FACULTY OF COMPUTING, ENGINEERING AND MEDIA

EXPLORING A GRAPHICS RENDERING FRAMEWORK FOR CLIENT-SIDE WEB APPLICATIONS

BY ADAM EVANS

DE MONTFORT UNIVERSITY
LEICESTER

A DISSERTATION SUBMITTED IN PARTIAL FULFILMENT FOR THE DEGREE OF
BATCHELOR OF SCIENCE IN COMPUTER SCIENCE

MAY 2025

Acknowledgements

I would like to thank my supervisor, Dr. Stuart O'Connor, for his guidance and feedback throughout the course of this project. His input was always helpful in refining the direction and scope of its research and development. I would also like to thank my partner for being enduringly supportive; I am very grateful for the many, many cups of tea and coffee.

Abstract

Computer graphics are an essential component within modern technology and their use in client-side applications has greatly improved web standards, allowing the flexibility to dynamically generate complex animations, data visualisations, even online multiplayer games. The HTML5 Canvas API, provides the foundation for many of these applications however it can impact development and maintainability of code due to its very low-level implementation. This has prompted the development of libraries that all aim to simplify these interactions and similarly, this project aims to reduce complexity using a rendering framework that includes a cohesive graphics library alongside a domain-specific syntax that extends vanilla JavaScript.

This project explores the impact that programming languages can have, quantitatively comparing the code complexity of the developed system against that of the Canvas API and other libraries. In doing so the developed framework yielded an average reduction of 36% compared to its equivalent implementation using the Canvas API, enhancing the reduction made by the p5.js library by a further 25%. This effectively demonstrates that, by introducing syntax and patterns employed by other rendering solutions, the development experience and evaluated maintainability of client-side applications can be greatly influenced by the programming languages used to write them.

Contents

Figures

Tables

Equations

1 Introduction

The development of computer graphics can be complex, especially when scaling the complexity of the graphics being rendered. This can be exacerbated when working with web applications, due to the moderately primitive nature of native interfaces provided, which is further highlighted by the large number of libraries that have been developed to simplify interactions with said APIs. While each of these libraries make notable improvements, they can often introduce their own complexity through boiler-plate code and data structure implementations. To explore how the existing technologies can be improved upon, this report will detail the development of a cohesive framework that builds upon their core functionalities while reducing their code complexity, making canvas rendering interactions more intuitive and accessible.

1.1 Background

Web applications form a large part of modern technology, accounting for most people's interactions with computing systems, and their ubiquity and complexity has only grown with improved capabilities of modern browsers. The role of computer graphics in web apps cannot be understated as it underpins visualisation of the entire *Document Object Model* (DOM) from scalable text and icons to interactive inputs and buttons. Beyond this, use of the HTML `<canvas>` element provides the *Canvas API*, allowing more intricate graphics to be dynamically rendered within web applications. This has not only expanded the available options for client-side data visualisation and user interaction but has even facilitated development of entire cross-platform browser games.

Having developed several personal projects that rely on the Canvas API, both directly and via other rendering libraries, interactive graphics and animations have often been the driving force behind their creation. During development however, these projects would regularly abstract the common or repeated functionality that occurred throughout their interaction with the Canvas API, with one such project beginning to consolidate these into a library of its own. In addition to the experience with the Canvas API, having worked with tools such as *Unity*, *Processing*, and shader languages has provided insight into the potential for a more consistent and cohesive approach within web applications.

1.2 Project Aims

This project investigates the development of a framework that collates the best patterns and principals of various graphics rendering systems, incorporating them into consistent and modular sets of functionalities, tailored for client-side graphics rendering. It aims to reduce both the boiler-plate code needed for implementing graphics, as well as the complexity of code that fulfils business logic, all while remaining suitable for a variety of applications including data visualisation, web apps, browser games and more.

Given that the development of any client-side application requires the use of *JavaScript*, the language's declarative rendering and lack of useful data structures such as vectors and matrices can make implementing vector graphics more complex than it potentially

need be. As such, this project aims to augment the existing JavaScript syntax, focussing on vector manipulation and effective integration with the graphics library also developed.

To evaluate the efficacy of the developed system, the complexity of the resultant code will be analysed and quantitively compared to that of other existing solutions. As this project introduces a novel programming syntax, this report will review the existing methods of code complexity analysis and implement an effective means of measuring these metrics, for both the extended syntax as well as vanilla JavaScript.

1.3 Project Deliverables

The developed system intends to include four key components: a graphics library, a transpiler, a complexity analyser, and an example application. The library will form the core logic of the framework and will be written in *TypeScript*, making it usable as a standalone import in any JavaScript or TypeScript project, without employing the framework's extended syntax. Should the extended syntax be used however, there would need to be a means by which it can be interpreted. In the case of the developed framework, a transpiler will parse and convert the extended syntax to vanilla JavaScript.

To effectively demonstrate the capabilities of the developed framework, an example application will be developed that uses the functionality offered to highlight common use cases. Additionally, a tool will be created to analyse the code complexity of comparable libraries against this framework, for an established set of benchmark graphic renderings. This data will go on to be analysed, forming the basis of the system's evaluation and informing the discussion throughout this report.

1.4 Personal Objectives

In addition to the quantitative and qualitative objectives outlined above, this project aims to expand understanding of programming language interpretation and design; studying how code complexity is impacted by a domain specific language and how that, in turn, impacts software maintainability. Moreover, this project also looks to further understanding of system architecture and software design by creating a framework containing several components, with a variety of uses, that operate together cohesively.

2 Research

This development project touches on two core concepts that form the basis for its research – **computer graphics** and **code complexity**. The former is an essential part of modern technology and has been instrumental in the adoption of computers by professionals and consumers alike. Since Ivan Sutherland implemented a complete *Graphical User Interface* (GUI) in 1963 for his program *Sketchpad* (Sutherland, Rodden and Kuhn, 2003), complicated systems have become far more accessible for wider audiences.

> *"The term computer graphics includes almost everything on computers that is not text or sound. Today ... people have even come to expect to control their computer through icons and pictures"*
>
> *(Cornell University, 1998)*

Today, with 94.4% of all relevant[1] websites using *HTML5* (W3Techs, 2025a), *its* introduction of the Canvas API has enabled developers to include real-time data visualisations and even full games into web applications able to run within almost any browser. Despite the canvas element's benefits, it only offers low-level APIs (MDN, 2025) that can contribute to increased complexity in the code written for any reasonably complicated graphics. This circles back conveniently to the latter of the two core concepts, code complexity. By looking at both concepts within the context of web application development, this section aims to elucidate both the research surrounding the use of vector graphics in web application development, as well as the impacts and considerations surrounding code complexity. Using the information gained from existing research, this project will then apply the understanding to the development of a vector graphics framework.

2.1 Computer Graphics

When exploring the possible improvements able to be introduced into the existing client-side environment for graphics rendering, it is crucial to widen the scope of investigation into all aspects of computer graphics. By looking into the way computers render graphics, how dedicated hardware can accelerate this process, and how the specificity of programming languages can support the implementation of computer graphics, this section will outline the usage of computer graphics within both the HTML specification and other third-party frameworks within web applications.

[1] A relevant website is considered to have meaningful content or functionality. (W3Techs, 2025b)

2.1.1 Raster vs. Vector Images

A computer has two principal means with which to digitally represent an image, the simplest being an image composed from a grid of pixels, a so-called **bitmap** or **raster** image. Alternatively, a **vector** image is described by a series of mathematical expressions that define the individual lines and curves programmatically, requiring the computer to redraw the image from scratch. As shown in Figure 2.1, for a raster image, the processor would iterate over each pixel of the image, displaying the appropriate colour value for that pixel. Nowadays, this is done incredibly rapidly, and several file formats such as JPEG, PNG, and GIF have popularised this technology on the web due to their efficient compression creating small files more appropriate for transfer over the internet.

