketch itself are used, and the amount of files per sketch on average.

By looking at table 3.10, it is possible to see that the three main languages among sketches are JavaScript (js), HTML and Arduino Sketch (in this case, Processing or pde), in that order. For this reason, some rates and statistics will be obtained for this group. And, given that the heavy part of the programming of the sketches relies on the JavaScript and Processing languages, this group will also be looked at.

From observing tables 3.4 and 3.7, it is possible to see that hearted sketches have more `GLSL` files than created sketches. `GLSL` is a high-level shading language, used to shade 3D graphics, so it could be associated with more complex shading

	created	hearted	all
avg files per sketch	1.95	1.987	1.969
total not js/pde/html files	694	825	1519
avg not js/pde/html files per sketch	0.048	0.058	0.053
percentage not js/pde/html files over total files	2.44%	2.92%	2.68%
total js/pde/html files	27673	27388	55061
avg js/pde/html files per sketch	1.903	1.929	1.916
percentage js/pde/html files over total files	97.55%	97.07%	97.32%
total js/pde files	17432	18515	35947
avg js/pde files per sketch	1.199	1.304	1.25
percentage js/pde files over total files	61.45%	65.63%	63.53%
percentage js files over js/pde files	73.8%	66%	69.77%
js over pde files relation	2.8	1.94	2.3

Table 3.12: average files per sketch in total and for groups of languages and rate between the JavaScript and Processing languages

and animations that could result in more appealing and popular sketches. On the other hand, more css files can be seen in created sketches than in hearted sketches. Css files also define visuals, but for HTML and in a less complex way. The higher existence of HTML files in created sketches can explain why there's also more css type files.

From table 3.12, we have that the group corresponding to all JavaScript, HTML and Arduino Sketch (Processing) together, correspond to about 97% of all files, whereas the other files, that correspond mostly to media, styling type files or data files, together correspond to less than 3% of the total files.

From this same table (3.12), it is possible to see that every sketch, has on average 1.25 files of type JavaScript or Processing, which makes sense given that these sketches are either based on Processing or JavaScript for its development.

The rate between JavaScript files and Processing files presents to be greatly favored towards JavaScript, with this language outnumbering verb|pde| files by almost three times in the case of created sketches, almost two times for hearted sketches, and more than two times in the case of all sketches (see 3.12), showcasing JavaScript as the prominent language.

Top Languages

After seeing the results from the previous section mainly from table 3.11, it is pretty clear that there's essentially just 3 languages that are common occurrences amongst sketches. These languages are, in order of most common to less common, JavaScript, HTML, Arduino Sketch. These types of files are what essentially can be used to create a creative coding project, and therefore, will be looked at in more detail in the following part.

JavaScript

Group	total files	JavaScript files	% JavaScript files	avg JavaScript files per sketch	total loc JavaScript	avg loc per file	total comments JavaScript	avg comments per file
created	28367	12860	45.33	0.884335	1109659	86.29	175887	13.68
hearted	28213	12220	43.31	0.860806	1530422	125.24	254824	20.85
total	56580	25080	44.33	0.872712	2640081	105.27	430711	17.17

Table 3.13: JavaScript language statistics per subgroup: file and line count and averages and percentage of files corresponding to JavaScript

From this table it is possible to see that created sketches have a higher relative percentage of JavaScript files. In all cases, almost one in every two files is a JavaScript file. This shows a clear popularity of this language and the higher percentage in newer sketches could indicate that this language is gaining more popularity.

HTML

Group	total files	HTML files	% HTML files	avg HTML files per sketch	total loc HTML	avg loc per file	total comments HTML	avg comments per file
created	28367	10241	36.10	0.704236	126201	12.32	453	0.04
hearted	28213	8873	31.45	0.625035	68436	7.71	681	0.08
total	56580	19114	33.78	0.665112	194637	10.18	1134	0.06

Table 3.14: HTML language statistics per subgroup: file and line count and averages and percentage of files corresponding to HTML

It can be seen that the percentage of HTML files is greater in the case of created files. This can be explained by the pairing that happens with JavaScript files, where, for each HTML file, there usually is at least one JavaScript file. The later's percentage is also higher in created files (see 3.13).

An interesting thing to look at here is the fact that these files don't really have many comments at all. This can be explained by the nature of html files in general, in which comments are rarely seen. Also, after some inspection, it was evident that most html files represent a basic embed and render of the sketch developed in the js file (see 3.1).

Arduino Sketch

Group	total files	Arduino Sketch files	% Arduino Sketch files	avg Arduino Sketch files per sketch	total loc Arduino Sketch	avg loc per file	total comments Arduino Sketch	avg comments per file
created	28367	4572	16.12	0.314400	309369	67.67	19439	4.25
hearted	28213	6295	22.31	0.443435	785111	124.72	70553	11.21
total	56580	10867	19.21	0.378140	1094480	100.72	89992	8.28

Table 3.15: Arduino Sketch language statistics per subgroup: file and line count and averages and percentage of files corresponding to Arduino Sketch

For the case of Arduino Sketch files, the percentage over all files is higher for hearted sketches. This subgroup of sketches has older sketches that were created before popularity of JavaScript and the library `p5.js`, so it is reasonable that theres more sketches developed using that language.

Looking at the data in general, it makes sense that the percentage of HTML files is close to the one from JavaScript files, because, as explained beforeJavaScript based sketches function with one HTML, but can have more js files than just the `mySketch.js` file. On the other hand, the percentage of Arduino Sketch files is more isolated, given that Processing based sketches mostly only contain the one pde file.

When considering that Arduino Sketch based sketches tend to have only one file corresponding to this language, and that JavaScript based sketches tend to contain only one HTML file, its reasonable to consider these as an approximate count of sketches belonging to either type. By looking at the average of those types of files per sketch, and interpreting it as a percentage of sketches of either type, its possible to see that about 38% of all sketches correspond to Arduino Sketch based projects, while about 66% of all sketches correspond to JavaScript based projects. This clearly is not an exact percentage, but its a good approximate of the general distribution, that about two thirds of sketches are JavaScript based.

With the exception of html files, which are as mentioned before almost solely used to embed js, files from hearted sketches are on average longer, having more lines of code, almost double than what created files have. More lines of code can be an indicator of more complex software that produces, in this case, more appealing results.

In general, hearted files have more lines of comments, which usually indicates a more documented and organized code, which is commonly associated with more readable and maintainable code, that can lead to better quality results.

3.4.2 Results from CR Report

The information presented here is a further analysis on the results given by the cr tool. As it was mentioned and explained before (see 3.2.2), this is an analysis only for the JavaScript files that exist within the sketches and only sketches that include JavaScript files are being considered. This is relevant for statistics like the amount of files per project, where the count of projects is a subsection of all existing sketches that were downloaded during the scraping (see section 2).

This section is divided into 3 subsections. The first section is a look into general patterns that appear within the sketches, including amount of files per project, functions per file and most common names of functions. The second section is related to lines of code and parameter count of both files and functions. The third and final section is related to various metrics, like cyclomatic complexity and Halstead metrics, both at file and function level.

It is important to note that, after a first inspection of the results generated, it came to light that some sketches included files that essentially represented a copy or a subsection of the files that form the library `p5.js`. As these files are not really created by the people developing the creative coding projects, it was decided to ignore these files for the further analysis that was then carried out.

General patterns

The statistics described in this section represent the general patterns that exist among the JavaScript based creative coding projects. Firstly, a look into counts of functions per file and per project and amount of files per project. Then an inspection on function names, and which of them are the most common.

Each of these analyses is split into the subgroups created sketches, hearted sketches, and then looked at in an aggregated way.

File and function counts

As mentioned before, this section presents a description and analysis of the amount of functions per file, functions per project and files per project. Firstly, separated by subgroups created and hearted sketches, and then in an aggregated way.

	files_per_project	funcs_per_file	funcs_per_project
mean	1.24	3.76	4.67
std	1.03	27.24	35.14
min	1.00	0.00	0.00
25%	1.00	2.00	2.00
50%	1.00	2.00	2.00
75%	1.00	3.00	4.00
max	14.00	1916.00	2591.00

Table 3.16: statistics for files per project and functions per file and per project for created sketches

	files_per_project	funcs_per_file	funcs_per_project
mean	1.37	6.75	9.28
std	1.13	49.99	74.65
min	1.00	0.00	0.00
25%	1.00	2.00	3.00
50%	1.00	3.00	4.00
75%	1.00	5.00	6.00
max	19.00	2196.00	4202.00

Table 3.17: statistics for files per project and functions per file and per project for hearted sketches

	files_per_project	funcs_per_file	funcs_per_project
mean	1.30	5.22	6.81
std	1.08	40.10	57.11
min	1.00	0.00	0.00
25%	1.00	2.00	2.00
50%	1.00	3.00	3.00
75%	1.00	4.00	5.00
max	19.00	2196.00	4202.00

Table 3.18: statistics for files per project and functions per file and per project for all sketches

By looking at these tables, it's observable that both created and hearted subgroups of sketches have very similar statistics. The exception being the amount of functions, but, by looking at percentiles, it seems to be a case of a few escaped

values that throw off the calculations of mean. In the following graphs, showing the concentration of the data for created, hearted and all sketches, this can be checked.

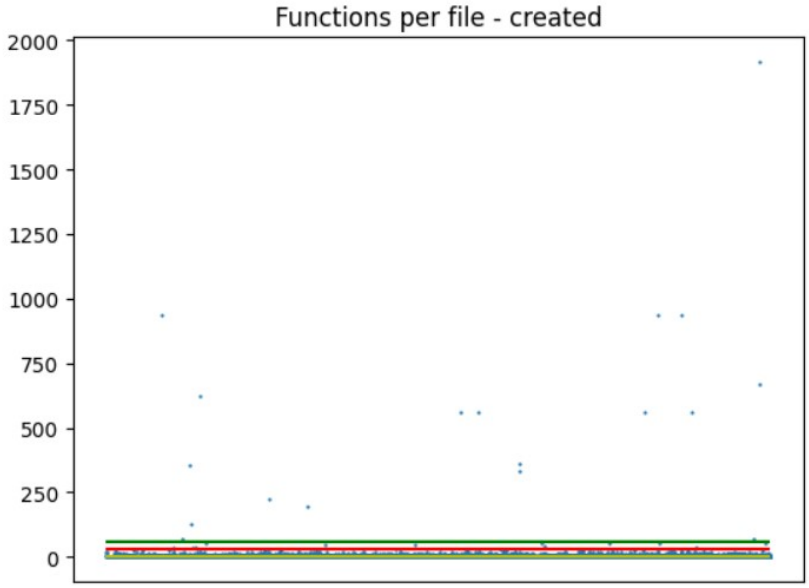

Figure 3.2: graph showing the amount of functions per file for created subset sketches

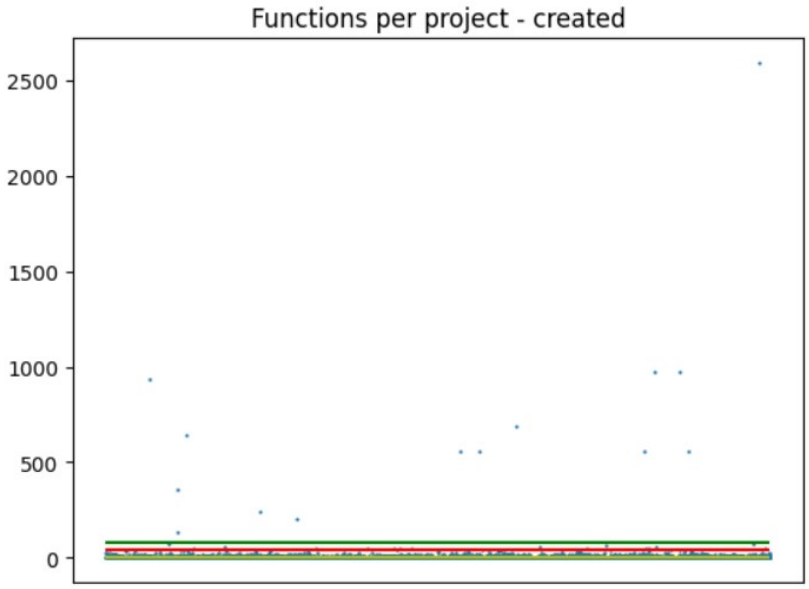

Figure 3.3: graph showing the amount of functions per project for created subset sketches

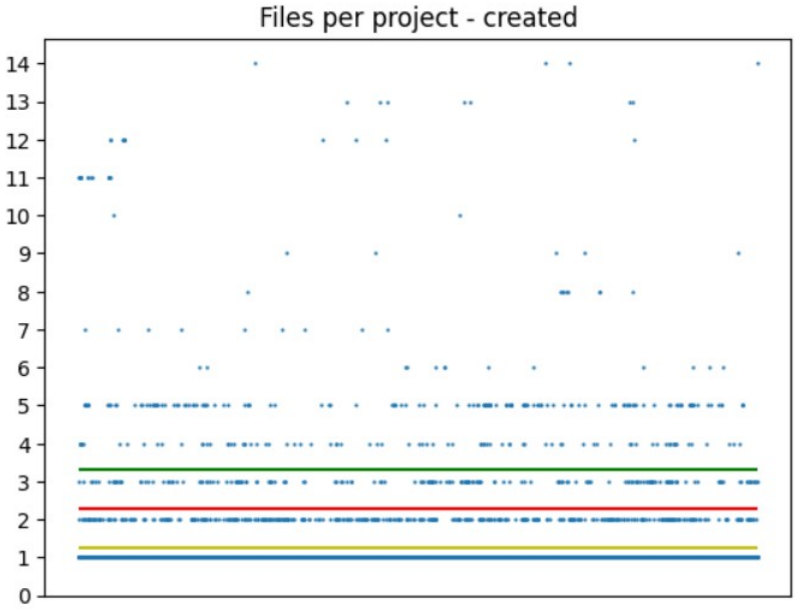

Figure 3.4: graph showing the amount of files per project for created subset sketches

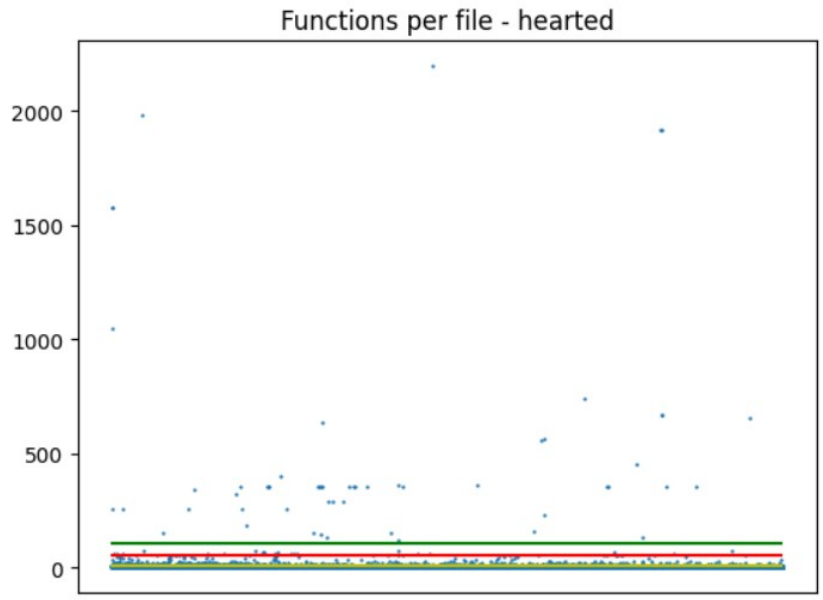

Figure 3.5: graph showing the amount of functions per file for hearted subset sketches

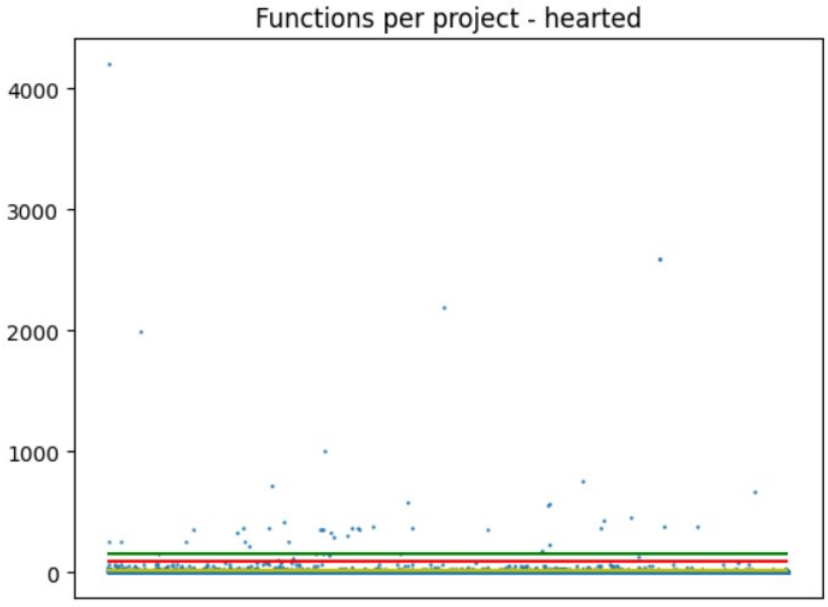

Figure 3.6: graph showing the amount of functions per project for hearted subset sketches

Figure 3.7: graph showing the amount of files per project for hearted subset sketches