Despite raster graphics being the de facto means of rendering on modern displays from CRT monitors to LED screens, there are considerable drawbacks, as they are bound by the resolution defined in the image being rendered. Exemplified by the low-resolution raster image in Figure 2.1, the per-pixel rendering of the image causes a phenomenon known as aliasing, which leads to the noticeable stair-case effect with the affectionate name **jaggies**. This highlights by far the largest benefit of vector versus raster, as vector images are programmatically rendered, and their resolution is determined by the image's consumer image, not its creator.

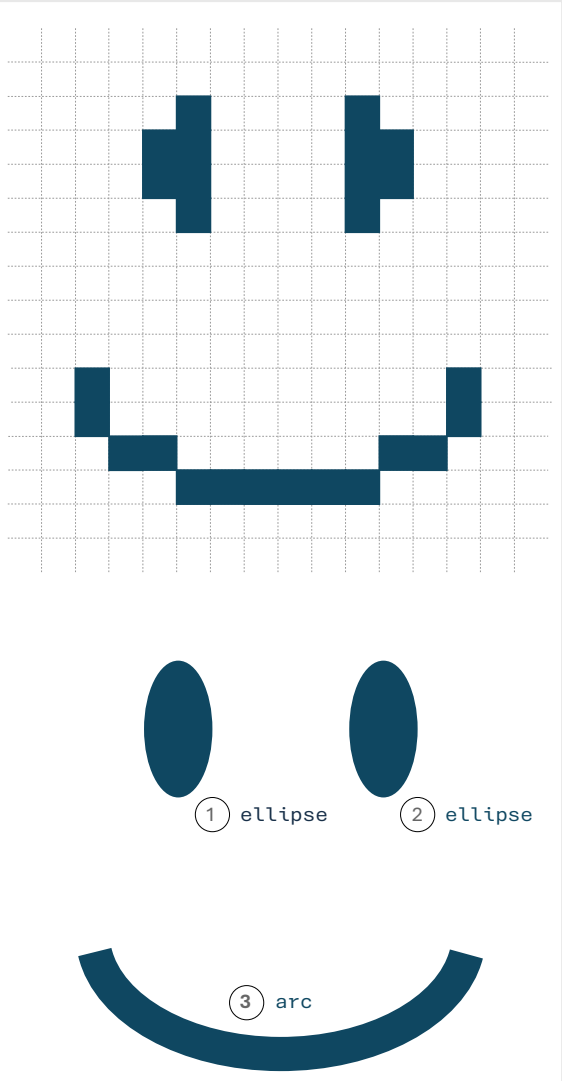


Figure 2.1 – Simplified examples of raster (top) and vector (bottom) images.

Because of this, vector graphics are scalable and are often far smaller in physical size given that they contain only the instructions to generate the raster image, not the raster image itself. Developed by the World Wide Web Consortium, *Scalable Vector Graphics* (SVG) were designed to be compatible with many other common web standards such as HTML and DOM, as well as JavaScript and *Cascading Style Sheets* (CSS). This has led to widespread use of vector graphics to instil responsive and accessible web-design, with the editors of Wikipedia pages going as far as to label any raster images that could later be reuploaded as SVG to more efficiently and accurately store the data (Wikipedia, 2023).

2.1.2 The HTML Canvas API

Given the comprehensive capabilities of vector graphics, it is understandable why their use is so prevalent throughout the web. With the introduction of the `<canvas>` element

in the *HTML5* specification, the ability to integrate complex graphics and animation into web applications has become far more accessible. Popular tools such *p5.js* as have created entire educational and artistic communities by leveraging the Canvas API (p5.js, 2025) and beyond that, HTML5 browser games have grown from the pixelated dinosaur developed by Google (Biersdorfer, 2017) to large multiplayer online games with globally ranked competitions (TagPro League, 2025).

> *"The Canvas API is extremely powerful, but not always simple to use. [Many libraries] can make the creation of canvas-based projects faster and easier."*
>
> *(MDN, 2025)*

Although the Canvas API goes a long way to facilitate web-based vector graphics, the numerous libraries that exist to simplify its usage clearly highlight the low-level nature of the provided functionality. The API implements a `CanvasRenderingContext2D` interface with which lines, and a basic set of 2D primitives, can be drawn onto the canvas. A prime example of the interface's complexity is found in drawing circles which, while seemingly a simple task, can prove somewhat cumbersome when compared to third-party libraries. As shown in Table 2.1, the manner with which circles are rendered is greatly simplified in each library, not only due to the addition of a dedicated circle method or class, but insofar that they also handle the drawing of those bound within the appropriate scope. The Canvas API, on the other hand, additionally requires a call to the `fill()` and/or `stroke()` methods after each call to ellipse. Beyond that, the interaction with CSS rapidly becomes obligatory, should styling be of any concern.

Table 2.1 – Code for rendering a circle in various vector graphics libraries.

Library	Code
Canvas API	`_.ellipse(50, 50, 25, 25, 0, 0, Math.PI * 2)`
p5.js	`_.circle(50, 50, 25)`
PixiJS	`new Circle(50, 50, 25)`
Paper.js	`new Shape.Circle(new Point(50, 50), 25)`
Konva	`new Konva.Circle(config)`

2.1.3 Language and Syntax

Rendering circles merely touches upon the brevity afforded by third-party libraries over the native Canvas API, with functionality for colour manipulation being a similar example. However, equally, these libraries each have their own individual benefits and drawbacks which need to be considered before their use within a project. One example is that many of these libraries, those listed in Table 2.1 inclusive, make extensive use of class structures. Given that JavaScript classes create objects that are instantiated with

their functionality built in, this can create a more fluid and simple coding style that can improve readability albeit at the expense of promoting mutability, as shown in Table 2.2.

Table 2.2 – Code for adding two vectors using OOP and functional paradigms.

Object-oriented	Functional
`const v1 = new Vector(1, 2, 3)`	`const v1 = [1, 2, 3]`
`const v2 = new Vector(2, 3, 4)`	`const v2 = [2, 3, 4]`
`v1.add(v2)`	`const v3 = add(v1, v2)`
`console.log(v1) // [3, 5, 7]`	`console.log(v1) // [1, 2, 3]`
`console.log(v2) // [2, 3, 4]`	`console.log(v2) // [2, 3, 4]`
	`console.log(v3) // [3, 5, 7]`

In the JavaScript ecosphere, large projects such as *React* have shifted the focus of their framework to employ more functional paradigms over those used in traditional *Object-Oriented Programming* (OOP) (Abramov, 2019). At the inaugural demonstration of hooks, Abramov, Alpert and Florence, (2018) claimed that *"classes are hard for humans ... [and] classes are also hard for machines"*, and with introduction of concepts such as immutable state, functional components and hooks, React has greatly reduced developers' dependence upon them.

Moving away from JavaScript, it should be noted how other languages provide functionality that supports common requirements of vector graphics. The popular game engine Unity is built upon C# which provides syntax such as **type casting** and **operator overloading**, the latter of which allows manipulation of objects using common operators such as `+`, `-`, `/`, and `*`, as opposed to `add()`, `sub()`, `div()`, and `mult()`.