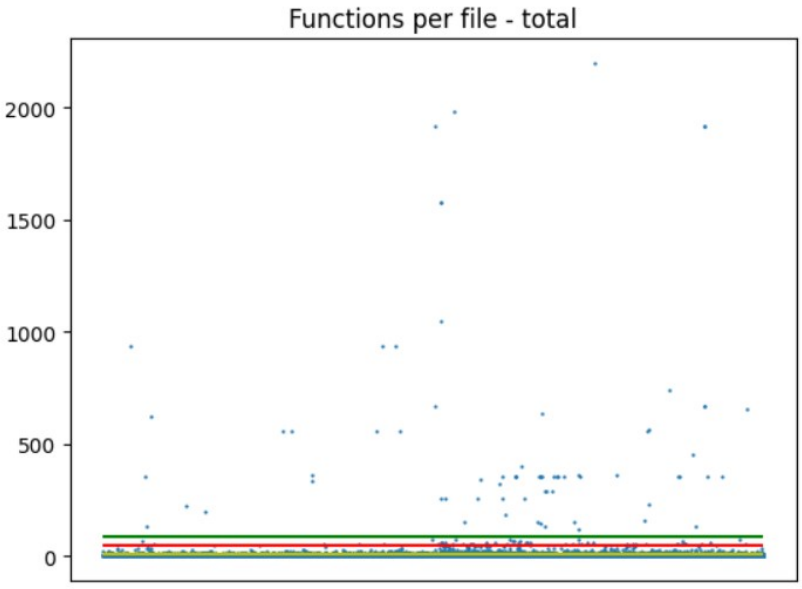

Figure 3.8: graph showing the amount of functions per file for all subset sketches

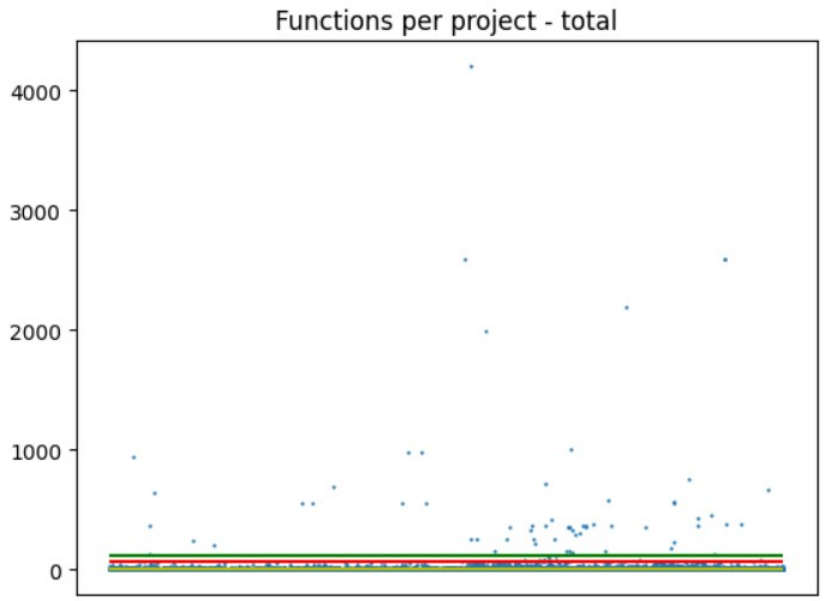

Figure 3.9: graph showing the amount of functions per project for all subset sketches

Figure 3.10: graph showing the amount of files per project for all subset sketches

In these graphs, the yellow line represents the mean of the data, the red lines

represent the mean plus and minus the standard deviation, and the green lines represent the mean plus and minus two times the standard deviation. If these exceed the range of the graph, the corresponding line is simply not shown.

By looking at the graphs that show files per project (figures 3.4, 3.7 and 3.10), it is possible to see that the majority of sketches contain 1 single JavaScript file. Some, containing 2 or 3, this being more common among Hearted sketches, which can indicate a better modularization being carried out in these sketches, which usually leads to more readable and organized code and therefore, in most cases, better output.

For the case of functions these graphs confirm the theory that there are some values that escape the normal for these projects. Given the 'escaped' values, the y axis, representing the count of functions, will be limited on the top, to have a better view of the data, by limit equal to percentile 99, which means 99% of the data will still be shown.

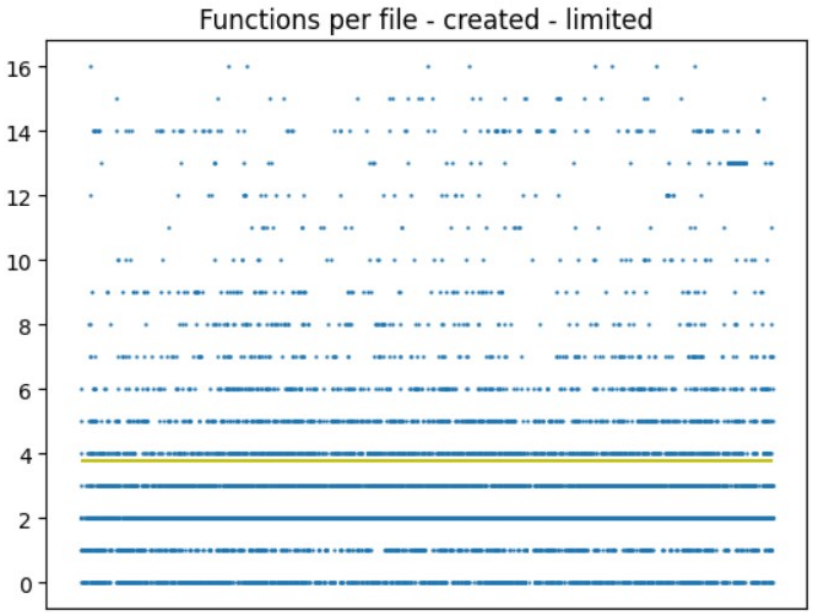

Figure 3.11: graph showing the amount of functions per file for created subset sketches, with y axis limited on top by percentile 99

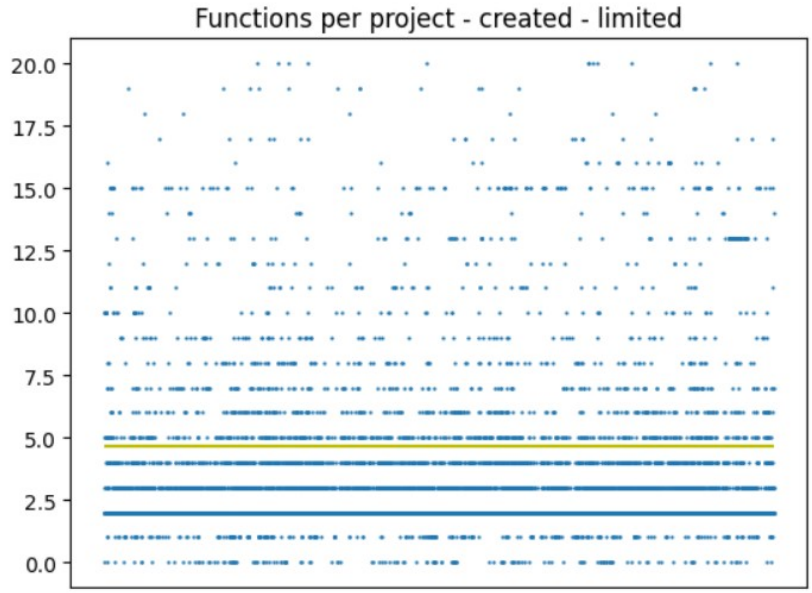

Figure 3.12: graph showing the amount of functions per project for created subset sketches, with y axis limited on top by percentile 99

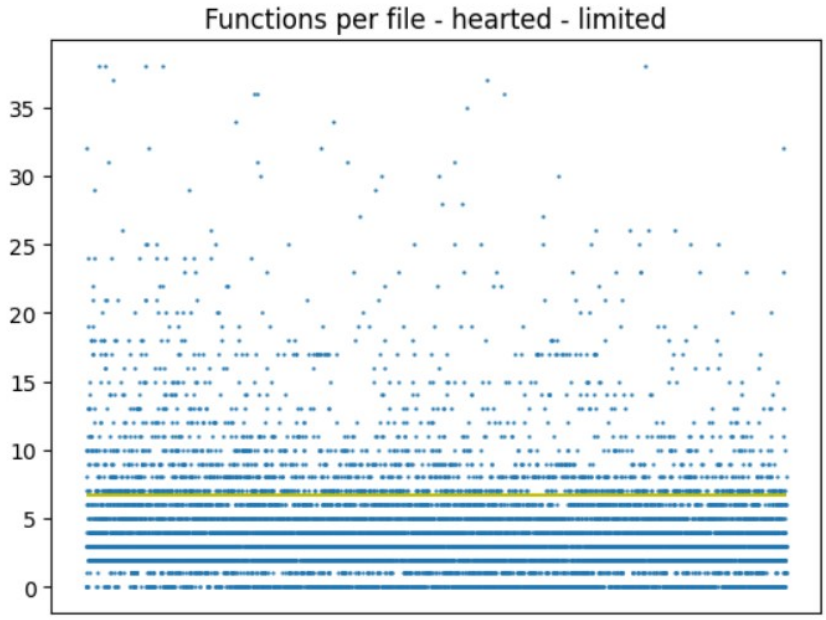

Figure 3.13: graph showing the amount of functions per file for hearted subset sketches, with y axis limited on top by percentile 99

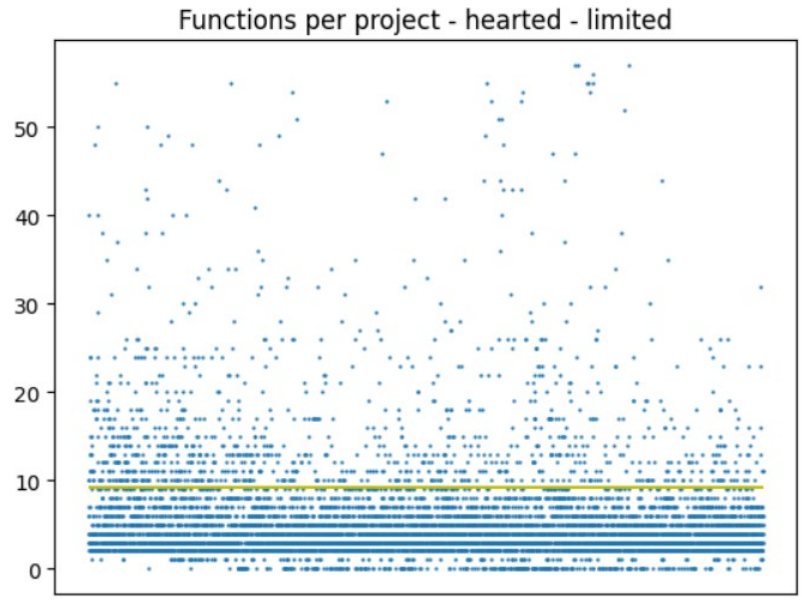

Figure 3.14: graph showing the amount of functions per project for hearted subset sketches, with y axis limited on top by percentile 99

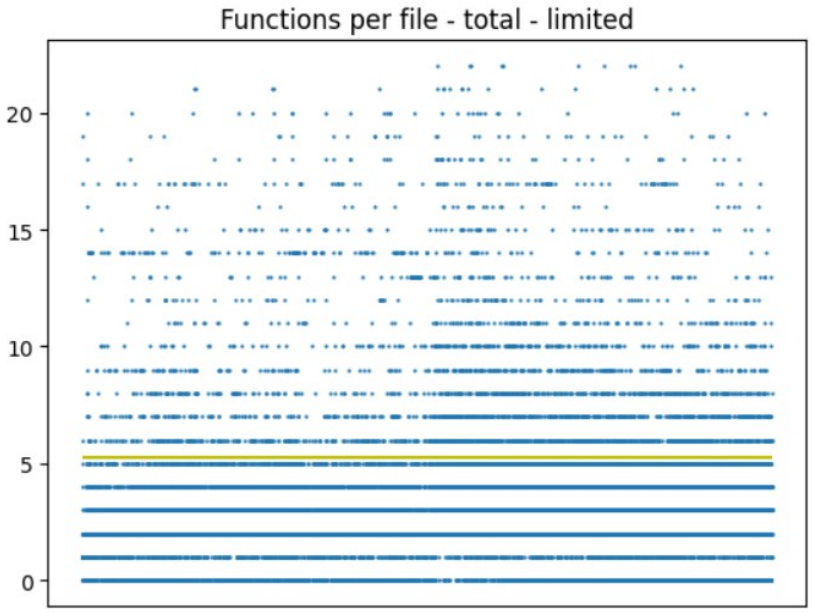

Figure 3.15: graph showing the amount of functions per file for all subset sketches, with y axis limited on top by percentile 99

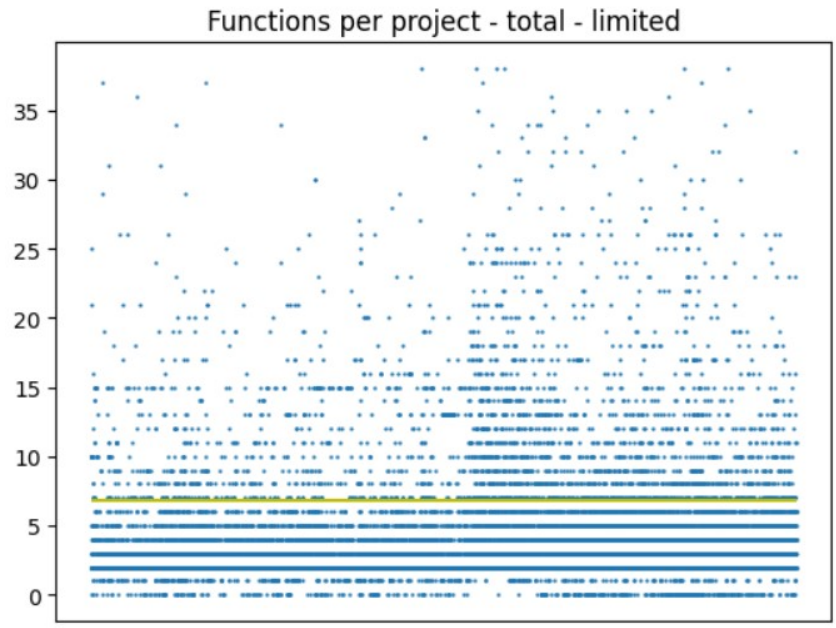

Figure 3.16: graph showing the amount of functions per project for all subset sketches, with y axis limited on top by percentile 99

When looking at the differences of concentration of data between functions per project in created (graph 3.12) and hearted (3.14) sketches, it can be seen that the sketches in general present a higher total count of functions, where for created sketches it mainly concentrated between 2 and 4 functions per project, whereas the hearted sketches move between 2 and 6 functions per project. This makes a lot of sense having in mind that hearted sketches present on average a higher file per project amount. This information can be in a way, translated to all sketches, where sketches present more repetitions of functions per project between 2 and 6.

Function names and occurrences

This section aims to have a general idea of how many functions are 'shared' between sketches. As realistically it is not really possible to fully have identical functions between different sketches to compare, the names of functions will be used to get some idea of possible sketch patterns.

The data obtained will be presented numerically, for created, hearted and all sketches, and the 20 most common function names will be shown for each set of sketches.

	created	hearted	total
total amount of function names	6972	16719	19818
function names with more than 1 occurrence	2714	6826	8278
function names with more than 10 occurrences	200	490	1064
function names with more than 50 occurrences	36	72	103
function names with more than 100 occurrences	16	38	55
function names with more than 500 occurrences	6	10	13
function names with more than 1000 occurrences	5	7	7
function names with more than 5000 occurrences	2	3	3

Table 3.19: total count of unique function names and amounts of functions names with multiple occurrences, for all sets of sketches

```
'setup': 9439,
'draw': 8941,
'<anonymous>': 4111,
'preload': 1342,
'mousePressed': 1037,
'keyPressed': 781,
'<anonymous>.show': 322,
'mouseClicked': 264,
'get': 214,
'Score': 190,
'keyTyped': 176,
'e': 150,
'mouseReleased': 148,
'set': 127,
'windowResized': 122,
'<anonymous>.update': 119,
'<anonymous>.move': 93,
'Star': 87,
'copyToClipboard': 87,
'setRandom': 86,
```

Figure 3.17: 20 most common names of functions among created sketches

```
'setup': 7978,
'draw': 7708,
'<anonymous>': 7503,
'preload': 1697,
'mousePressed': 1346,
'keyPressed': 1053,
'value': 1053,
'<anonymous>.update': 584,
'get': 578,
'<anonymous>.display': 520,
'<anonymous>.move': 379,
'keyTyped': 332,
'mouseClicked': 325,
'<anonymous>.show': 320,
'mouseReleased': 305,
'Particle': 302,
'run': 294,
'windowResized': 267,
'<anonymous>.draw': 246,
'e': 229,
```

Figure 3.18: 20 most common names of functions among hearted sketches

```
'setup': 17417,
'draw': 16649,
'<anonymous>': 11614,
'preload': 3039,
'mousePressed': 2383,
'keyPressed': 1834,
'value': 1065,
'get': 792,
'<anonymous>.update': 703,
'<anonymous>.show': 642,
'<anonymous>.display': 605,
'mouseClicked': 589,
'keyTyped': 508,
'<anonymous>.move': 472,
'mouseReleased': 453,
'windowResized': 389,
'e': 379,
'Particle': 360,
'set': 351,
'<anonymous>.draw': 306,
```

Figure 3.19: 20 most common names of functions among all sketches

By looking at all the information in the tables and lists, it is possible to see that in the case of hearted sketches, there is a larger amount of unique function names in general than in the created sketches. This could be explained by a higher rate of modularization of functions, and a better application of simplified functions that have one purpose, instead of functions that execute many things at once. This is, in a way, considered better and more manageable coding, which usually leads to

better results, which in this case, could turn into popular sketches.

It can also be seen that in the case of hearted sketches, there are more functions that are repeated and more repetitions of the functions in general. A possible explanation of this, is a better understanding of the tool being used and what different things can be achieved, which leads to using all the available resources that, in this case, the `p5.js` library offers.

Lines of code and Parameter counts

In this section, a deeper look into the physical and logical lines of code, and parameter count is presented, both in the file and function levels. The analysis is split into the subgroups created sketches and hearted sketches, and then looked at in an aggregated way. Lastly, a description of the "setup" and "draw" functions that constitute the base of how the `p5.js` library works will be presented.

File level

	sloc_physical_mod_list	**sloc_logical_mod_list**	**param_mod_list**
count	12009.00	12009.00	12009.00
mean	87.67	61.36	2.86
std	447.41	264.66	42.12
min	1.00	0.00	0.00
25%	21.00	11.00	0.00
50%	43.00	23.00	0.00
75%	87.00	50.00	0.00
max	29837.00	13492.00	3296.00

Table 3.20: statistics for total physical and logical lines of code and parameters per module/file for created sketches

	sloc_physical_mod_list	sloc_logical_mod_list	param_mod_list
count	11566.00	11566.00	11566.00
mean	141.76	92.45	7.09
std	655.62	625.40	72.53
min	1.00	0.00	0.00
25%	42.00	19.00	0.00
50%	76.00	42.00	0.00
75%	128.00	75.00	4.00
max	47549.00	34121.00	3296.00

Table 3.21: statistics for total physical and logical lines of code and parameters per module/file for hearted sketches

	sloc_physical_mod_list	sloc_logical_mod_list	param_mod_list
count	23575.00	23575.00	23575.00
mean	114.21	76.61	4.94
std	559.97	477.28	59.07
min	1.00	0.00	0.00
25%	29.00	13.00	0.00
50%	58.00	31.00	0.00
75%	110.00	64.00	2.00
max	47549.00	34121.00	3296.00

Table 3.22: statistics for total physical and logical lines of code and parameters per module/file for all sketches

From looking at the statistics presented in these tables (3.20, 3.21 and 3.22), it is possible to identify that there are some values escape a lot from the others, so, the following graphs are limited to show only the majority of the data that explains general patterns. In the case of the graphs related to lines of code, given the larger values, the y axis will be limited to be at most the value of percentile 95, this means, still 95% of the data is shown. In the case of parameters, the values are much smaller and easier to see on a graph, so the y axis will be limited to percentile 99, showing 99% of the data.

In the following graphs, the yellow line represents the mean of the data, value which is shown in the tables presented above.

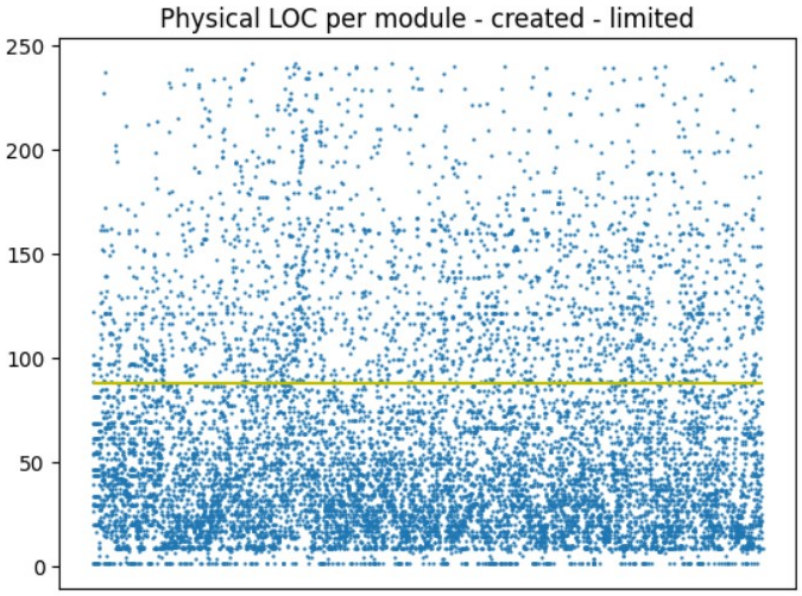

Figure 3.20: graph showing the physical lines of code count per file for created sketches, with y axis limited on top by percentile 95

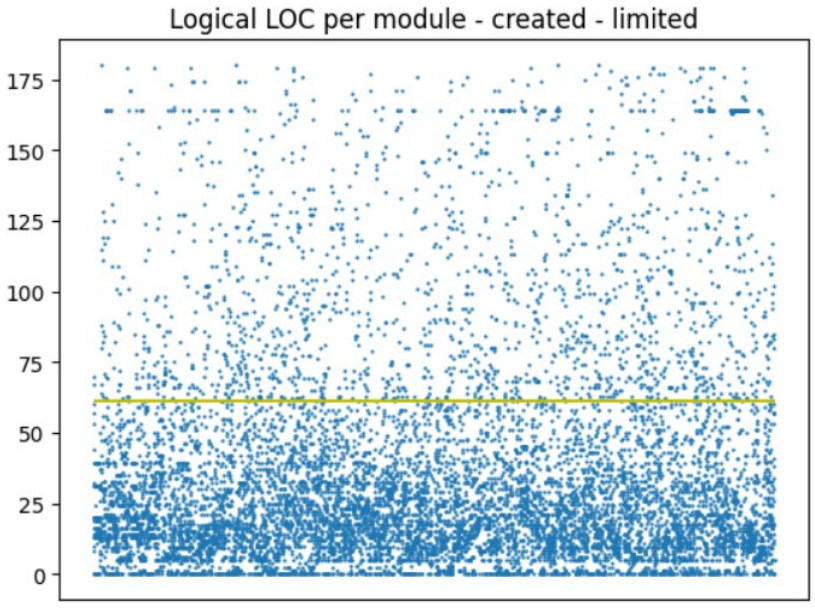

Figure 3.21: graph showing the logical lines of code count per file for created sketches, with y axis limited on top by percentile 95

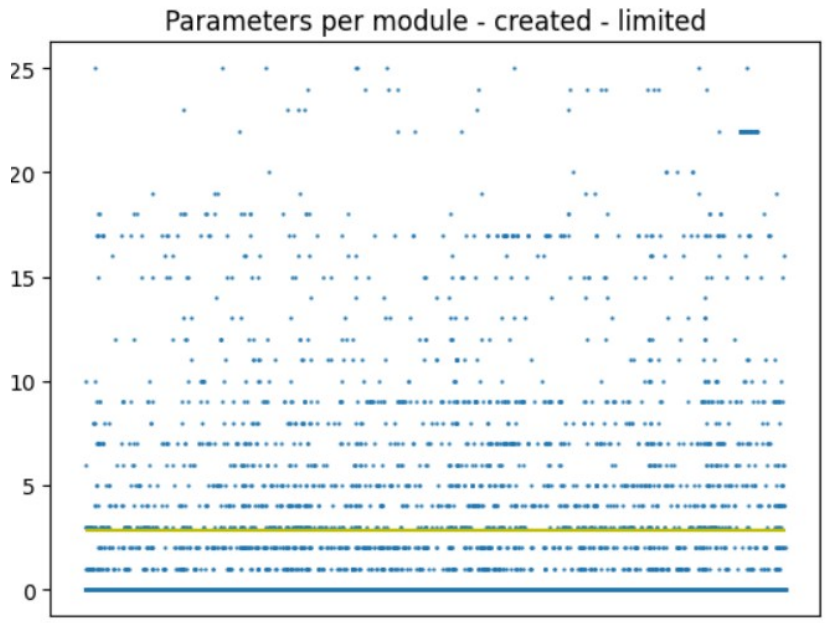

Figure 3.22: graph showing the parameters count per file for created sketches, with y axis limited on top by percentile 99

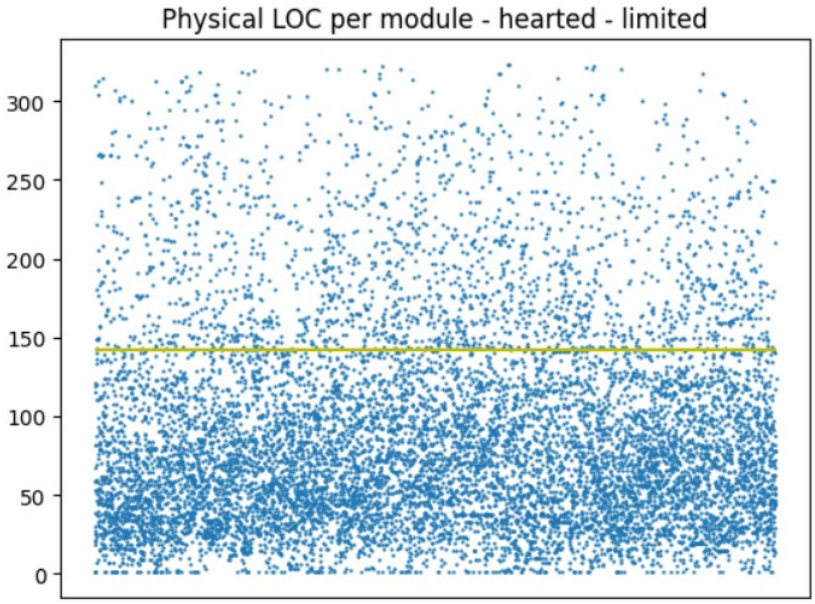

Figure 3.23: graph showing the physical lines of code count per file for hearted sketches, with y axis limited on top by percentile 95

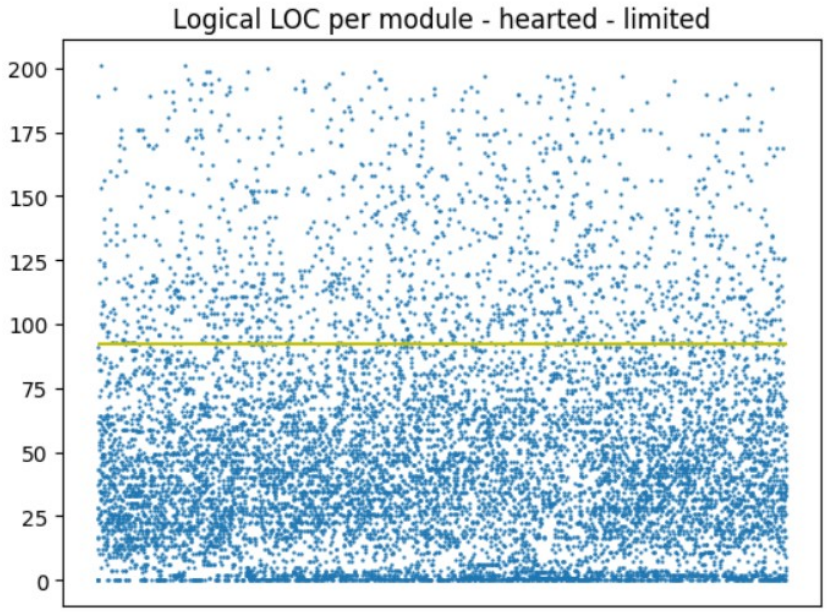

Figure 3.24: graph showing the logical lines of code count per file for hearted sketches, with y axis limited on top by percentile 95

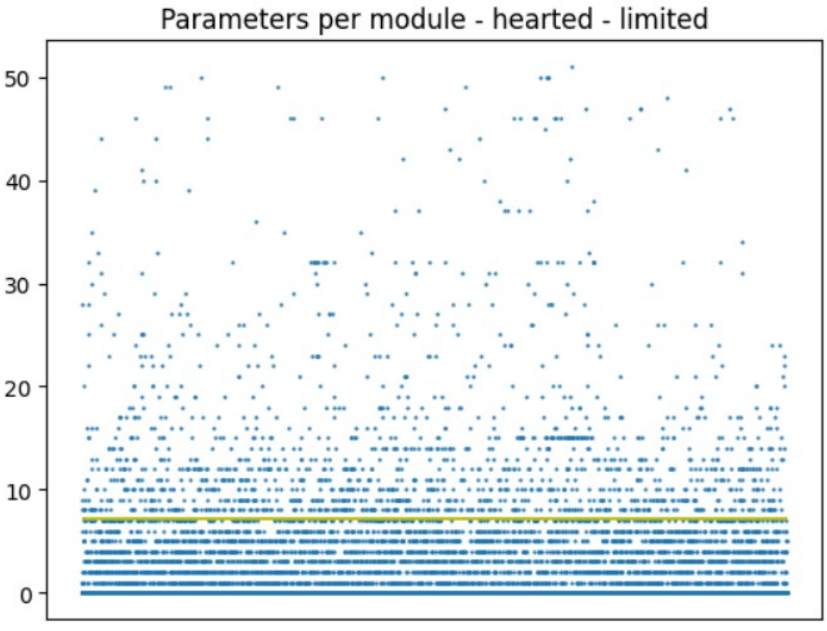

Figure 3.25: graph showing the parameters count per file for hearted sketches, with y axis limited on top by percentile 99

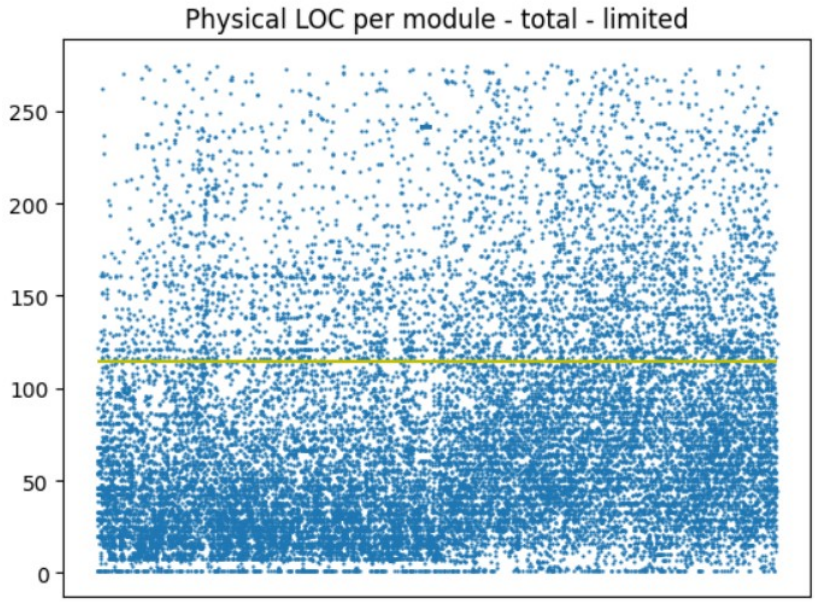

Figure 3.26: graph showing the physical lines of code count per file for all sketches, with y axis limited on top by percentile 95

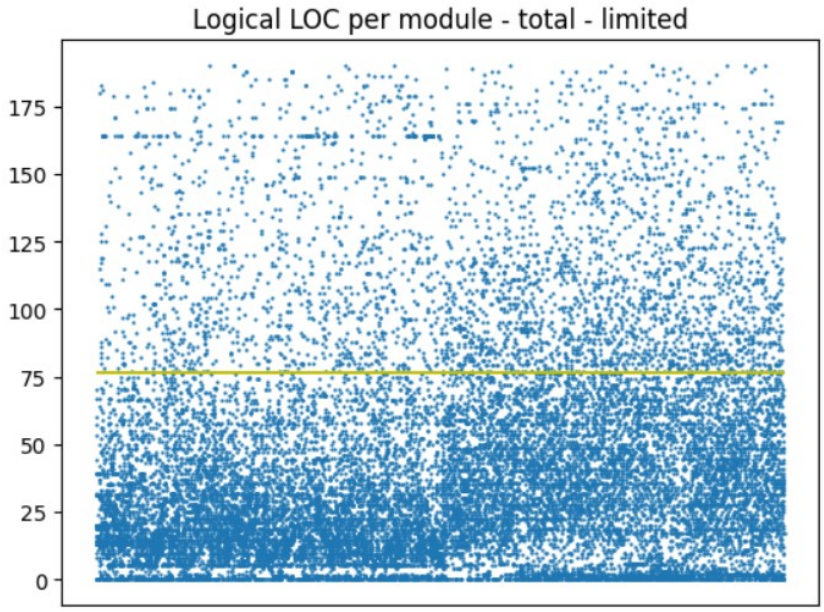

Figure 3.27: graph showing the logical lines of code count per file for all sketches, with y axis limited on top by percentile 95

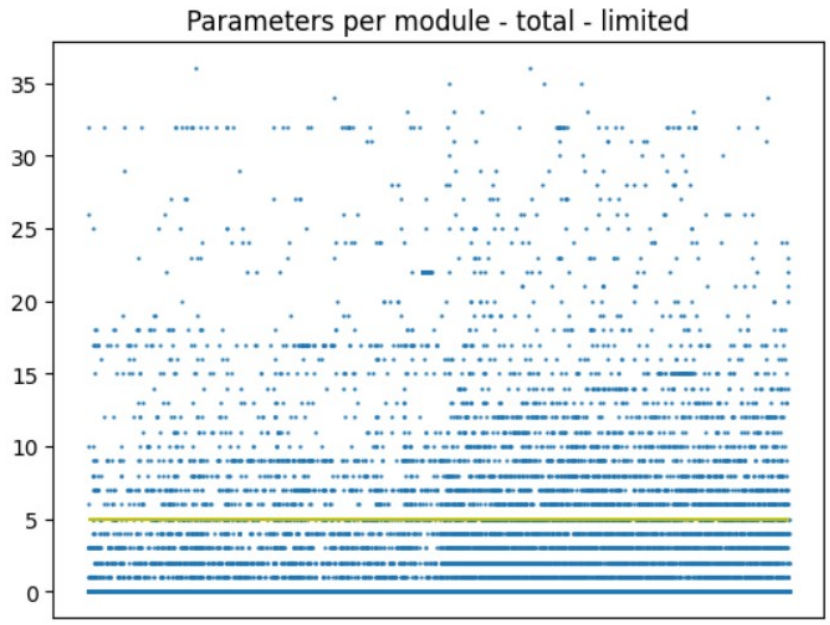

Figure 3.28: graph showing the parameters count per file for all sketches, with y axis limited on top by percentile 99

In the case of lines of code, the following observations can be made:

From graph 3.21, it can be seen that there is a cluster of sketches at around the 165 logical lines of code. This could indicate that a few of the newly created sketches either take inspiration and fork from an existing sketch with a file with those qualities, or that the people developing these sketches are opting to download files from the library they're working with and adding it directly to their project, instead of importing the module from an external source.