Although the scope of this project does not include the development of shaders and 3D graphics, they remain a central component to graphics rendering and languages such as *OpenGL Shading Language* (GLSL) have allowed developers to accelerate the calculations required for calculating transformations, colours, and more. Due to the nature of GLSL, it too contains syntax specific to vector calculations including **infix operators** for vectors and matrices, as well as a unique syntax called **swizzling** that enables simple extraction of vector components as shown below in Table 2.3.

Table 2.3 – Code required to create vectors using dot notation and swizzling.

Dot notation	Swizzling
`v1 = vec3(1.0, 2.0, 3.0)`	`v1 = vec3(1.0, 2.0, 3.0)`
`v2 = vec3(v1.x, v1.x, v1.y)`	`v2 = v1.xxy`
`v3 = vec2(v1.z, v1.y)`	`v3 = v1.zy`

As this section has shown, the language used to write a program can greatly impact both the functionality and readability of its code, and further discussed in the following section, it remains a keenly discussed topic as to whether the complexity of source code can notably impact the efficiency of development and the future maintainability of services.

2.2 Code Complexity

From the earliest days of computer programming, there have been several ways in which the 'human' complexity of source code can purportedly be measured. This section aims to not only outline the most common interpretations of code complexity but also aims to investigate the utility and efficacy of their implementations.

> *"You can't control what you can't measure."*
>
> *(DeMarco, 1982)*

The word complexity bears many meanings in the world of computer science, with the complexity of time and space likely holding much more weight in the mind of developers than that of the literal code written. Modern technology stacks will often employ the use of **code linting,** to create consistency within source code, specifying extensive rules that enforce how it is structured and written. These linting tools, such as the popular *ESLint* for JavaScript and TypeScript, can easily be included into development pipelines and are often supported by *Integrated Development Environments* (IDEs), allowing code to be checked whilst it is written. This can prove to be an indispensable tool for developers, providing automatic formatting and refactoring, as well as in development pipelines to permit only linted code to be merged into an upstream repository.

2.2.1 Lines of Code

In its infancy, the complexity of source code was correlated with the number of **source lines of code** (SLOC or LOC) in a program or file (McCabe, 1976, p. 308; Humphrey, 1997, p. 15). Generally, this includes comments and whitespace, however penalising developers for documentation and formatting is somewhat inhibitive, and thus only counting the **logical lines of code** (LLOC) may provide a better indication of the actual complexity.

Although the limitation of physical code size has long been used to maintain modularity and reliability (Hatton, 1993; McCabe, 1976; Fenton and Niel, 1999), unconditional enforcement of this limitation may lead to the inadvertent introduction of more defects. Basili and Perricone, (1983) noticed a phenomenon whereby the number of defects found in code was not proportional to the number of lines, conversely revealing a comparable error rate in both small and large files. This is supported by Fenton and Niel, (1999) as well as V. Y. Shen et al. (1985) who, in their concluding remarks, suggest to *"promote programming practices related to modularization that discourage the development of either extremely large or extremely small modules"*.

2.2.2 Cyclomatic Complexity

First introduced by Thomas McCabe (1976), **cyclomatic complexity** leverages the mathematical field of graph theory to analyse the possible paths a given program could take. By analysing control structures used within the source code itself, a **control graph** is constructed from the **edges** and **vertices** of a given program. From that control graph, the cyclomatic complexity can then be derived applying the equation below:

Equation 2.1 – Cyclomatic complexity of program g with edges e, vertices n, and connected nodes p.

$$v(g) = e - n + p$$

(McCabe, 1976, p. 308)

This is better shown in Figure 2.2 and Figure 2.3, where each `if` statement creates a fork in the program that is conditionally dependent on the internal state at runtime. Despite being written somewhat verbosely, just this simple example can clearly demonstrate the potential for complexity within source code. With the introduction of switch statements and loops, these control graphs, and code they represent, could easily increase human cognitive demand (Hao et al., 2023).

```
function fizzbuzz(n) {
  if (n % 3 == 0) {
    if (n % 5 == 0) {
      return "fizzbuzz"
    } else {
      return "fizz"
    }
  } else if (n % 5 == 0) {
    return "buzz"
  } else {
    return ""
  }
}
```

Figure 2.2 – Code for `fizzbuzz`.

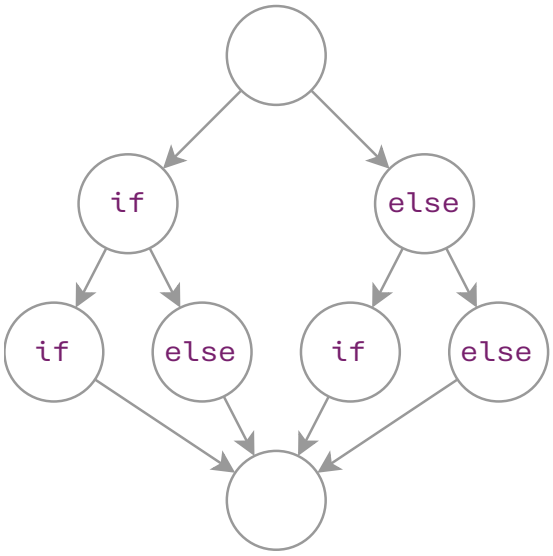


Figure 2.3 – Control graph for `fizzbuzz`.

Equally difficult as interpreting complicated graphs, creating a graph for each program would prove untenable in any large-scale system. McCabe (1976) acknowledges this, going on to prove that the cyclomatic complexity of any program is equal to the number of predicates, plus one. Confirming this using his initial formula in Equation 2.1, the cyclomatic complexity of `fizzbuzz` can be evaluated as $10 - 8 + 2$ or $3 + 1$, which fortunately for graph theory, both equal four.

While a standardised limit has not yet been agreed upon, the consensus is that 10-15 is appropriate for most applications (McCabe, 1976, p. 314) and given its apparent simplicity, the `fizzbuzz` function makes a notably non-negligible attempt at this threshold. Many IDEs and linters can check and rewrite code to reduce the complexity and McCabe (1976) posited that a program's control graph can be used to evaluate the code coverage provided by its test suite, which continues to be a widely used static analysis tool in modern software development.

2.2.3 Halstead Complexity Measures

Moving to a less esoteric measure of complexity, the eponymous Maurice H. Halstead (1977, cited in Felician and Zalateu, 1989, p. 1630) introduced a distinctly different set of metrics for measuring complexity. He proposed, by counting the individual **operators** and **operands** from which a program is comprised, several values could be deduced such as the estimated number of bugs present within the code, as well as the estimated effort and time required to write and maintain said code.

For a given program g, let the number of unique operators and operands be η_1 and η_2, with total counts of N_1 and N_2, respectively. Shown in the following equations, Halstead defined calculations for the **vocabulary** η, **length** N, **volume** V, and **difficulty** D of the program, using these in turn to estimate its **effort** E, **time** T, and **bugs** B.

Equation 2.2 – Halstead vocabulary of program g.

$$\eta = \eta_1 + \eta_2$$

Equation 2.3 – Halstead length of program g.

$$N = N_1 + N_2$$

Equation 2.4 – Halstead volume of program g.

$$V = N \log_2 \eta$$

Equation 2.5 – Halstead difficulty of program g.

$$D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$$

Equation 2.6 – Halstead effort of program g.

$$E = D \times V$$

Equation 2.7 – Halstead time of program g.

$$T = \frac{E}{18}$$

Equation 2.8 – Halstead bugs of program g.

$$B = \frac{E^{\frac{2}{3}}}{3000} \approx \frac{V}{3000}$$

(Halstead, 1977, cited in T. Hariprasad et al., 2017, p. 1110)

Using these formulae, the `fizzbuzz` program in Figure 2.2 could be evaluated as shown in the following tables below:

Table 2.4 – Operators and operands used in `fizzbuzz`.