From comparing graphs 3.21 and 3.24, it can be observed that the files from hearted sketches present in general higher count of lines that the ones from created sketches. This could be explained by what was discussed in previous section 3.4.2, where hearted sketches presented a higher rate of functions per file, which typically translates into more lines of code.

In general, (see 3.26 and 3.27) it is possible to see that most sketches files have a logical line count lower than 50, and a physical line count lower than 100, which is in general, very readable and manageable code.

In the case of parameters per file, it is possible to see on graph 3.28, that

there is a very high concentration of files that have at most 5 parameters among their functions, being the most common having no parameters at all. With the knowledge from section 3.4.2, where it was discussed that a large percentage of files are formed by the functions setup and draw, functions that commonly don't receive parameters, this observation makes sense and leads to conclude, that most files are versions of `mySketch.js`, which commonly only has these two functions.

Function level

	sloc_physical_func_list	sloc_logical_func_list	param_func_list
count	45151.00	45151.00	45151.00
mean	19.36	13.18	0.76
std	109.13	84.65	1.48
min	1.00	0.00	0.00
25%	3.00	2.00	0.00
50%	6.00	5.00	0.00
75%	17.00	12.00	1.00
max	10722.00	13465.00	18.00

Table 3.23: statistics for total physical and logical lines of code and parameters per function for created sketches

	sloc_physical_func_list	sloc_logical_func_list	param_func_list
count	78045.00	78045.00	78045.00
mean	22.38	11.85	1.05
std	267.66	125.38	1.52
min	1.00	0.00	0.00
25%	2.00	2.00	0.00
50%	6.00	5.00	0.00
75%	17.00	12.00	2.00
max	47549.00	27827.00	18.00

Table 3.24: statistics for total physical and logical lines of code and parameters per function for hearted sketches

	sloc_physical_func_list	sloc_logical_func_list	param_func_list
count	123196.00	123196.00	123196.00
mean	21.28	12.33	0.94
std	223.05	112.18	1.52
min	1.00	0.00	0.00
25%	3.00	2.00	0.00
50%	6.00	5.00	0.00
75%	17.00	12.00	1.00
max	47549.00	27827.00	18.00

Table 3.25: statistics for total physical and logical lines of code and parameters per function for all sketches

Once again, it's possible to see from the data on tables that some values escape the norm a lot, so, some of the following graphs are limited to show only the majority of the data that explains general patterns. In the case of the graphs related to lines of code, given the larger values, the y axis is limited to be at most the value of percentile 95, this means, still 95% of the data is shown. In the case of parameters, since the maximum value does not surpass 18, the graph will be left unlimited, showing all the available data.

In each of the following graphs, the yellow line represents the mean of the data, value which is shown in the tables presented above. The red line represent the mean plus the standard deviation, and the green line represent the mean plus two times the standard deviation. If these exceed the range of the graph, the corresponding line is simply not shown.

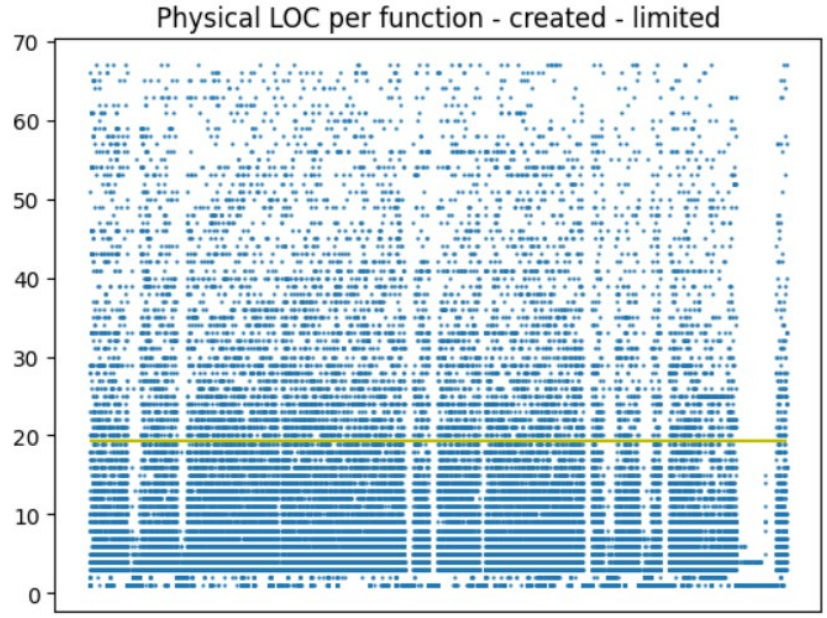

Figure 3.29: graph showing the physical lines of code count per function for created sketches, with y axis limited on top by percentile 95

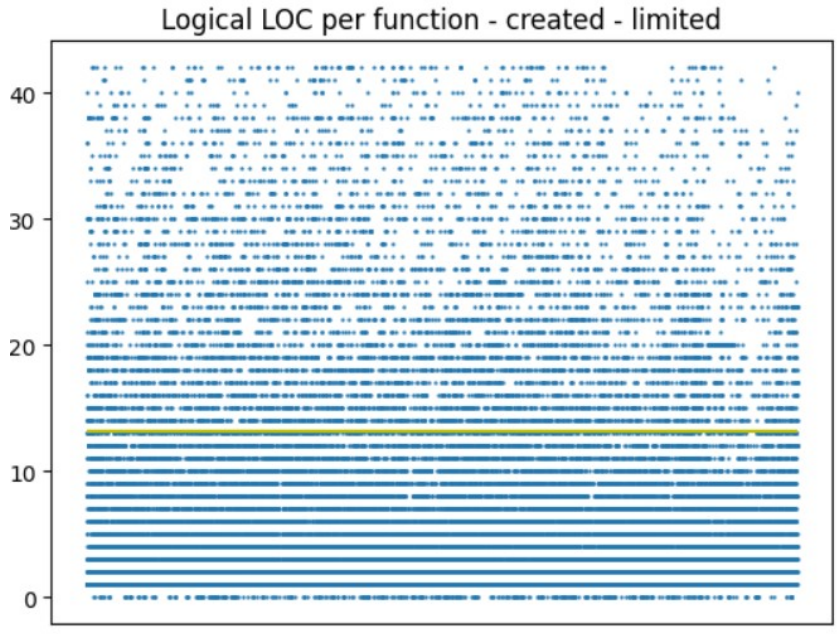

Figure 3.30: graph showing the logical lines of code count per function for created sketches, with y axis limited on top by percentile 95

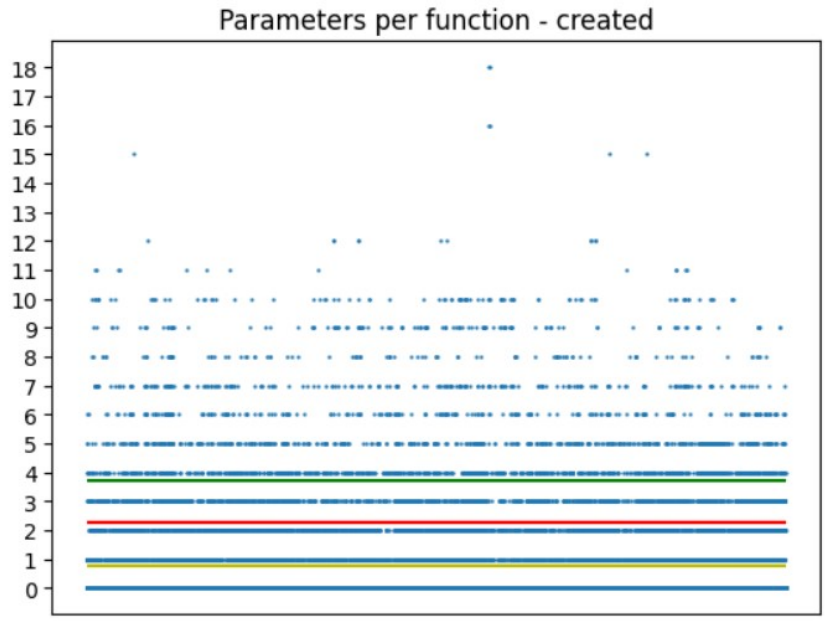

Figure 3.31: graph showing the parameters count per function for created sketches

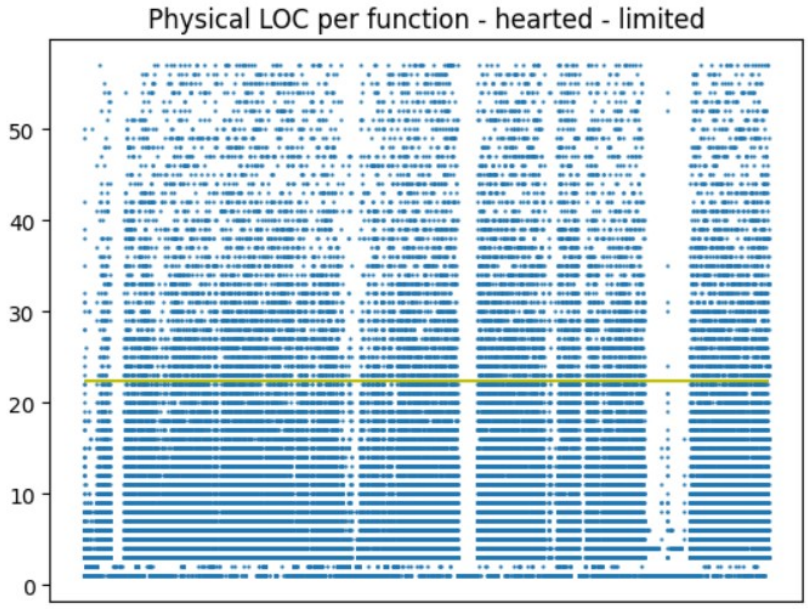

Figure 3.32: graph showing the physical lines of code count per function for hearted sketches, with y axis limited on top by percentile 95

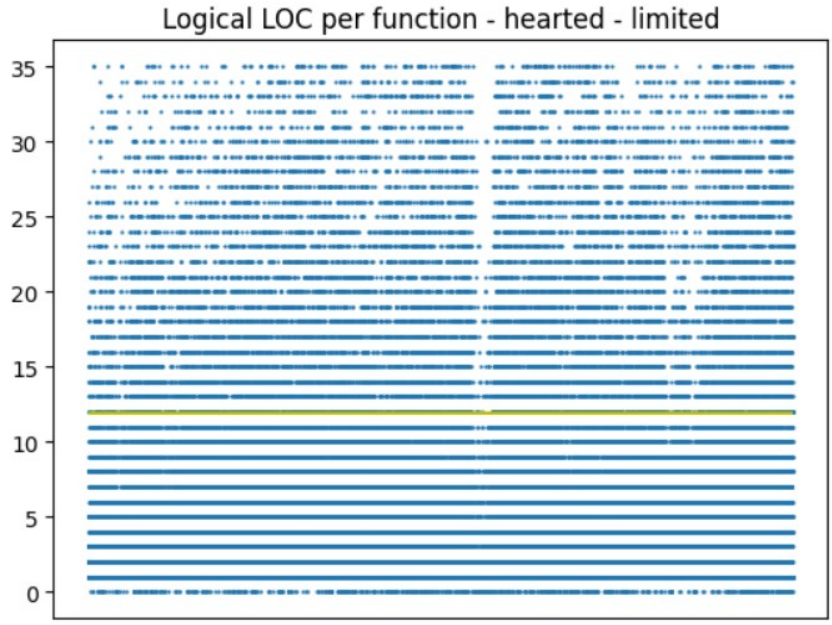

Figure 3.33: graph showing the logical lines of code count per function for hearted sketches, with y axis limited on top by percentile 95

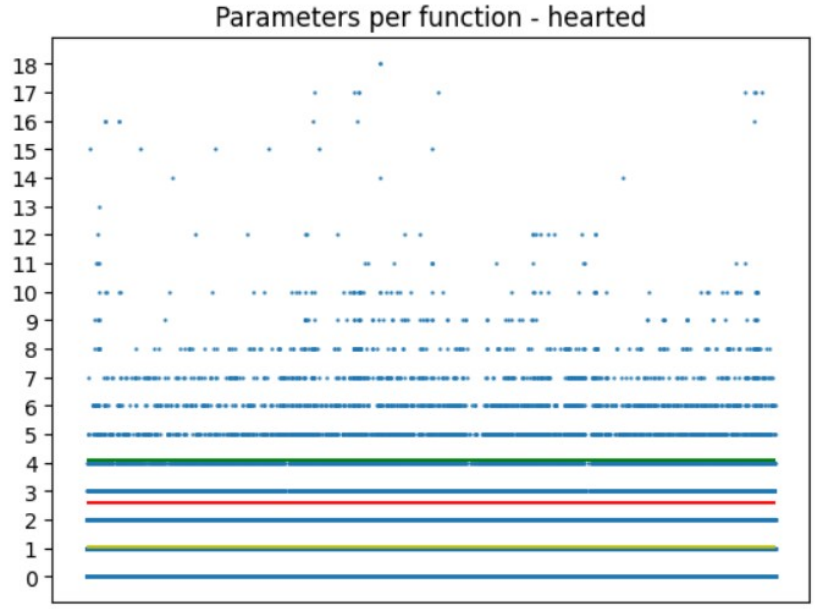

Figure 3.34: graph showing the parameters count per function for hearted sketches

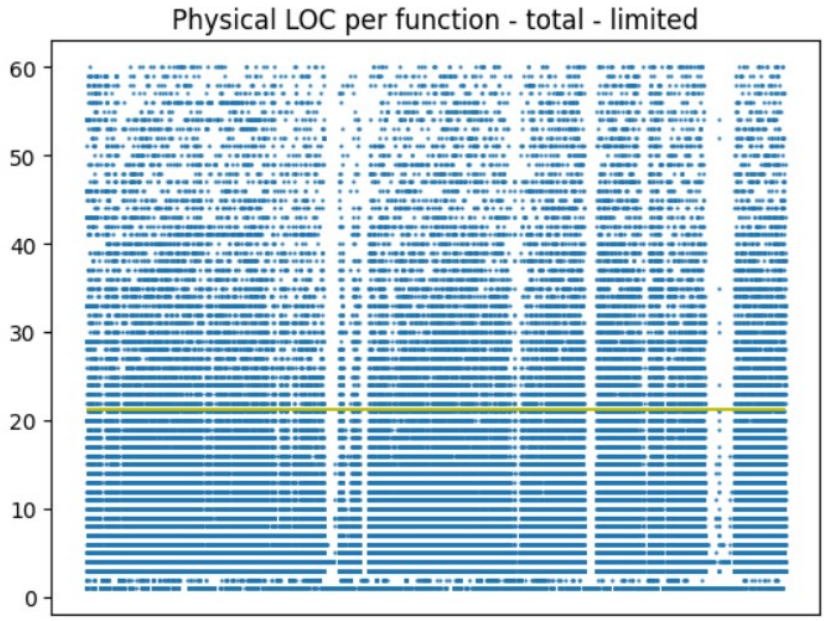

Figure 3.35: graph showing the physical lines of code count per function for all sketches, with y axis limited on top by percentile 95

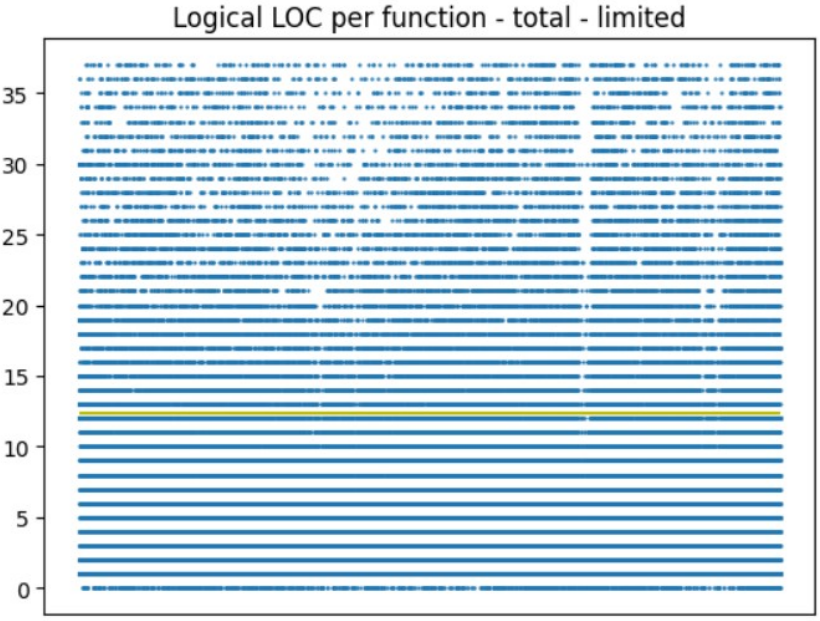

Figure 3.36: graph showing the logical lines of code count per function for all sketches, with y axis limited on top by percentile 95

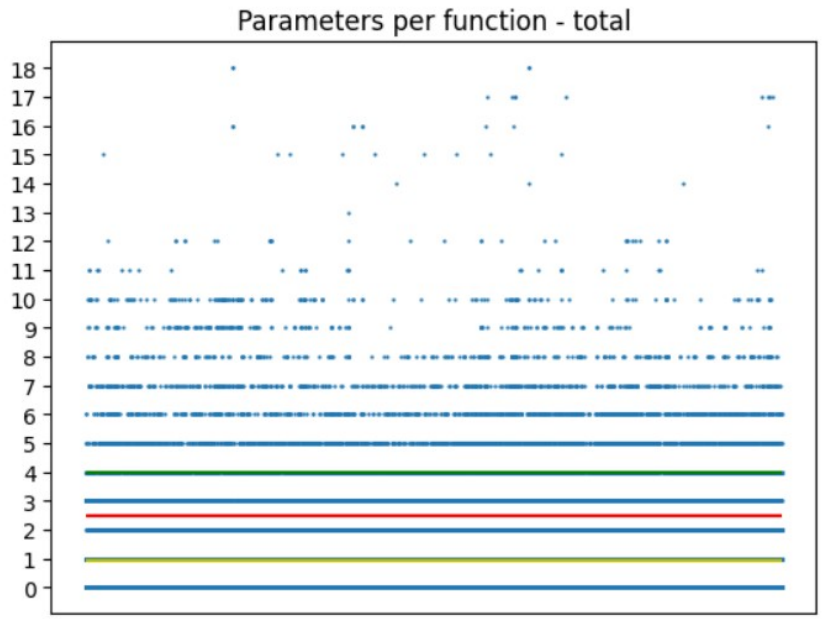

Figure 3.37: graph showing the parameters count per function for all sketches

After observing these graphs, the following comments can be made:

From comparing graphs 3.30 and 3.33, it can be observed that, contrary to the results observed at module/file level, the functions from hearted sketches present in general a lower count of lines than functions from created sketches. This, together with the information from 3.4.2, could be correlated to more modularized functions being present in hearted sketches, having shorter functions with less lines of code, but longer files with more lines due to the use of more functions in general.