Count	Operator	Count	Operand
1	`function`	1	`fizzbuzz`
4	`()`	4	`n`
5	`{}`	1	`3`
3	`if`	2	`5`
3	`else`	1	`"fizzbuzz"`
3	`%`	1	`"fizz"`
3	`==`	1	`"buzz"`
4	`return`	1	`""`
26	**8**	**12**	**8**

The Halstead metrics, unlike SLOC and cyclomatic complexity, leans heavily into lexical analysis of the code. In both Equation 2.5 and Equation 2.4, the direct proportionality of a program's operators to its volume and difficulty, and by virtue of Equation 2.6 its effort, show that these metrics promote smaller programs with fewer operators. Additionally, Equation 2.5 also shows the difficulty to be proportional to the ratio between total and distinct operands, implying that a program's code becomes easier to understand with a greater reuse of its operands.

Table 2.5 – Halstead metrics for `fizzbuzz`.

Metric	Value	Metric	Value
η_1	$= 8$	V	$= 38 \times \log_2 16 = 45.8$
η_2	$= 8$	D	$= 8/2 \times 12/8 = 6$
N_1	$= 26$	E	$= 45.8 \times 6 = 274.5$
N_2	$= 12$	T	$= 274.5/18 = 15.3$
η	$= 8 + 8 = 16$	B	$= 45.8/3000 = 0.015$
N	$= 26 + 12 = 38$		

2.2.4 Other Measures and Derivations

In addition to cyclomatic complexity, the **cyclomatic density** represents the quotient of cyclomatic complexity and LLOC, which normalises the measurements for programs of differing size. This, along with the other metrics mentioned, could be used to better highlight the worst-offending code, providing insight useful that effectively informs project planning and development (T. Hariprasad et al., 2017; McCabe, 1976). Furthermore, by combining the previously discussed metrics, D. Coleman et al. (1994) proposed a means of evaluating the **maintainability index** (MI) of a system. Their polynomial relies on the average Halstead volume V_{avg}, average extended[2] cyclomatic complexity $v(g')_{avg}$, as well as the average lines of code LOC_{avg} and percentage of those that are comments $C_{\%}$.

Equation 2.9 – Maintainability index of program g.

$$MI = 171 - 5.1 \ln V_{avg} - 0.23 \cdot v(g')_{avg} - 16.2 \ln LOC_{avg} + 50 \sin(\sqrt{2.46 \cdot C_{\%}})$$

(D. Coleman et al., 1994)

Outside of those mentioned, there are a wide variety of complexity metrics and variants thereof, all with an equally wide variety of benefits and considerations. As briefly touched upon earlier, ubiquitous implementation of complexity limitation can become a drawback; each measure should be carefully applied to ensure it supports development of maintainable code as opposed to hindering it.

2.2.5 Perception and Utility

A study by Hao et al. (2023) questioned the accuracy of various complexity measures, specifically that of cyclomatic complexity, when 27 participants were asked to analyse three different Java programs whilst their eye movement and brain activity were monitored. The three programs were constructed to have complexity scores that reflected three tiers of difficulty, however despite their initial predictions, they found the resultant cyclomatic complexity of the first two programs were not indicative of the cognitive load exhibited by developers. They go on to say that variables, which are not accounted for in cyclomatic complexity, have a significant impact on code comprehension and are more

[2] Extended cyclomatic complexity not only accounts for control structures but also conditional operators.

proportionally represented by the Halstead metrics. This is supported by T. Hariprasad et al. (2017) but, conversely, Flater (2018, p. 8) claimed that the Halstead metrics were *"a less plausible, more confusing set of metrics for coding effort"* a statement, in turn, supported by Fenton and Niel, (1999). Flater (2018, p. 7) additionally questions the validity of the divisor in Equation 2.7, known as the **Stroud number**, claiming that the value of 18 was defined by Halstead to *"obtain the best fit of [his] model to a sample of data"*.

This just goes to show how the perception of complexity measures can vary broadly, as can its implementations and consequences. If used as metrics for productivity or even true indicators of complexity, they may cause unwanted complexity in their stringency. In lieu of this, they are best used to highlight where complexity may have arisen over time, informing the decisions that steer the direction of project planning and development.

Given the discussed shortcomings of the Canvas API, and how complexity can impact the development and maintenance, a framework that implements the simpler paradigms and beneficial syntax of other languages could greatly improve the developer experience when creating graphics within web applications. The proposed system aims to not only create a library that greatly improves upon the Canvas API but also improve upon other comparable libraries by extending JavaScript with new syntax, such as that explored in section 2.1.3. To validate the success of the developed framework, it will be compared to other existing systems using the complexity measures outlined in section 2.2. In the following section, the system development and design will be reviewed before going on to analyse the collected data and discussing the systems overall efficacy.

3 Development

To begin implementing the desired system, the project was broken down into constituent components, outlining their required functionality to fulfil a **minimum viable product**. By structuring work to iteratively build each component, the focus of development was to implement fully operational features that could easily be extended and modified in future iterations. Throughout development, documentation and testing were both key considerations, with automated testing and test-driven development being used where possible. Additionally, the various common standards for both application design and system architecture were investigated prior to implementing features, taking advantage of the well-documented patterns and libraries that are used by many existing projects.

3.1 Components

To maintain a single source code repository with several distinct components, most of which use JavaScript, using `yarn` package manager along with *workspaces* allows each component to be self-contained while better enabling management of dependencies and versioning across the entire project. Figure 3.1 shows each package within `grafig`, and their interactions, highlighting their logically isolated, yet unified design.

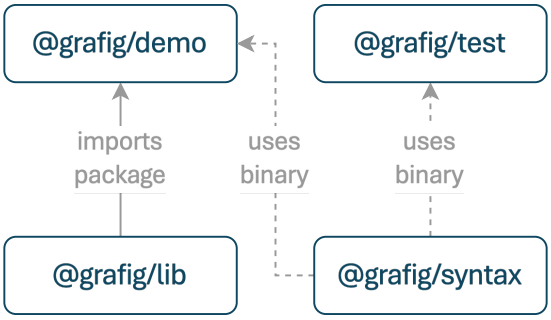


Figure 3.1 – System package structure.

3.1.1 Graphics Library

The *@grafig/lib* package contains the core functionality of the framework and can be used as a standalone package, making it not too dissimilar to other existing libraries. This library, however, avoids any reliance on globally hoisted variables and instead opts for a more functional and immutable approach. The library groups exported functionality into separate modules, shown in Figure 3.2, with each being minimally reliant on the others. This allows the `vector` module to be used in isolation within any project needing vector functionality, and for the `shape` and `style` modules to be used without importing the `rendering` module. As the library is designed with modularity and extendibility in mind, new functionality is easily able to be implemented in an incremental fashion.

The library is written in TypeScript, and makes uses of strict typing throughout, allowing for usage in any TypeScript or JavaScript project while enabling tools such as *IntelliSense* to provide autocompletion and type checking. This promotes *shift-left* methodology as, for example, by using generic types, function overloading, and type narrowing, the type inference is greatly improved and allows type errors to be caught much earlier on in the development pipeline. The vector module exemplifies this as it overloads each function to validate the parameters and return types, as shown in Appendix A, where the `from()` will validate the object type given to it based on the given keys used to specify the new vector's components.

Figure 3.2 – Module and export structure for @grafig/lib

Another design decision was to implement closures within rendering elements, for example, the `Canvas()` function takes a `CanvasProps` object and a closure defining how to manipulate the underlying rendering context. This abstracts the use of the Canvas API, providing a more consistent and comprehensive interface that can be further expanded with future iterations. This can be seen in Appendix B where the `Canvas()` function provides a context object to the given closure, greatly reducing the boiler plate code needed to access the rendering context in comparison to the Canvas API. Concepts such as closures and function overloads are fundamental to implementing the extended syntax which, as will be discussed in the following section, works in tandem with the *@grafig/lib* package to further simplify development of vector graphics.

3.1.2 Extended Syntax

The existing JavaScript specification, due to its vast array of uses, offers a broad syntax that can create minimally complex code but when applied to the development of vector graphics it can prove overly verbose. These additional features introduced by the new syntax, *FigScript*, specifically aim to reduce code complexity when interacting with the HTML `<canvas>` and, as it is interoperable with both HTML and JavaScript, it is suitable for us in any project where its syntax may prove useful.

As discussed in section 2.1.2, many libraries aim to mitigate this by means of class structures and global hoisting. By minimising its reliance on the OOP paradigms and enforcing immutability, the *@grafig/lib* package goes a long way to avoid the use of context-free variables and pass-by-reference[3] functionality. The *@grafig/syntax* package builds upon this, simplifying commonly used structures and syntax by introducing new language features such as infix vector operators and trailing lambda functions.

Vector Literals

Most implementations of a vector object have their components as properties that are accessible with **member dot notation** – i.e. `v.x`, `v.y`, and so on. Given this, the natural way to express this in JavaScript is with **object literals** such as `{ x: 1, y: 2, … }`, for example. In vanilla JavaScript, it could be similarly expressed as an **array literal** such as `[1, 2, …]`, which concisely conveys the same information, but must then use the far less concise **member index notation** to get component values – i.e. `v[0]`, `v[1]`, and so on. Neither option is untenable but, conversely, neither provides the ability to declare vectors in a manner common in other languages.

When using the extended syntax however, a **vector literal** offers equal brevity to the array literal, as well as the common nomenclature and clarity of component accessors that use member dot notation. Using the following syntax, vectors can be easily declared with parentheses, e.g. `(1, 2, 3)`, which is then converted to the equivalent object literal upon transpilation – i.e. `{ x: 1, y: 2, z: 3 }`.

Infix Operators

Further supporting vector manipulation, the extended syntax adds operator overloads for use with vector objects, allowing statements such as `(1, 2) + (3, 4)` in much the same way mentioned in section 2.1.3. These operators adhere to the same rules of precedence as in vanilla JavaScript, allowing parenthesised expressions to be evaluated in the correct order. This is clearly demonstrated in Appendix B, where the final snippet calculates a position vector in a single line combining multiple operators with multiple vector and number operands, greatly reducing the code needed in comparison to the analogous Canvas API example. When transpiled, operators are replaced with functional counterparts exported by the *@grafig/lib* package, acting similarly to *React* with how it handles the use of the JSX syntax within components by ensuring `React` is in scope.

Swizzle Expressions

As discussed in section 2.1.3, swizzling is a very useful means of both rearranging vector components as well as coercing dimensionality. The new syntax includes the ability to use **swizzle expressions** to create new vector objects. Although swizzling is potentially less utilised than the previous two, it greatly simplifies syntax and creates consistency with similar applications of vector functionality.

---

[3] When objects are passed by reference, their value may be altered, whereas when passed by value, the object is copied and remains unchanged. Functional paradigms dictate that arguments should always be immutable, i.e. passed by value, thus reducing potential for side-effects in code.

Trailing Lambdas

Taking direct inspiration from Kotlin, the addition of **trailing lambdas** greatly simplifies the use of closures which, due to the design patterns explained in the previous section, are used extensively by the *@grafig/lib* package. As shown in Appendix B, the final code snippet brings the closure passed to the `Canvas()` function into a code block after the function expression. Additionally, this block is scoped such that each statement acts upon the first argument of the closure which, in this case, is the `Context` object used to render to the canvas. This provides a more fluent means of interacting with the canvas that promotes composability and declarative functionality, without relying on globally scoped instances to achieve the same effect.

3.1.3 Transpiler Binary

To extend the JavaScript syntax, there needed to be a means of transpiling written code into vanilla JavaScript before it is run. For this project, this is done by the *@grafig/syntax* package which contains a Golang project that builds down to a cross-platform binary. By using a compiled binary, transpilation can be done on any device and, as such, can be easily integrated into a development pipeline. By using the *Cobra* library, a comprehensive CLI tool was developed that follows a similar structure to other common tooling such as `git`, utilising subcommands, arguments, and flags as shown in Appendix C. This allows the binary to encompass parsing, transpilation, and complexity analysis within a single, familiar command line interface.

The *@grafig/syntax* package uses the *ANTLR parser generator* tool to parse code in either the extended syntax or vanilla JavaScript. The tool creates a **lexer** that splits code into **tokens**, and a **parser** that builds a **parse tree** from those tokens, as shown in Appendix D. This parse tree can then be traversed to either transpile or analyse the given code with a parse tree **listener**. To determine how the lexer tokenises input and how the parser builds the parse tree, a set of rules, called a **grammar**, is used to define the syntax able to be parsed. For the extended syntax, a new grammar was developed that implements all the rules defined by the ANTLR grammar for JavaScript, augmenting them to support the additional syntax. Finally, once able to parse the new syntax, a parse tree listener was developed to allow transpilation into vanilla JavaScript.

3.1.4 Complexity Reporter

Building upon the parser developed for use in the transpiler, the complexity reporter uses a parse tree listener to collate complexity data from the parse tree. By implementing the metrics described in section 2.2, the tool calculates the complexity of each function and aggregates the nested complexity of any parent function scopes. To create efficient and repeatable reporting, the complexity reporter can accept glob patterns, dynamically determine file language, and output data in both JSON and CSV file formats.

This tool was used extensively during comparative testing of the developed framework and provides the sole data source for complexity analysis. As such, it was important to ensure that the reporting was accurate and, to do so, the data output was compared to that of a different, established JavaScript complexity tool called *complexity-report*. By evaluating the difference between the two reporters for multiple, large source code files from other well-known projects, the resultant error rate was determined to be below 4%

per source line of code. The testing of the developed complexity reporter is explained further in section 3.2.2 but, given that it was used in isolation to examine different pieces of code, this error should minimally impact comparisons between generated reports.

3.1.5 Example Application

As a means of demonstrating the usage of the framework, a web application was made to exhibit the features introduced. Shown in Figure 3.3 and Figure 3.4, the demo contains two subpages. The first renders the *Ulam Spiral*, wherein ascending integers are displayed as a square spiral and the primes are highlighted, while the second generates a chess board from a query parameter given in *Forsythe-Edwards Notation* (FEN).

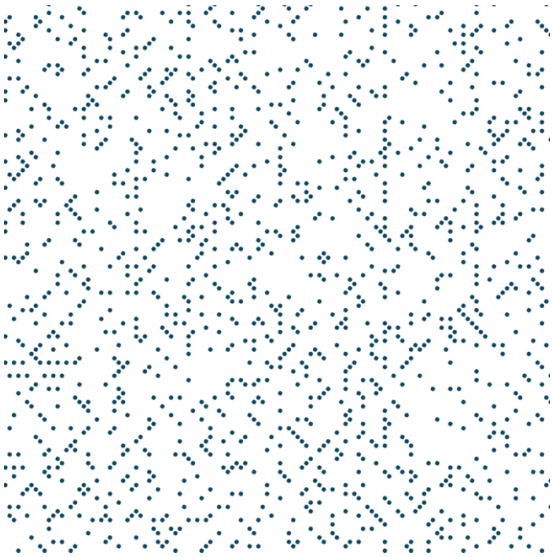


Figure 3.3 – Ulam spiral graphic generated with the example application.