In general, (see 3.36 and 3.35) it is possible to see that most sketches functions have a logical line count lower than 25, and a physical line count lower than 40, which is in general, readable and manageable functions.

It is possible to see on graph 3.37, that there is a very high concentration of functions that only take at most 5 parameters, being the most common functions taking no parameters at all. This makes sense having in mind what was presented in sections 3.4.2 and 3.4.2, where it was observed that the majority of total functions corresponds to functions setup and draw, these being the most common among sketches, and that these functions unusually receive no parameters as input.

Functions setup and draw

The functions setup and draw represent a very important part of JavaScript based creative coding projects. These two functions are the base for developing with the `p5.js` (figure 3.38) library, and therefore the most commonly used functions among sketches (as seen in 3.4.2). For this reason, it seemed like an interesting thing to further look at.

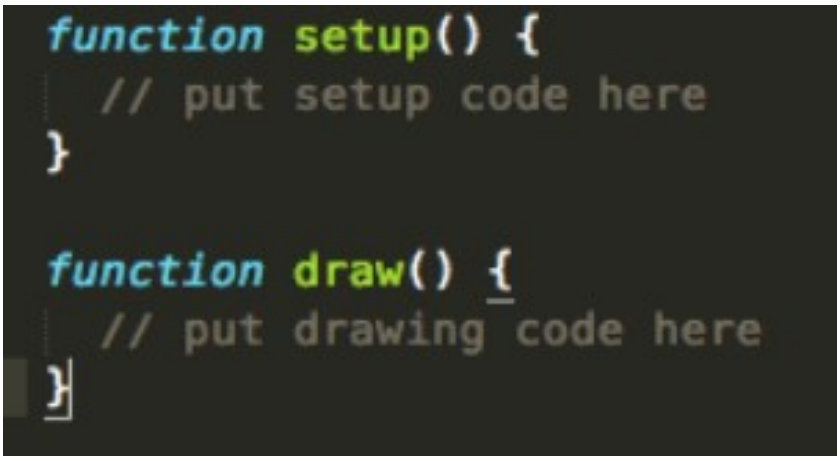

Figure 3.38: basic skeleton of a p5.js based sketch

Firstly, a look into just quantities. While going through the information for each file, it was counted how many of them had both setup and draw functions. Also, the total amount of draw and setup functions was obtained.

group	files with setup and draw	funcs setup	funcs draw	number files	% files with both	number sketches	% sketches with both
created	8840	9439	8941	12009	73.61	9673	91.39
hearted	7625	7978	7708	11566	65.93	8414	90.62
all	16465	17417	16649	23575	69.84	18068	91.13

Table 3.26: count of setup and draw functions, files containing both, and percentage of files and sketches containing them

A great majority of all JavaScript files contain both these functions, and assuming each sketch has only one file with both functions in it, around 9 in every 10 sketches uses these (see table 3.26).

Now, a further look into how these functions are coded will be presented. The following tables present statistics on parameter count and logical and physical lines of code for both setup and draw functions, for created, hearted, and all sketches.

	params_setup	loc_logical_setup	loc_physical_setup
count	9439.00	9439.00	9439.00
mean	0.00	10.49	12.72
std	0.01	141.19	43.88
min	0.00	0.00	1.00
25%	0.00	2.00	4.00
50%	0.00	3.00	5.00
75%	0.00	7.00	11.00
max	1.00	13465.00	3546.00

Table 3.27: statistics for setup function for created sketches

	params_draw	loc_logical_draw	loc_physical_draw
count	8941.00	8941.00	8941.00
mean	0.00	26.05	36.32
std	0.02	116.03	132.05
min	0.00	0.00	1.00
25%	0.00	5.00	8.00
50%	0.00	10.00	17.00
75%	0.00	23.00	35.00
max	2.00	5714.00	6588.00

Table 3.28: statistics for draw function for created sketches

	params_setup	loc_logical_setup	loc_physical_setup
count	7978.00	7978.00	7978.00
mean	0.00	11.33	16.71
std	0.03	45.54	48.89
min	0.00	0.00	1.00
25%	0.00	4.00	6.00
50%	0.00	7.00	11.00
75%	0.00	13.00	18.00
max	1.00	3755.00	3819.00

Table 3.29: statistics for setup function for hearted sketches

	params_draw	loc_logical_draw	loc_physical_draw
count	7708.00	7708.00	7708.00
mean	0.01	22.36	32.55
std	0.20	51.90	54.73
min	0.00	0.00	1.00
25%	0.00	6.00	11.00
50%	0.00	14.00	21.00
75%	0.00	25.00	36.00
max	4.00	1985.00	1533.00

Table 3.30: statistics for draw function for hearted sketches

	params_setup	loc_logical_setup	loc_physical_setup
count	17417.00	17417.00	17417.00
mean	0.00	10.88	14.55
std	0.02	108.41	46.29
min	0.00	0.00	1.00
25%	0.00	2.00	4.00
50%	0.00	5.00	8.00
75%	0.00	10.00	15.00
max	1.00	13465.00	3819.00

Table 3.31: statistics for setup function for all sketches

	params_draw	loc_logical_draw	loc_physical_draw
count	16649.00	16649.00	16649.00
mean	0.01	24.34	34.57
std	0.14	92.09	103.70
min	0.00	0.00	1.00
25%	0.00	5.00	9.00
50%	0.00	12.00	19.00
75%	0.00	24.00	35.00
max	4.00	5714.00	6588.00

Table 3.32: statistics for draw function for all sketches

From all these tables, it is easy to see that in all cases, both setup and draw functions unusually don't take parameters, with few exceptions, so it's not worth it to look into or further analyzing this metric.

In relation to both logical and physical lines of code, it is possible to see that in all subgroups, the 'draw' function is, on average, about twice as long as the 'setup' function. This can be explained by what these functions do, what they achieve and how they work. The draw function more often contains further calculations and even conditional code, and is run repeatedly, whereas the setup function is only run once, at initialization. This can be seen in more detail in `https://github.com/processing/p5.js/wiki/p5.js-overview` [24] where there's a detailed overview explanation of the library used to create these sketches.

Also, the difference between logical and physical lines is greater in the draw function. This again can be explained by the fact that the draw function more often contains further calculations and even conditional code, which leads to more lines dedicated to separating sections or comments explaining functionalities.

It is also possible to observe that in all cases, the maximum value escapes the other statistics by a lot, which gives the implication that these are isolated and specific cases. Either way, next, graphs are presented to show the concentration of the data for the total amount of sketches. For these, given the 'escaped' values, the y axis, representing the count of logical lines of code, will be limited on the top by the value percentile 99, showing still 99% of the data.

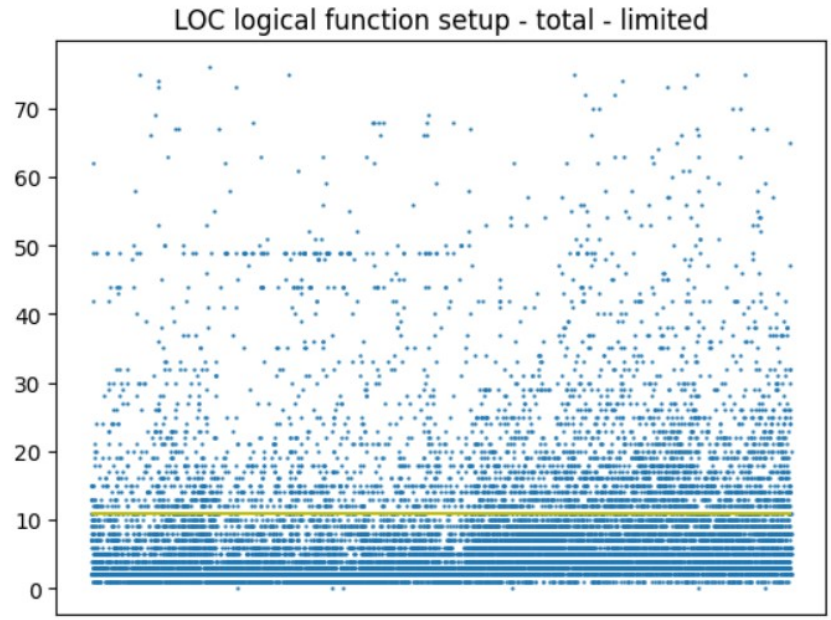

Figure 3.39: graph for logical lines of code for function setup for all sketches

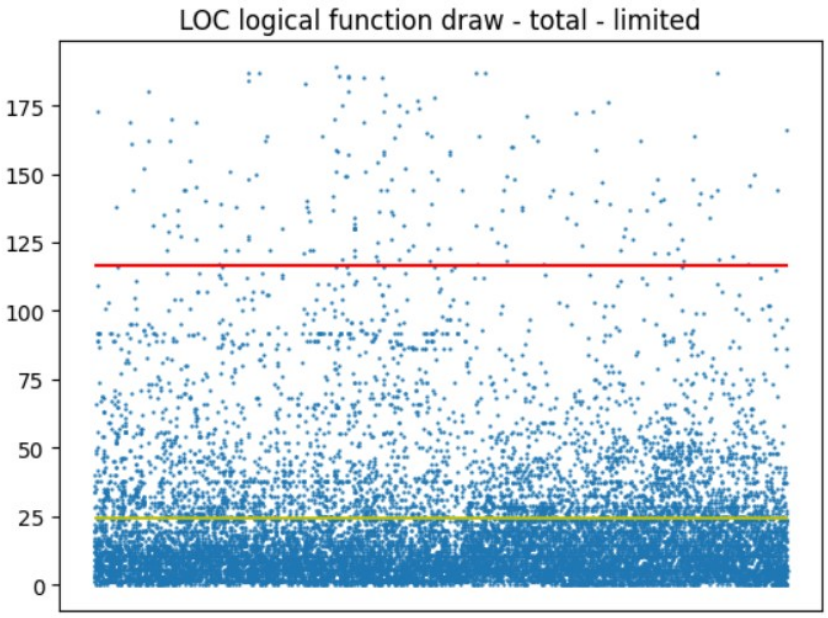

Figure 3.40: graph for logical lines of code for function draw for all sketches

In these graphs, each dot represents a sketch, the y axis represents the amount of lines, and the x axis, as in this case it's just an index for sketches, it is irrelevant. The yellow line represents the mean of the data.

It is possible to notice that for the setup function, there's a big concentration of sketches with under 10 lines, and under 20 lines for the draw function, leading to think that these functions are in most cases, very readable.

Metrics

In this last section, a further analysis is presented for various metrics. The metrics being looked at are cyclomatic complexity, cyclomatic complexity density, maintainability and Halstead metrics. These are presented and described both in a module or file level and function level.

Cyclomatic complexity and density

	cyclomatic per file	cyclomatic function avg per file	cyclomaticDensity per file
count	12009.00	12009.00	11313.00
mean	8.47	2.65	15.55
std	48.31	6.08	16.32
min	1.00	1.00	0.02
25%	1.00	1.00	7.14
50%	2.00	1.50	12.50
75%	5.00	2.40	18.75
max	3376.00	138.00	180.00

Table 3.33: statistics for cyclomatic complexity and density for files and average of cyclomatic complexity of functions per file from created sketches

	cyclomatic per file	cyclomatic function avg per file	cyclomaticDensity per file
count	11566.00	11566.00	10905.00
mean	13.25	2.64	18.33
std	82.74	4.20	18.74
min	1.00	1.00	0.01
25%	2.00	1.27	9.09
50%	5.00	2.00	13.33
75%	10.00	3.00	19.64
max	3376.00	240.00	160.00

Table 3.34: statistics for cyclomatic complexity and density for files and average of cyclomatic complexity of functions per file from hearted sketches

	cyclomatic per file	cyclomatic function avg per file	cyclomaticDensity per file
count	23575.00	23575.00	22218.00
mean	10.82	2.65	16.91
std	67.47	5.24	17.61
min	1.00	1.00	0.01
25%	1.00	1.00	8.33
50%	3.00	1.75	12.82
75%	7.00	2.67	19.15
max	3376.00	240.00	180.00

Table 3.35: statistics for cyclomatic complexity and density for files and average of cyclomatic complexity of functions per file from all sketches

	cyclomatic per func created	cyclomaticDensity per func created	cyclomatic per func hearted	cyclomaticDensity per func hearted	cyclomatic per func all	cyclomaticDensity per func all
count	45151.00	44658.00	78045.00	77081.00	123196.00	121739.00
mean	2.98	50.08	2.81	54.04	2.88	52.59
std	7.04	45.91	6.23	51.75	6.54	49.72
min	1.00	0.03	1.00	0.00	1.00	0.00
25%	1.00	20.00	1.00	20.00	1.00	20.00
50%	1.00	34.21	2.00	35.00	1.00	34.62
75%	3.00	66.67	3.00	100.00	3.00	100.00
max	234.00	1000.00	954.00	1000.00	954.00	1000.00

Table 3.36: statistics for cyclomatic complexity and density per function from created, hearted, and all sketches

It is important to notice that in the case of cyclomatic complexity density, the functions with no lines are ignored, given that the value of lines of code is used as the divisor in the calculation of the rate. That's why it can be observed that in the tables 3.33, 3.34, 3.35 and 3.36, the 'count' value is lower for cyclomatic complexity density than for cyclomatic complexity.

In all these metrics, for both file and functions, it can be seen in the tables that the maximum value escapes the mean and percentile data. For this reason, it's sensible to assume that these are extreme cases that escape the norm. For this reason, for the following graphs, the information will be capped on the top by the value equal to percentile 95. In other words, 95% of the lower-valued data will be shown.

Additionally, after observing that the statistics are very similar between hearted and created sketches, the graphs will only be presented for the aggregated data, that is, for all sketches.

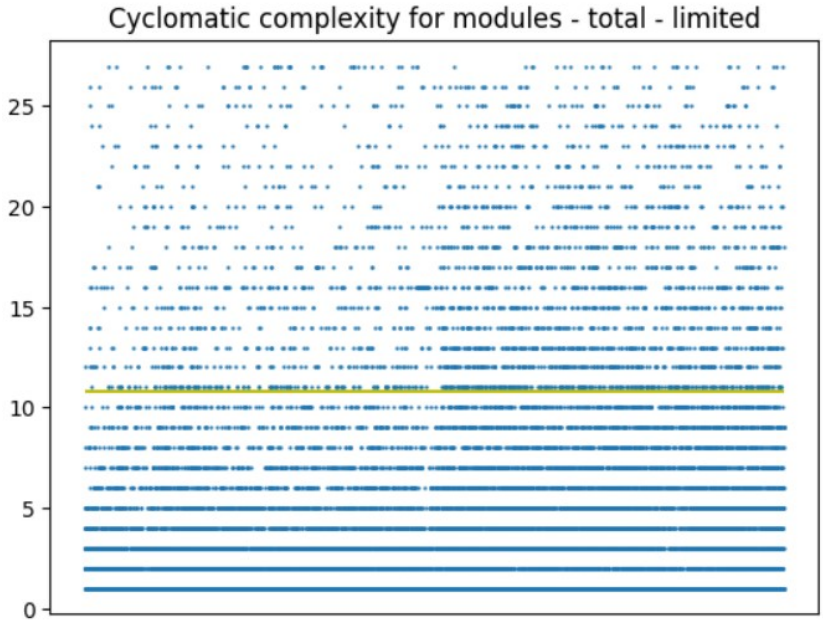

Figure 3.41: graph showing the cyclomatic complexity metric score for all sketches, with y axis limited on top by percentile 95

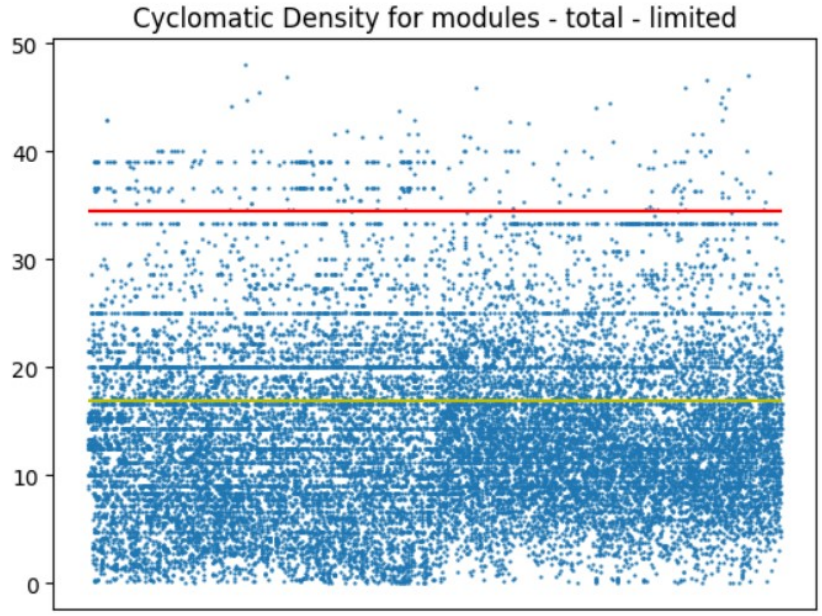

Figure 3.42: graph showing the cyclomatic complexity density for all sketches, with y axis limited on top by percentile 95