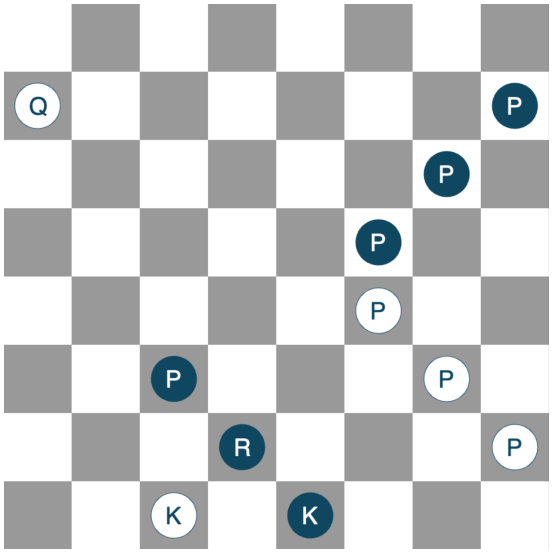


Figure 3.4 – Chess board graphic generated with the example application.

FEN string: 8/Q6p/6p1/5p2/5P2/2p3P1/3r3P/2K1k3

The chess board provides a good example of handling interaction from external sources, as it will redraw whenever a new FEN string is used. Although the Ulam Spiral demo is a relatively simple example, the source code shown in Appendix E demonstrates its effective use of the both the library and syntax, as well as how the framework can support modular, composable application design.

All non-HTML application code was written in the extended syntax, and the application could be easily built using an open-source build tool called `parcel`. This was done by running the `yarn start` command, which is defined within the `package.json`. This then begins a development server wherein any changes to source code are detected, prompting the application to be rebuilt. As `parcel` is unable to interpret the extended syntax, the *@grafig/syntax* package exports a `Transformer`, allowing the configuration of asset management within `parcel` to integrate transpilation as shown in Appendix F. This greatly sped up the development of the example application by reducing the interaction needed to view code changes as they were put into effect upon saving the file. In addition, the *@grafig/lib* package includes generated source mappings for transpiled typescript code, providing the means for more meaningful IntelliSense and auto-completion within the IDE, as well as efficient testing and debugging. This lends credence as to the qualitative suitability of the developed framework as an alternative to the Canvas API and other graphics rendering libraries.

3.2 Methodology and Lifecycle

Development was predominantly planned with the *Kanban* methodology with emphasis on iterability. This was done to allow a vast backlog of tasks to be accumulated that could be refined and collated into iterations that each wholly complete a feature and form the basis for a versioned release of the project. By keeping a record of planned, in progress, and completed work, the project was able to remain structured and avoid *feature creep* by maintaining well defined bounds within which each task should be completed. During the initial stages of development, major design decisions were documented along with elucidation of forces motivating the decision and evaluation of potential solutions.

3.2.1 Source Control

The project employs *Git* to manage source control and, as there was only a single developer, the branching strategy was kept simplistic and uses a `dev` branch that is merged into `main` once the feature was completed. This is exemplified in the git graph shown in Figure 3.5, where the changes made on the `dev` branch are made up of several commits that are then squashed into a single merge commit on the `main` branch and given a tag indicating the semantic version number.

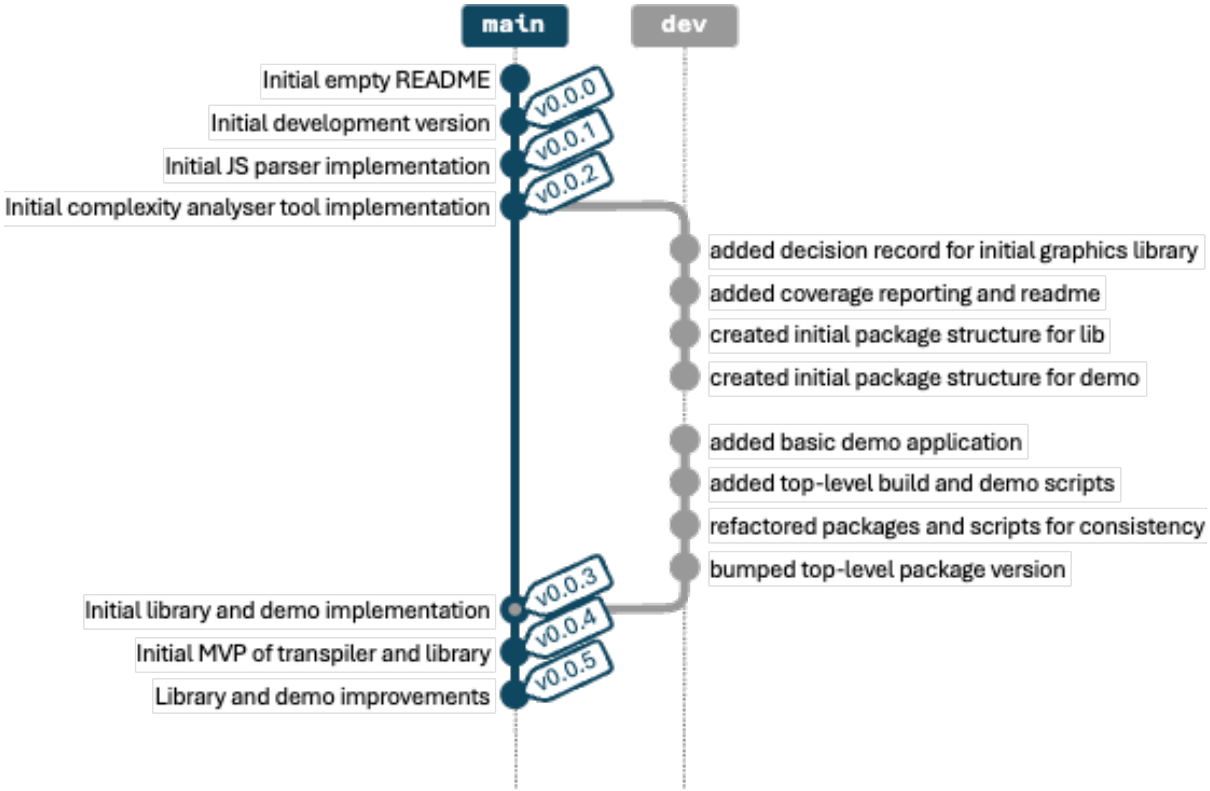


Figure 3.5 – Git graph of development branching strategy.

3.2.2 Testing

Both the *@grafig/lib* and *@grafig/syntax* packages have automated testing suites that are integrated into the npm package scripts such that they can be easily run as part of the development pipeline. This allowed for rapid testing during development of new features as well as providing regression testing to mitigate impacts upon existing functionality.

To test the complexity reporter, as mentioned earlier, the output was compared to that of an existing solution. From the difference in reported their values, an average rate of error was calculated for each metric per source line of code. As shown in Figure 3.6, the error percentages for LLOC, parameters, and cyclomatic complexity are all below 2% with the Halstead metrics coming to around 10%. Given that nearly over 6000 lines of code were analysed, this result seems respectable and, although notably higher than the rest, the definition of Halstead operators and operands is inconsistent between solutions at best, so a higher error rate is to be somewhat expected. However, as previously stated, its consistency is more crucial than its accuracy and, considering there no errors were reported for SLOC, it can be inferred to be reliably consistent for its intended use.

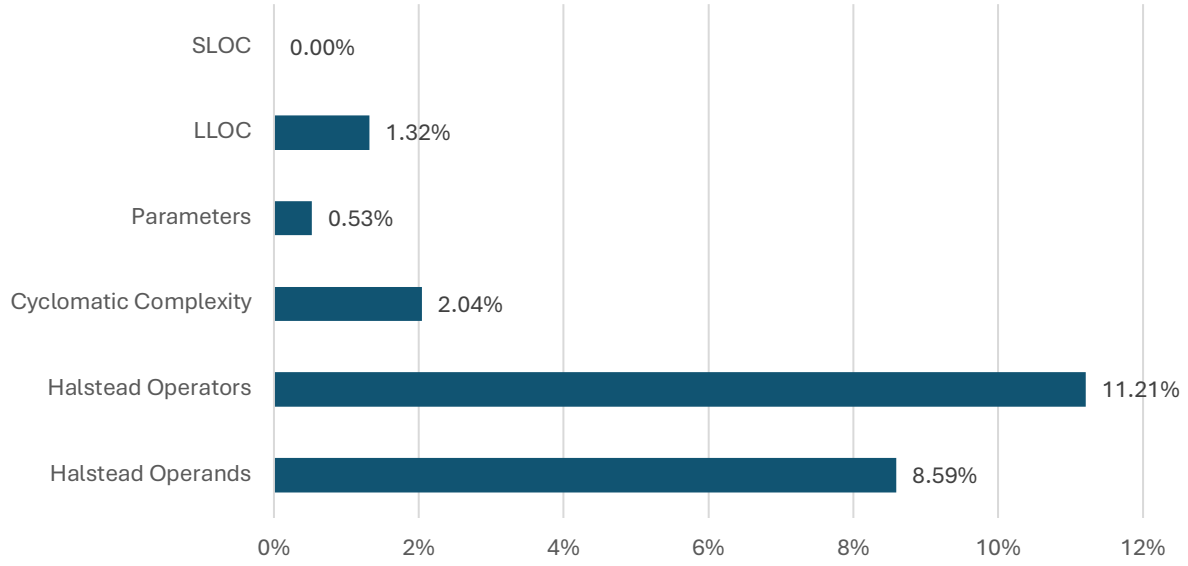


Figure 3.6 – Average complexity reporter error rate (%) per SLOC.

Testing of the *@grafig/lib* package was done using the `jest` framework with 100% of all functions being unit tested, covering 80% of logical statements, as shown in Figure 3.7. Both testing and coverage, alongside linting, enabled a more consistent approach to the development of the library, providing static analysis tools that create a reusable pipeline for managing releases. These tools are easily run with the `yarn prepare` command and could easily be integrated into an even wider pipeline, using *commit hooks* or *merge rules* to further enforce development standards.

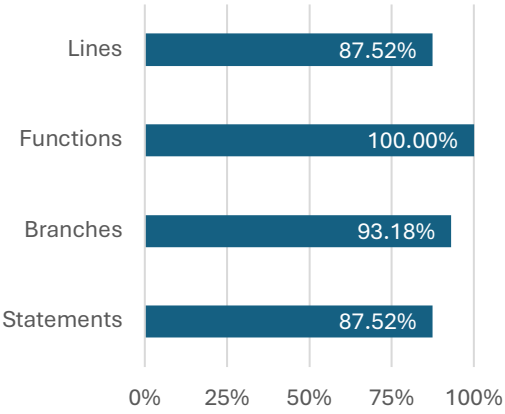


Figure 3.7 – Coverage report totals for @grafig/lib.

3.2.3 Documentation

Documentation forms a key part of the development cycle and uses common conventions such as each package having a README and its own `docs` folder. Additionally, the project architecture and the extended syntax are both thoroughly documented in their own files within the top-level project's `docs` folder.

As mentioned in section 3.1.3, Appendix C shows that the *@grafig/syntax* binary offers standardised usage information via the `--help` flag, with the *@grafig/lib* package doing the same for all available package scripts. Beyond this, the library also extensively uses *JSDoc* for documentation, generating a web-based reference with the `typedoc` tool, as shown in Figure 3.8. The generation of this reference was also included in the development pipeline to ensure it aligns with any released code prior to being published for use.

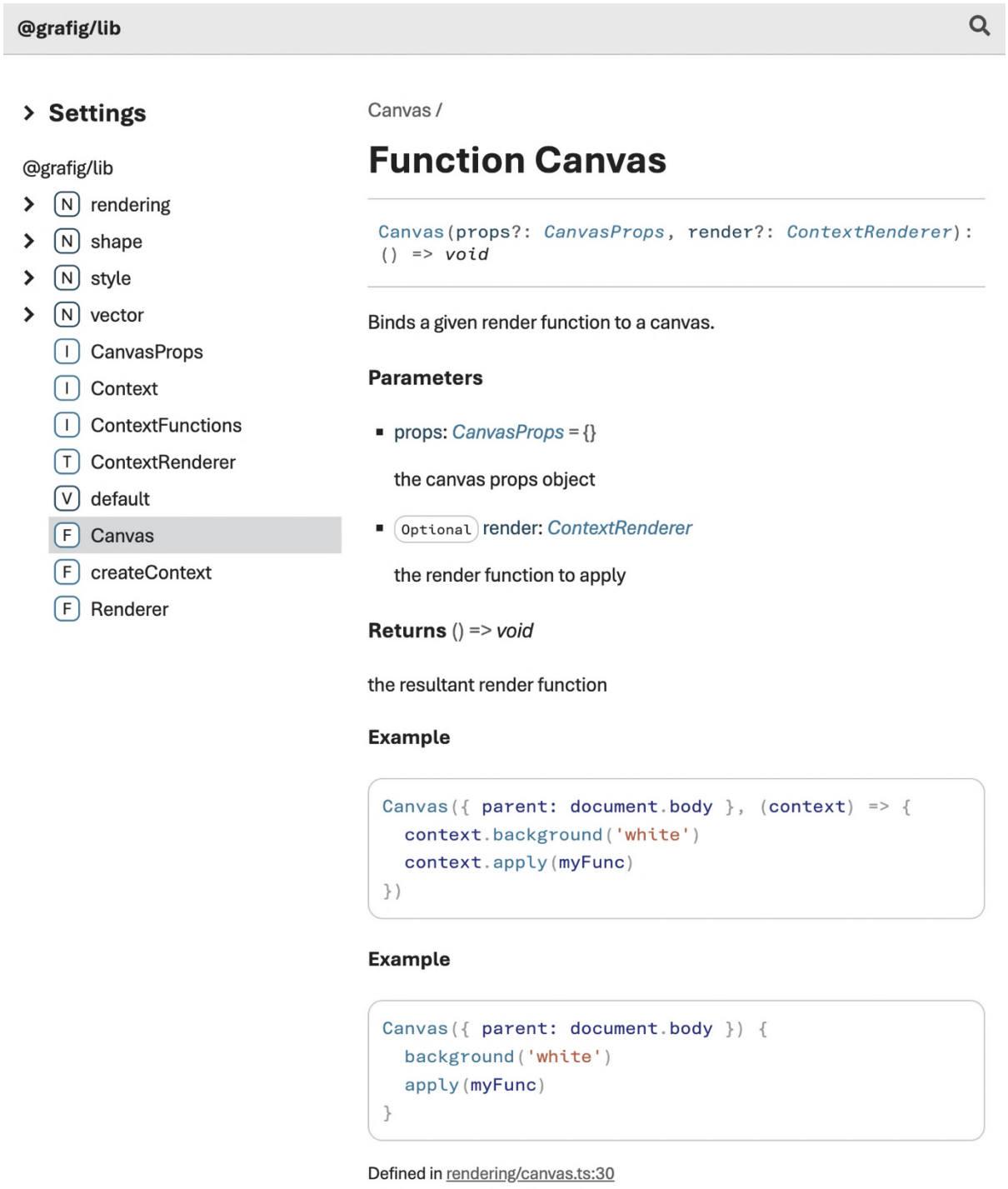


Figure 3.8 – Generated library reference application.