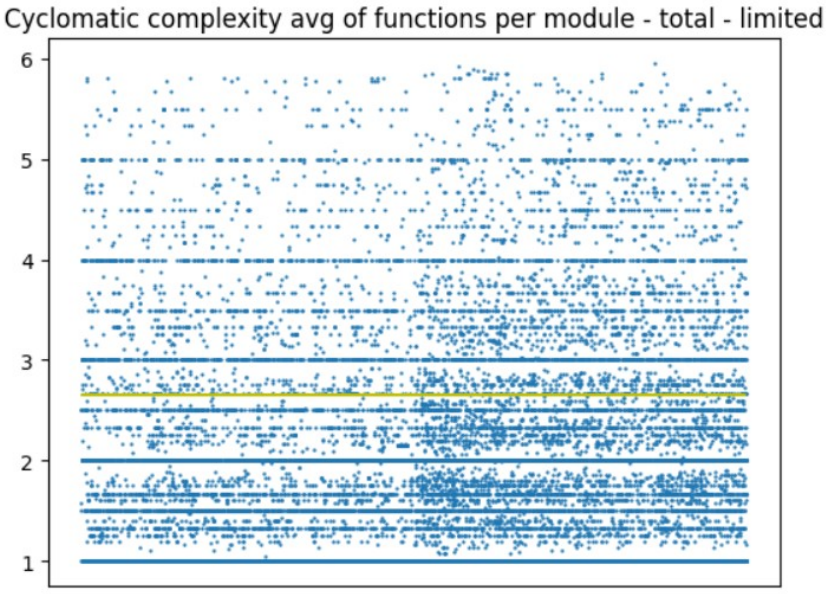

Figure 3.43: graph showing the average of cyclomatic complexity for the functions per file for all sketches, with y axis limited on top by percentile 95

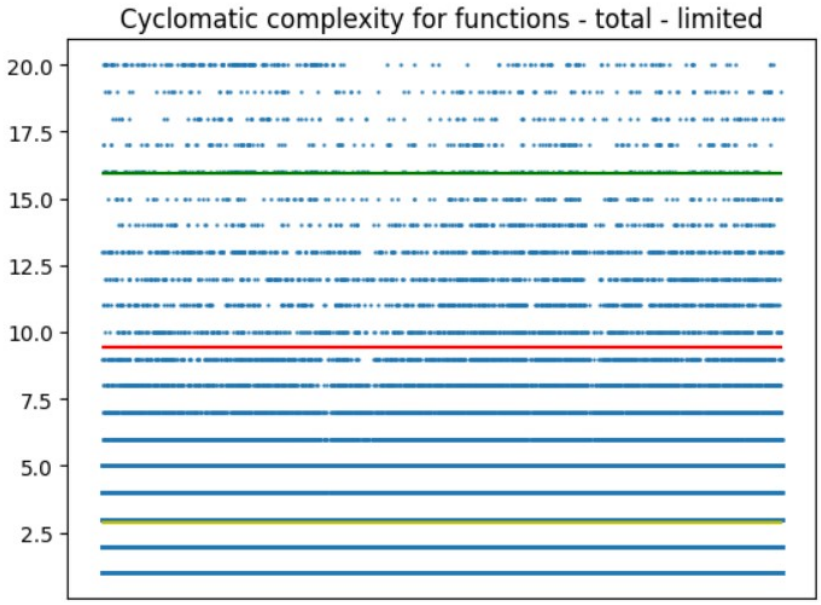

Figure 3.44: graph showing the cyclomatic complexity of functions for all sketches, with y axis limited on top by percentile 99

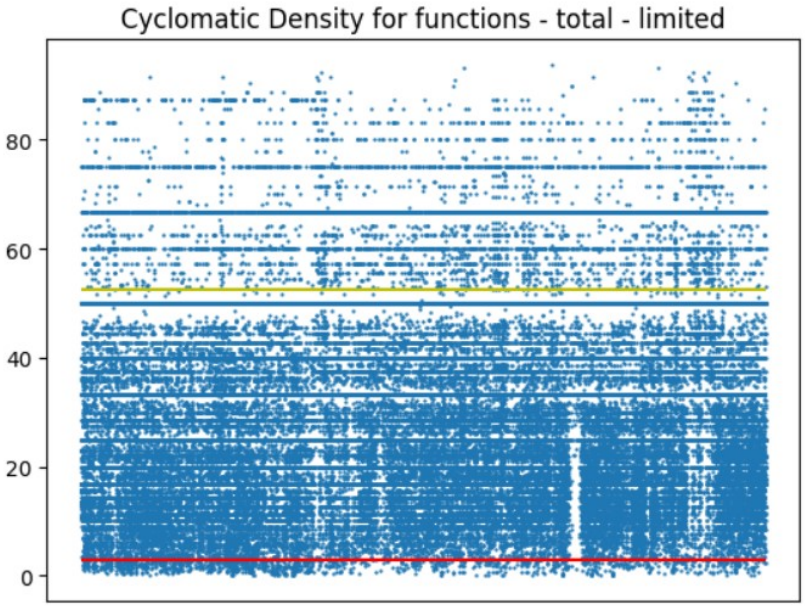

Figure 3.45: graph showing the cyclomatic complexity density of functions for all sketches, with y axis limited on top by percentile 95

From graphs 3.41 and 3.44, its possible to see that for both functions and files, the cyclomatic complexity mostly stays under a value of 10, which is usually considered to describe well written and structured code, that's highly testable and has low cost to modify.

For the case of average of functions (graph 3.43), and for the cyclomatic complexity density (graphs **??** and 3.45), some clusters can be seen, which can be explained by the standardized 'skeleton' of creative coding projects, that in general consists of very similar style code with functions setup and draw, and some other common functions. (see 3.4.2, 3.4.2).

Halstead

Per Module or File

	length per file	vocabulary per file	difficulty per file	volume per file	effort per file	bugs per file	time per file
count	12009.00	12009.00	12009.00	12009.00	1.200900e+04	12009.00	12009.00
mean	411.59	78.74	17.01	3306.01	4.933051e+05	1.10	27405.84
std	2456.67	576.71	30.02	27146.96	1.285014e+07	9.05	713896.71
min	0.00	0.00	0.00	0.00	0.000000e+00	0.00	0.00
25%	51.00	25.00	3.25	237.74	9.874200e+02	0.08	54.86
50%	123.00	45.00	9.65	670.64	6.361770e+03	0.22	353.43
75%	297.00	77.00	19.63	1849.58	3.283469e+04	0.62	1824.15
max	122886.00	35103.00	1351.35	1454096.52	9.509604e+08	484.70	52831135.65

Table 3.37: statistics of halstead metrics for files of created sketches

	length per file	vocabulary per file	difficulty per file	volume per file	effort per file	bugs per file	time per file
count	11566.00	11566.00	11566.00	11566.00	1.156600e+04	11566.00	1.156600e+04
mean	706.86	120.23	29.30	6540.80	1.349422e+06	2.18	7.496791e+04
std	5985.68	1263.02	62.26	88255.62	3.107965e+07	29.42	1.726647e+06
min	0.00	0.00	0.00	0.00	0.000000e+00	0.00	0.000000e+00
25%	107.00	42.00	9.73	582.23	5.809420e+03	0.19	3.227500e+02
50%	252.00	71.00	20.52	1549.49	3.224142e+04	0.52	1.791190e+03
75%	481.00	103.00	34.48	3209.55	1.093308e+05	1.07	6.073930e+03
max	340332.00	71958.00	4530.28	5435206.91	2.631289e+09	1811.74	1.461827e+08

Table 3.38: statistics of halstead metrics for files of hearted sketches

	length per file	vocabulary per file	difficulty per file	volume per file	effort per file	bugs per file	time per file
count	23575.00	23575.00	23575.00	23575.00	2.357500e+04	23575.00	2.357500e+04
mean	556.45	99.10	23.04	4893.01	9.133201e+05	1.63	5.074001e+04
std	4546.73	975.92	48.97	64801.08	2.362562e+07	21.60	1.312535e+06
min	0.00	0.00	0.00	0.00	0.000000e+00	0.00	0.000000e+00
25%	65.00	29.00	4.89	320.21	1.833180e+03	0.11	1.018400e+02
50%	179.00	58.00	14.15	1052.82	1.430778e+04	0.35	7.948800e+02
75%	404.00	93.00	28.34	2633.34	7.134817e+04	0.88	3.963790e+03
max	340332.00	71958.00	4530.28	5435206.91	2.631289e+09	1811.74	1.461827e+08

Table 3.39: statistics of halstead metrics for files of all sketches

Per Function

	length per function	vocabulary per function	difficulty per function	volume per function	effort per function	bugs per function	time per function
count	45151.00	45151.00	45151.00	45151.00	4.515100e+04	45151.00	45151.00
mean	94.06	25.41	7.91	594.14	4.619183e+04	0.20	2566.21
std	876.12	235.12	12.86	9306.29	3.923135e+06	3.10	217951.93
min	0.00	0.00	0.00	0.00	0.000000e+00	0.00	0.00
25%	10.00	8.00	1.20	30.00	3.619000e+01	0.01	2.01
50%	30.00	16.00	4.00	123.19	5.180000e+02	0.04	28.78
75%	81.00	30.00	10.00	398.51	3.720100e+03	0.13	206.67
max	106746.00	35101.00	542.08	1082219.96	5.754927e+08	360.74	31971816.96

Table 3.40: statistics of halstead metrics for functions of created sketches

	length per function	vocabulary per function	difficulty per function	volume per function	effort per function	bugs per function	time per function
count	78045.00	78045.00	78045.00	78045.00	7.804500e+04	78045.00	78045.00
mean	93.74	27.81	9.75	623.21	2.620433e+04	0.21	1455.80
std	1029.82	353.70	15.51	14901.38	9.531585e+05	4.97	52953.25
min	0.00	0.00	0.00	0.00	0.000000e+00	0.00	0.00
25%	14.00	9.00	2.18	44.97	1.007700e+02	0.01	5.60
50%	37.00	18.00	5.60	155.32	8.579200e+02	0.05	47.66
75%	87.00	31.00	12.00	432.66	4.965350e+03	0.14	275.85
max	166959.00	55549.00	1100.56	2631519.82	1.392335e+08	877.17	7735196.72

Table 3.41: statistics of halstead metrics for functions of hearted sketches

	length per function	vocabulary per function	difficulty per function	volume per function	effort per function	bugs per function	time per function
count	123196.00	123196.00	123196.00	123196.00	1.231960e+05	123196.00	123196.00
mean	93.86	26.93	9.08	612.56	3.352969e+04	0.20	1862.76
std	976.30	315.46	14.62	13130.51	2.493253e+06	4.38	138514.05
min	0.00	0.00	0.00	0.00	0.000000e+00	0.00	0.00
25%	13.00	9.00	1.80	39.86	7.430000e+01	0.01	4.13
50%	35.00	17.00	5.09	143.40	7.057200e+02	0.05	39.21
75%	85.00	31.00	11.25	422.34	4.483670e+03	0.14	249.09
max	166959.00	55549.00	1100.56	2631519.82	5.754927e+08	877.17	31971816.96

Table 3.42: statistics of halstead metrics for functions of all sketches

By looking at these tables a few observations can be made:

- The difficulty metric, which is generally interpreted as the difficulty to write or understand the code, presents on average low values, for all sets of sketches, for both functions and files, with an expected higher value for files. This is a good indicator that creators are writing manageable, readable code, which is specially relevant for open source code.

- On average, hearted sketches present a higher value of length and vocabulary, with a more noticeable difference when it comes to files. This can indicate more complex code that uses more variables, operands and operators. But, this could also mean a better use of different variables for different purposes.
- The bugs or errors, defined as an estimate for the number of errors in the implementation, presents expected larger values on file level than at function, but at file level, this metric shows to be higher in hearted sketches. Again, this can indicate more complex program.
- The effort and time measures, that represent the estimated mental effort and time for implementation in seconds respectively, show on average a lower value for functions in hearted sketches than in created sketches. This can signify simpler more atomized functions being implemented in hearted sketches. On the other hand, in the case of files, it's the opposite, where created sketches show lower values. This could indicate that files from hearted sketches are more complex as a whole.
- The time measure,on average it's set at around 14 hours for files and around 30 minutes for functions, which, including debugging and considering that creators could be inexperienced and the fact that sometimes a single file constitutes as a result a fully functioning sketch, these seem like reasonable times.

From the tables presented above, it can be seen that, as is common by now, the maximum value of all the different measures escapes the rest of the statistics by a lot, so it's reasonable to assume that these are extreme cases that escape the norm. For this reason, for the following graphs, the information will be capped on the top by the value equal to percentile 95. In other words, 95% of the lower-valued data will be shown.

Moreover, given the great amount of data and the fact that in general, created and hearted sketches present fairly similar statistics, graphs will be presented only for all sketches aggregated.

Per File

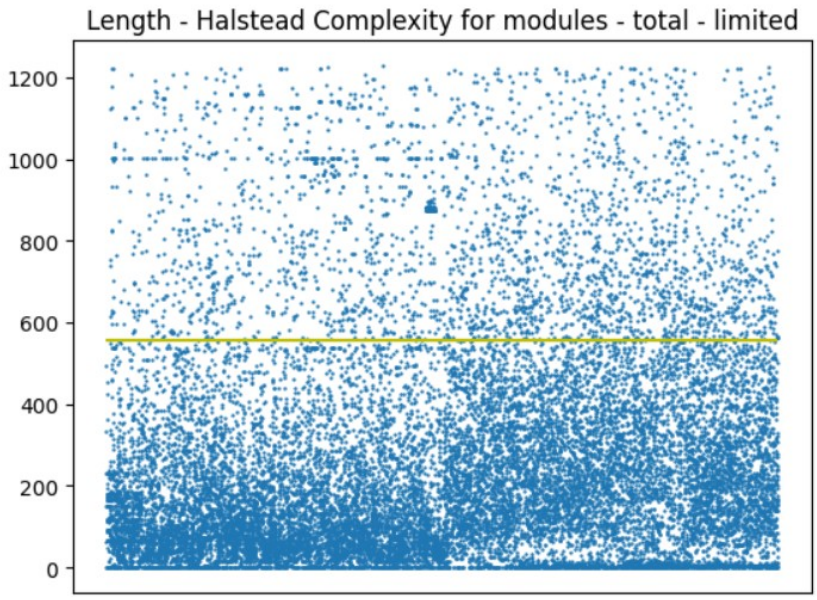

Figure 3.46: graph showing the length halstead metric scores for per file for all sketches, with y axis limited on top by percentile 95

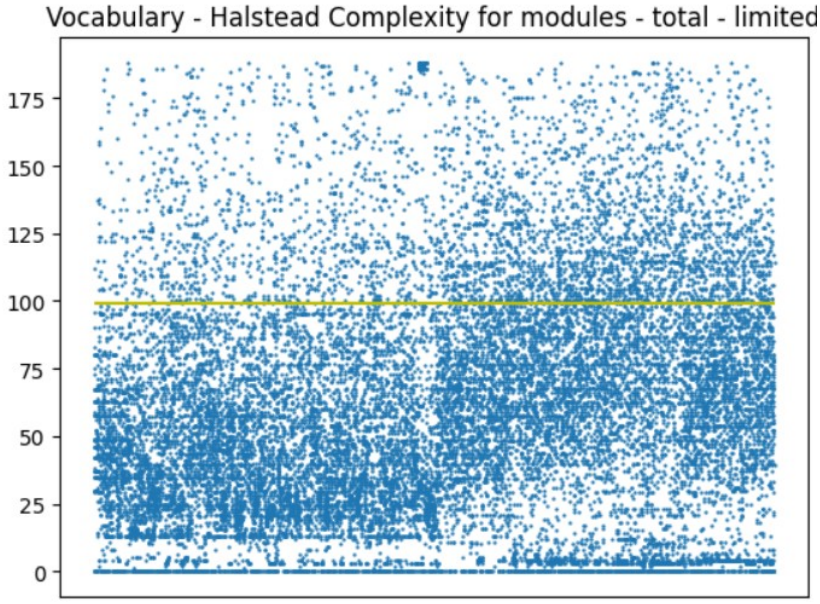

Figure 3.47: graph showing the vocabulary halstead metric scores for per file for all sketches, with y axis limited on top by percentile 95

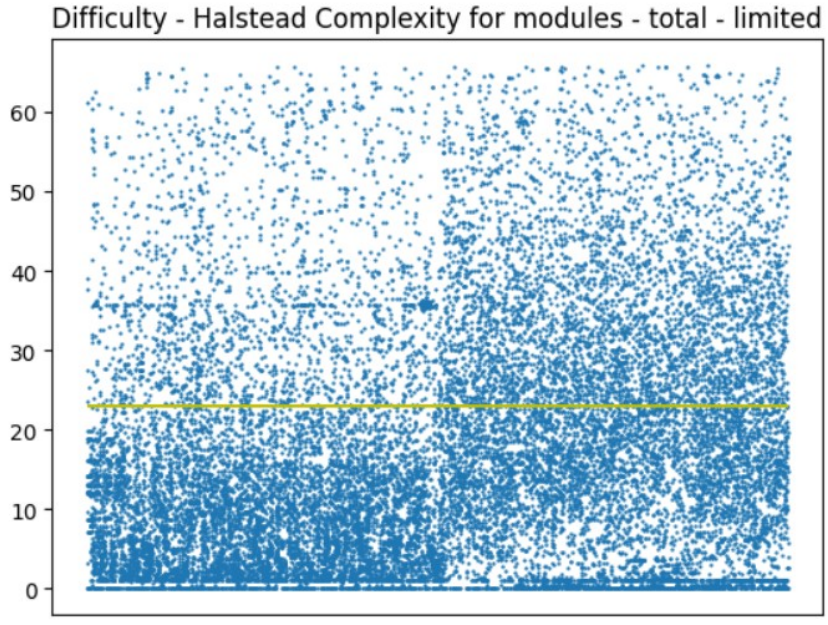

Figure 3.48: graph showing the difficulty halstead metric scores for per file for all sketches, with y axis limited on top by percentile 95

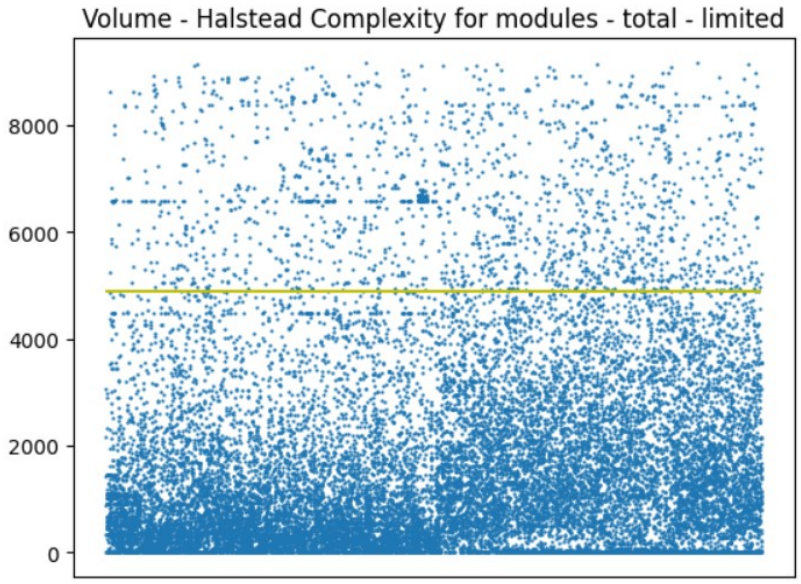

Figure 3.49: graph showing the volume halstead metric scores for per file for all sketches, with y axis limited on top by percentile 95

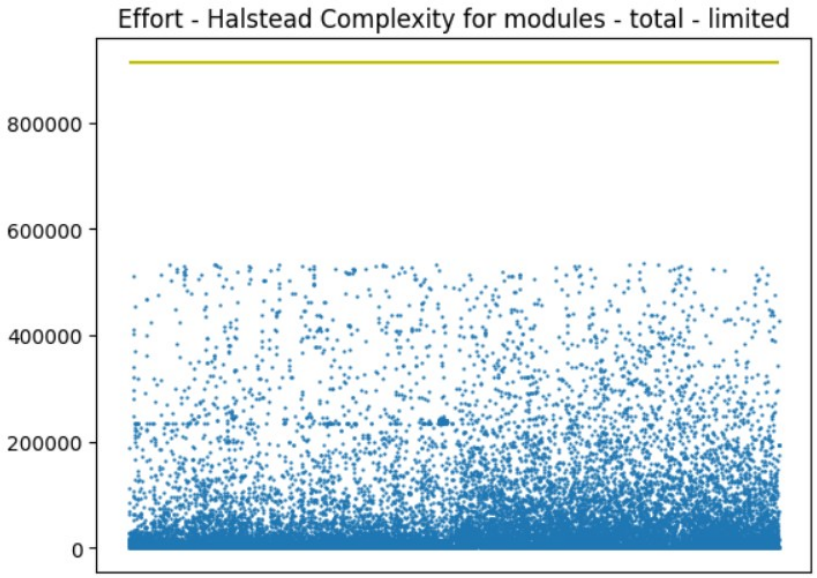

Figure 3.50: graph showing the effort halstead metric scores for per file for all sketches, with y axis limited on top by percentile 95

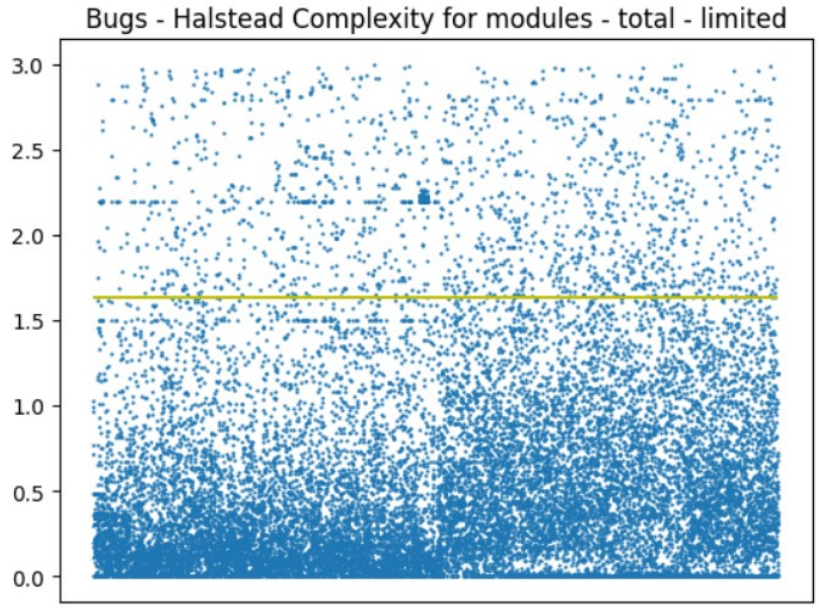

Figure 3.51: graph showing the bugs halstead metric scores for per file for all sketches, with y axis limited on top by percentile 95

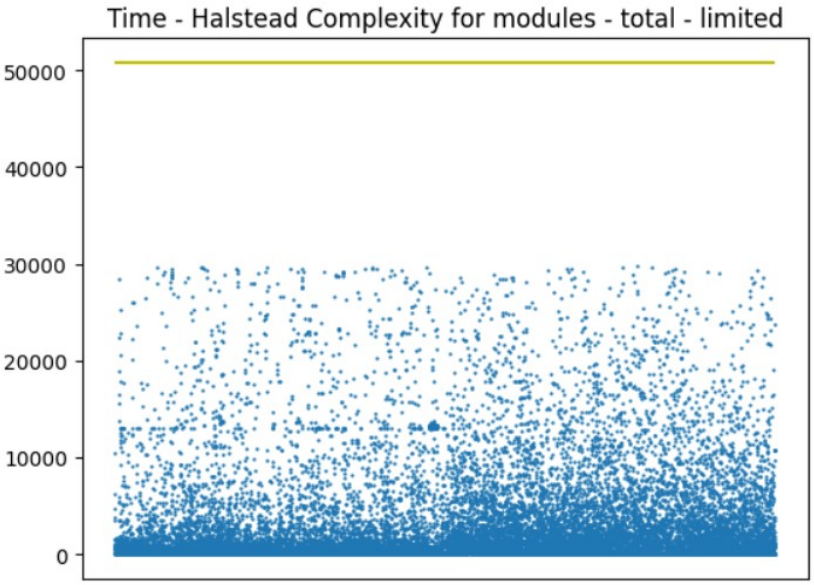

Figure 3.52: graph showing the time halstead metric scores for per file for all sketches, with y axis limited on top by percentile 95

Per Function

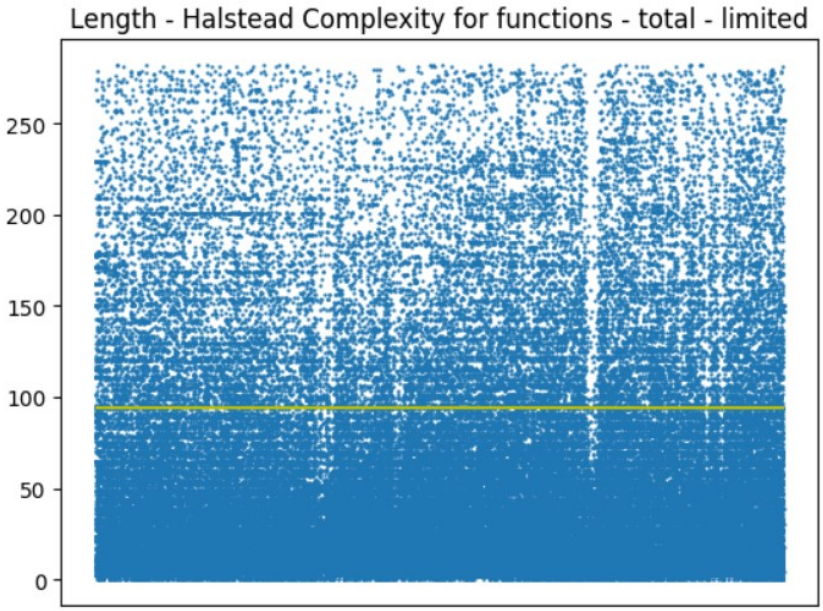

Figure 3.53: graph showing the length halstead metric scores for per file for function sketches, with y axis limited on top by percentile 95

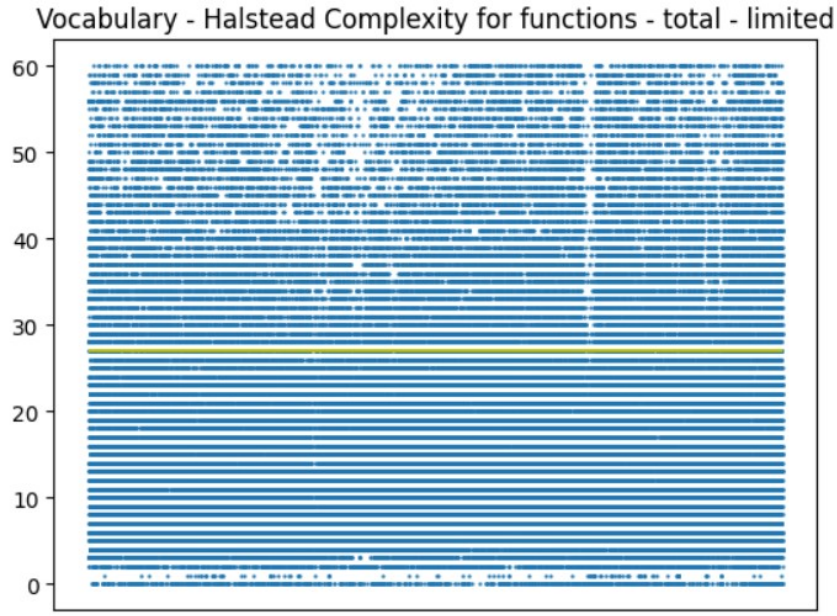

Figure 3.54: graph showing the vocabulary halstead metric scores for per function for all sketches, with y axis limited on top by percentile 95

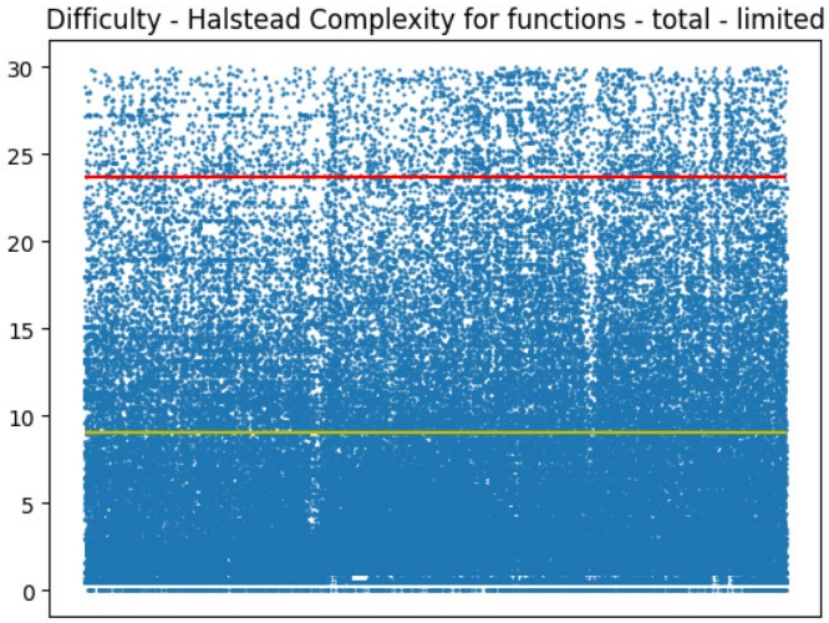

Figure 3.55: graph showing the difficulty halstead metric scores for per function for all sketches, with y axis limited on top by percentile 95

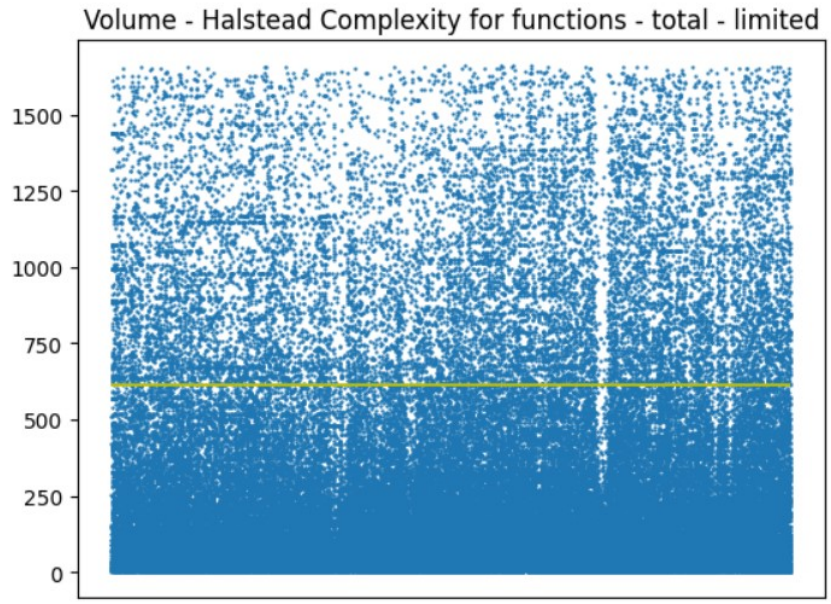

Figure 3.56: graph showing the volume halstead metric scores for per function for all sketches, with y axis limited on top by percentile 95

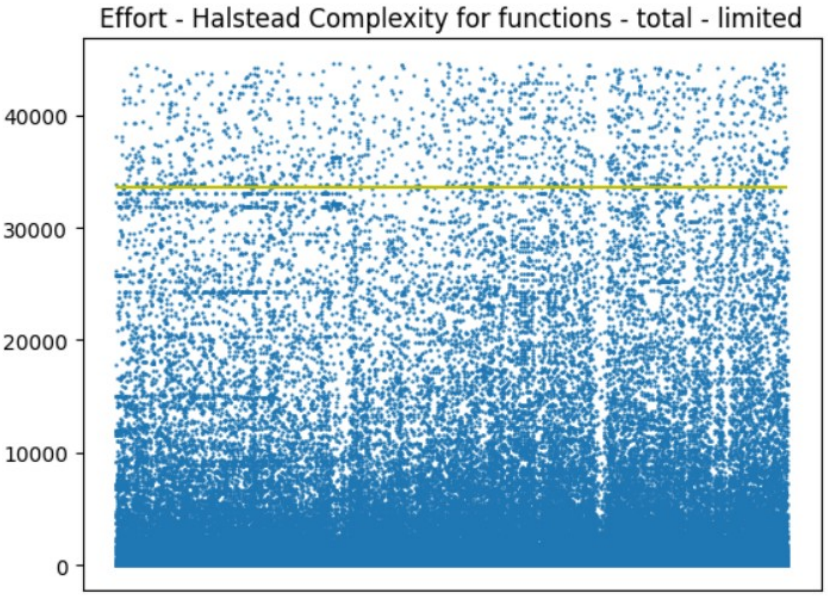

Figure 3.57: graph showing the effort halstead metric scores for per function for all sketches, with y axis limited on top by percentile 95

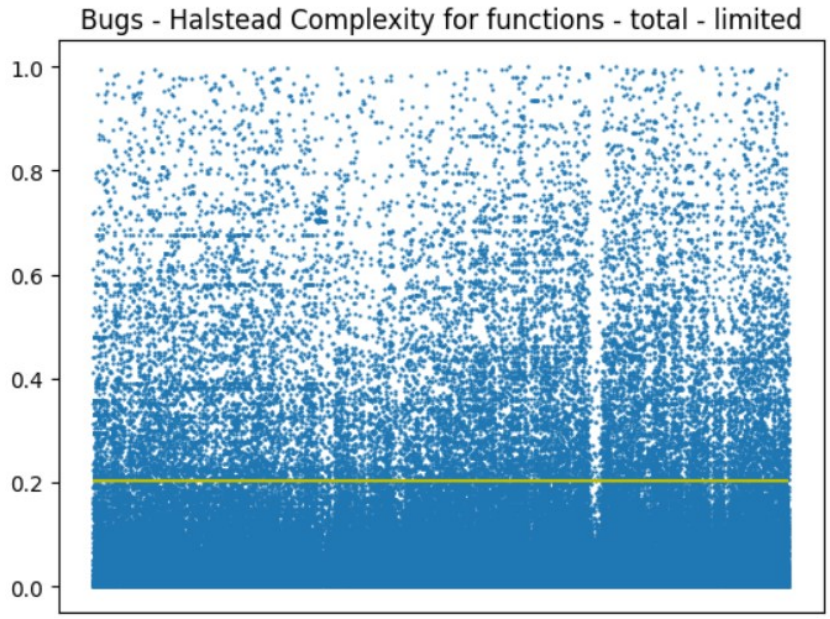

Figure 3.58: graph showing the bugs halstead metric scores for per function for all sketches, with y axis limited on top by percentile 95

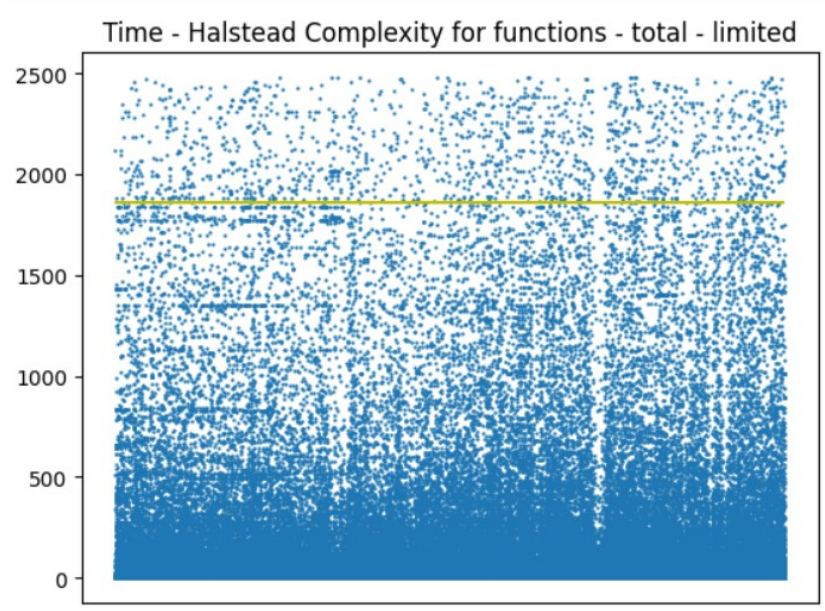

Figure 3.59: graph showing the time halstead metric scores for per function for all sketches, with y axis limited on top by percentile 95

In all these graphs it is possible to see that, in general, the values of the metrics presented, both at file and function levels, tend to stay towards lower values, which in the case of halstead metrics, lower values indicate better, more readable and maintainable code. The values show mostly around or lower than the mean (the yellow line in the graphs) and rarely coming close or over the value of mean plus standard deviation (represented by the red line in the graphs). This goes to show that the values are very concentrated, with a few cases of values that escape the norm a lot.

Maintainability

	maintainability per module created sketches	**maintainability per module hearted sketches**	**maintainability per module all sketches**
count	12009.00	11566.00	23575.00
mean	115.49	112.23	113.89
std	27.59	25.60	26.68
min	-30.08	-31.27	-31.27
25%	98.93	96.72	97.55
50%	114.00	107.65	110.47
75%	130.59	121.16	126.63
max	171.00	171.00	171.00

Table 3.43: statistics for maintainability index per module for created, hearted, and all sketches

By looking at this table it is clear that the statistics for the maintainability index across all types of sketches are very similar. In all cases, the maximum achievable 'score' is obtained. In the following graphs, it is possible to see the distribution of sketches in the range of values.

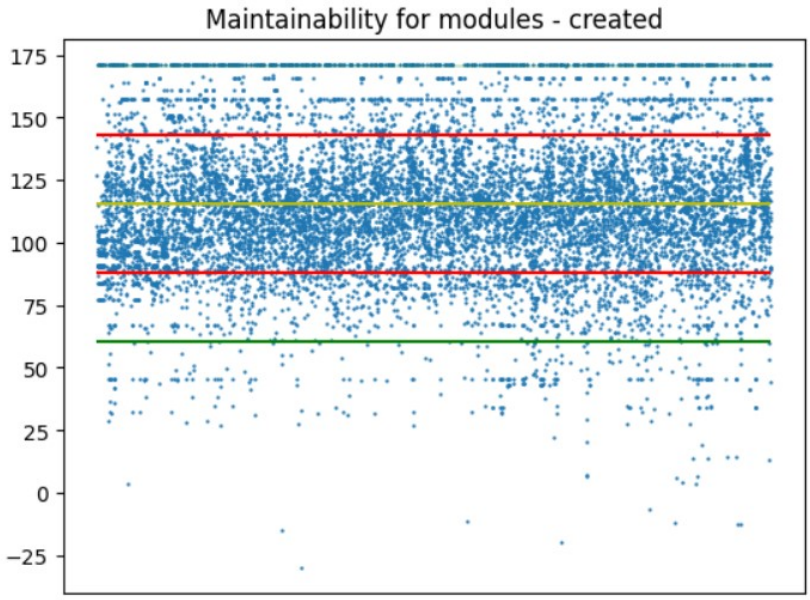

Figure 3.60: graph showing the maintainability index scores for created sketches

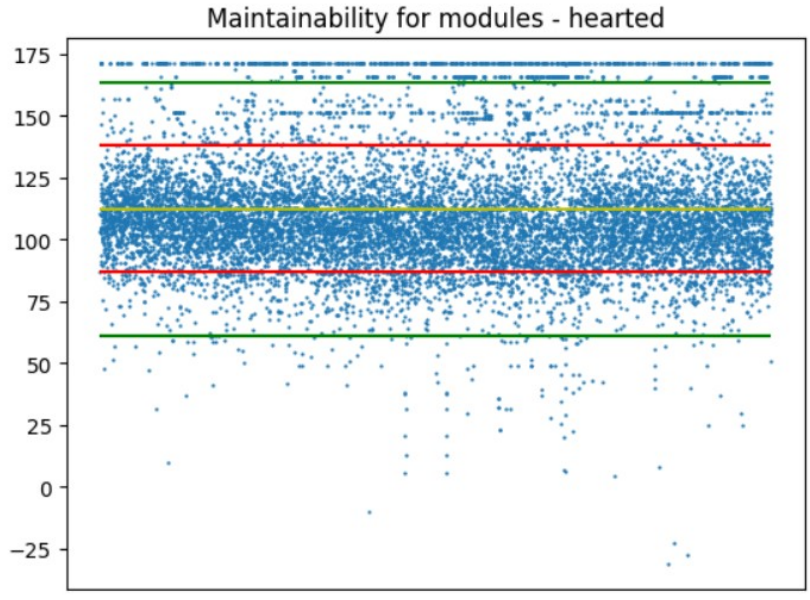

Figure 3.61: graph showing the maintainability index scores for hearted sketches

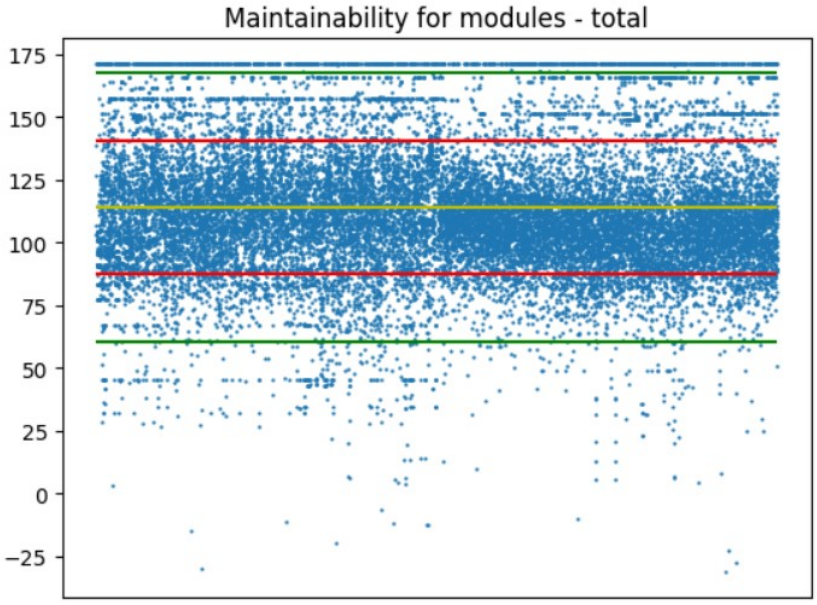

Figure 3.62: graph showing the maintainability index scores for all sketches

In these graphs, the yellow line represents the mean of the data, the red lines represent the mean plus and minus the standard deviation, and the green lines represent the mean plus and minus two times the standard deviation. If these exceed the range of the graph, the corresponding line is simply not shown. Each dot represents a file whose y axis value represents the score obtained, while the x axis, as in this case it's just an index for sketches, is irrelevant.

It is possible to see that the majority of files, in all cases, obtain a score that ranges between 75 and 130, which is commonly considered as good scores. A significant amount of files can be seen obtaining the maximum score which means these are very maintainable and readable programs. A few files can be seen in the lower part of the score range, even into the negatives, but, from table 3.43, it's possible to see that less than 25% percent of the data falls under a score of 90, which is still considered a high score, so it's safe to say, that the low scoring modules are isolated cases that deviate from the norm.

Chapter 4

Discussion

In this section, the results of the research and analysis that was carried out and detailed in the section above (3) will be presented and discussed in a summarized and conclusive manner, showing the main takeaways and observations obtained.

Additionally, some general conclusive comments and observations will be presented along with a brief proposal of some possible future research and work that could extend the results obtained in this thesis.

4.1 Results

After the analysis was carried out and its results presented, there are two main themes on which some conclusive observations can be made. These two being; the general patterns found in creative coding projects, with a more detailed description of a typical JavaScript based project, and observations in relation to metrics.

In relation to the general patterns and layout of the source code of creative coding projects, the first thing to mention is the fact that there are two types of projects. Sketches are either based and developed on the Processing language (file extension being pde), also referred to as Arduino Sketch, or they are based and developed on the language JavaScript (file extension being js), in conjunction with HTML.

In terms of the use of other languages or different media files other than JavaScript, HTML or Processing, to complement the creation of the sketch, it was found that the absolute majority of projects don't have any of these files present, with only about a 3% of total files belonging to these. It was observed that hearted sketches have a slightly higher percentage of these files in comparison to created

sketches, with the difference being about only 0.5%.

When it comes to Processing based projects, a higher percentage of these files is found among the subgroup of sketches denominated as hearted, with about 5% more files found than in the subgroup of created sketches. In general, considering all subgroups, about 20% of all existing files correspond to Arduino Sketch files. When considering the total number of sketches, about one third of sketches were found to be Processing based. This information is only an approximation based on the information that Processing based projects contain a single pde file, and that JavaScript based files tend to contain only one HTML file (more details on section 3.4.1).

The previously stated observation, and the fact of there being no tools for Arduino Sketch static code analysis, lead to only carrying a further analysis for JavaScript based projects. The following observations are made having only files from this language in mind.

If the typical 'skeleton' of a JavaScript based project wants to be defined, the first thing to mention is the fact that they are composed of at least one main JavaScript file, commonly named 'mySketch.js' (or in some cases 'sketch.js'). Some sketches contain more than only this JavaScript file, these extra files corresponding to either definition of functions then used by the main file or different semi-individual components to the sketch.

In all cases, the JavaScript file(s) is accompanied by one HTML file, that is in charge of embedding the script that defines the sketch written usually in the 'mySketch.js' file, into the web browser. Sometimes, the HTML will embed content from multiple JavaScrip files if the sketch's components are defined separately in different JavaScript files.

When it comes to the main file, 'mySketch.js', it is usually conformed by the setup and draw functions (see 3.4.2 and 3.4.2), sometimes exclusively or sometimes including other functions defined by the library `p5.js`, or other user defined functions.

In the case of hearted sketches, when compared with the created subgroup of sketches, a higher percentage of functions per file and per project was observed. They also presented a higher rate of files per project on average. Both of these observations can implicate a higher modularization, both of components of sketches and functions in general, and also possible a higher complexity of the sketches developed. (see 3.4.2)

In relation to different metrics analyzed in detail in section 3.4.2, it was observed that among all subgroups of sketches, the statistics and values obtained showed to be very similar. In the case of the hearted subgroup of sketches, they showed, on average, slightly higher values on metrics related to complexity, which goes along with the results and conclusions mentioned above.

When it came to the values of the metrics themselves, all the data showed on average what would be considered as "good and acceptable ranges". With the exception of a few records that 'escaped' the tendencies of all other records, for complexity related metrics, the values were on average low, and in the case of maintainability indexes sketches showed values tending to the higher segment of the range. All this implies that creators in general have code that is maintainable, readable, manageable and not too complex.

4.2 Conclusion and Future Research

After the research done in this thesis, observing the state of the art and the results obtained and presented, it appears that JavaScript, along with the library `p5.js`, is a language and way of developing creative coding projects that is gaining popularity over the Processing language among creators. The library in question seems to be a very accessible way for people with no expertise in programming to be able to code and experiment with creative coding.

It is important to notice, that even though at source code level, these projects seem to have a very similar structure and characteristics, the results are very dynamic and diverse, and many of them involve the interaction of the user to actually create a meaningful result. For this reason, it is very hard to discern and actually identify what makes a good sketch good by just looking at the code alone. In general, it's hard to obtain many concrete conclusions without also analyzing visually what the result of these projects is.

The only differentiation in use was the division of sketches into hearted sketches and created or recent sketches. When using this as a factor to determine 'good' sketches, it seems that better, more modularized but at the same time more complex code, seems to lead to more appealing sketches that receive more hearts.

The problem that only having the source code of projects without the output of it could present an opportunity for further research in the future, where more data about the popularity and visual results of the sketches is considered.

For the case of this thesis, it was deemed that only the JavaScript files were worth looking at and analyzing in detail, leaving the Processing sketches out of the research, since, besides the great difference in quantity, there were no tools found that could analyze this language for the factors desired. This could also be an opportunity for further future research, along with the creation of a tool to analyze this language.

Finally, a possible area to further investigate is the use of external files (not JavaScript, HTML or Processing). The functions and uses of these could be investigated, how they can help the creation of better creative coding projects and why they are currently not being used much. Furthermore, from that investigation, tools to help creators take advantage of these types of files could be created and developed.

Bibliography

[1] Maddie Ball. *What is creative coding?* Oct. 2019. URL: https://www.arts.ac.uk/study-at-ual/short-courses/stories/how-to-start-creative-coding#:~:text=What%20is%20creative%20coding%3F,and%20the%20wider%20design%20industry. (cit. on p. 1).

[2] Donovan Alexander. *Everything You Need to Know About the Artistic World of Creative Coding.* Nov. 2020. URL: https://interestingengineering.com/culture/everything-you-need-to-know-about-the-artistic-world-of-creative-coding (cit. on p. 1).

[3] Wikipedia contributors. *Creative coding — Wikipedia, The Free Encyclopedia.* 2004. URL: https://en.wikipedia.org/wiki/Creative_coding (cit. on p. 1).

[4] HiSoUR Arte Cultura Historia. *VJing.* URL: https://www.hisour.com/vjing-2984/ (cit. on p. 1).

[5] Martin Perez. *What is Web Scraping and What is it Used For?* Apr. 2023. URL: https://www.parsehub.com/blog/what-is-web-scraping/ (cit. on p. 3).

[6] Kinsta. *¿Qué Es el Web Scraping? Cómo Extraer Legalmente el Contenido de la Web.* 2022. URL: https://kinsta.com/es/base-de-conocimiento/que-es-web-scraping/#:~:text=El%20web%20scraping%20se%20refiere,precios%20de%20varias%20tiendas%20online (cit. on p. 3).

[7] Srishti Saha. *The Economy of the Web Scraping Industry.* 2018. URL: https://www.blog.datahut.co/post/the-economy-of-the-web-scraping-industry (cit. on p. 3).

[8] Wikipedia contributors. *Web scraping — Wikipedia, The Free Encyclopedia.* 2004. URL: https://en.wikipedia.org/wiki/Web_scraping (cit. on p. 3).

[9] Baiju Muthukadan. *Selenium with Python.* URL: https://selenium-python.readthedocs.io/ (cit. on p. 4).

[10] Albert Danial. *cloc: v1.96.* Version v1.96. Dec. 2022. DOI: 10.5281/zenodo.5760077. URL: https://doi.org/10.5281/zenodo.5760077 (cit. on p. 10).

[11] Albert Danial. *Selenium with Python*. URL: https://github.com/AlDanial/cloc (cit. on pp. 10, 11).

[12] Jared Stilwell. *complexity-report*. URL: https://github.com/escomplex/complexity-report (cit. on pp. 12, 13).

[13] Jared Stilwell. *escomplex*. URL: https://github.com/escomplex/escomplex/blob/master/ (cit. on p. 14).

[14] PVS-studio. *Source Lines of Code*. July 2013. URL: https://pvs-studio.com/en/blog/terms/0086/ (cit. on pp. 15, 16).

[15] IBM Corporation. *Cyclomatic complexity*. Mar. 2021. URL: https://www.ibm.com/docs/en/raa/6.1?topic=metrics-cyclomatic-complexity (cit. on p. 16).

[16] Gate Vidyalay. *Cyclomatic Complexity | Calculation | Examples*. URL: https://www.gatevidyalay.com/cyclomatic-complexity-calculation-examples/ (cit. on p. 16).

[17] Microsoft. *Code metrics values*. Oct. 2022. URL: https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022 (cit. on pp. 16, 17).

[18] IBM Corporation. *Complexity Reports*. June 2021. URL: https://www.ibm.com/docs/en/addi/5.1.0?topic=reports-complexity (cit. on p. 16).

[19] Geoffrey K. Gill and Chris F. Kemerer. «Cyclomatic complexity density and software maintenance productivity». In: *IEEE Trans.Softw* (Dec. 1991). URL: https://sites.pitt.edu/~ckemerer/CK%20research%20papers/CyclomaticComplexityDensity_GillKemerer91.pdf (cit. on p. 16).

[20] IBM Corporation. *Halstead Metrics*. Mar. 2021. URL: https://www.ibm.com/docs/en/rtr/8.0.0?topic=SSSHUF_8.0.0/com.ibm.rational.testrt.studio.doc/topics/csmhalstead.html (cit. on p. 17).

[21] GeeksforGeeks. *Software Engineering | Halstead's Software Metrics*. Apr. 2023. URL: https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/ (cit. on p. 17).

[22] JavaTpoint. *Halstead's Software Metrics*. URL: https://www.javatpoint.com/software-engineering-halsteads-software-metrics (cit. on p. 17).

[23] Microsoft. *Code metrics - Maintainability index range and meaning*. Apr. 2022. URL: https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2022 (cit. on p. 17).

[24] Processing Foundation. *p5.js*. URL: https://github.com/processing/p5.js (cit. on p. 58).

[25] Processing Foundation. *p5.js*. URL: https://p5js.org/.

[26] Processing Foundation. *Processing*. URL: https://processing.org/.