4 Results

The testing of the resultant framework predominantly relies on the comparative analysis of common complexity metrics against that of the Canvas API and *p5.js*, a popular alternative library. This section aims to illustrate whether their usage can influence source code complexity, and if so, to what extent. By comparing the results for each framework, it can be determined whether the developed system is a suitable solution for graphics rendering and whether it improves upon the existing available solutions.

4.1 Data Collection

As previously mentioned, the complexity reporter implemented in the *@grafig/syntax* package, once validated to be suitable, was used to evaluate the complexity of the three libraries. To provide test cases, a file containing nine functions of varying complexity was written for each solution, with separate test cases for the developed library both with and without the extended syntax. An example of this can be seen in Table 4.1 wherein the values reported for LLOC are shown for each test case and file. The tests were each then analysed by the complexity reporter, aggregating the data into a single CSV file able to be imported into a spreadsheet for further analysis.

Table 4.1 – LLOC values reported for each test function.

	Test Function	Canvas API	p5.js	grafig	grafig+figscript
A	createCanvasElement_withBodyParent_withDifferingDimensions	7	1	1	1
B	createCanvasElement_withBodyParent_withMatchingDimensions	7	2	1	1
C	createCanvasElement_withElementParent_withMatchingDimensions	10	5	1	1
D	createCanvasElement_withoutParent_withDefaultDimensions	4	1	1	1
E	createCanvasElement_withoutParent_withNonDefaultDimensions	6	1	1	1
F	renderCanvasElement_withBackground	8	1	1	1
G	renderCanvasElement_withBackground_withOutlinedShapes	24	1	1	1
H	renderCanvasElement_withDynamicTasklist	9	3	3	3
I	renderCanvasElement_withRandomCircleWithinBounds	12	2	3	3

4.2 Data Transformation

Given that the data was evaluated, collated, and aggregated by the complexity reporter, there was minimal need for data sanitisation or formatting; the only calculations done further aggregated averages of pivoted data. As will be discussed in the following section, the only values derived at this point are averages and the maintainability index, as shown in Equation 2.9, which provides a consistent means to compare the programs across several different metrics.

4.3 Metric Comparison

Looking at the raw metrics gathered for each solution, it is evident that the developed framework consistently outperforms the Canvas API in nearly every metric for the given test cases. As shown in Figure 4.1, both SLOC and LLOC were greatly reduced in comparison to the Canvas API as well as *p5.js*. This trend continues in Figure 4.2 where the Halstead operators and operands both see lower values for the developed library, with or without using the extended syntax.

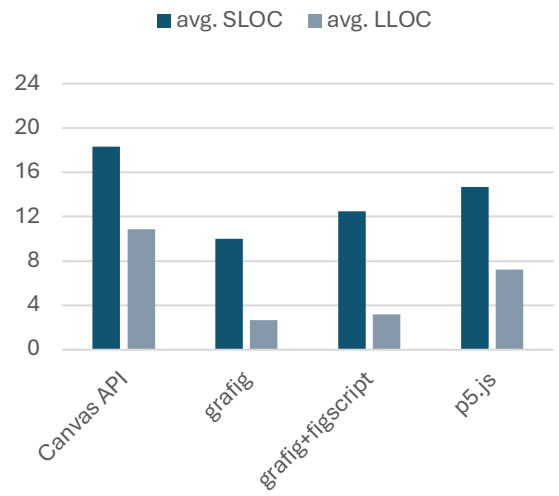


Figure 4.1 – Average SLOC and LLOC per test file.

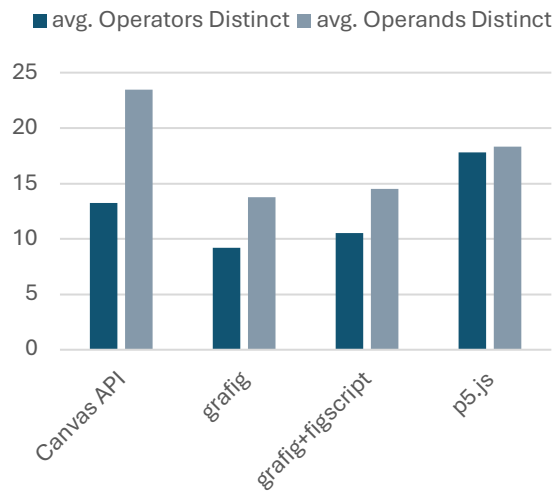


Figure 4.2 – Average distinct Halstead operators and operands per test file.

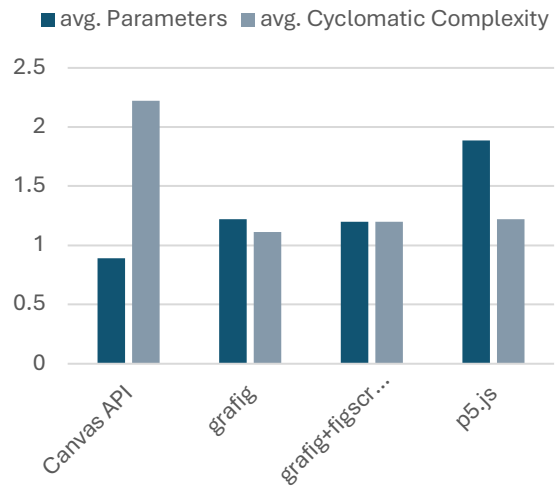


Figure 4.3 – Average parameter count and cyclomatic complexity per test file.

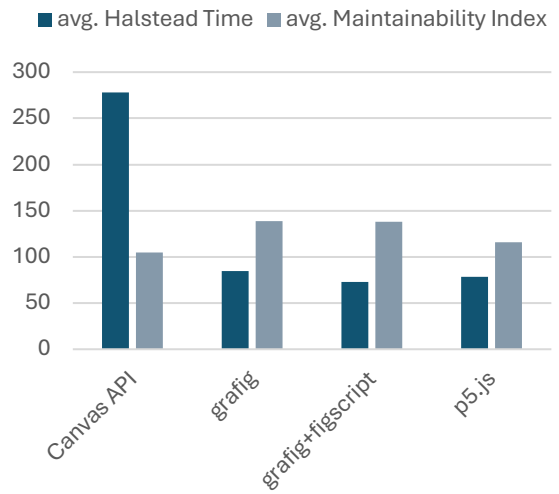


Figure 4.4 – Average Halstead time and Maintainability Index per test file.

The data in Figure 4.3 differs slightly, as the parameters increase with the use of the new library. This, however, is likely due to the limited number of test cases, as is also true of cyclomatic complexity which was consistently halved due to the relatively low complexity of each test case. With a larger test sample, these values may have yielded more reliable results but, as they are, they may not best represent reduction in complexity for larger volumes of code.

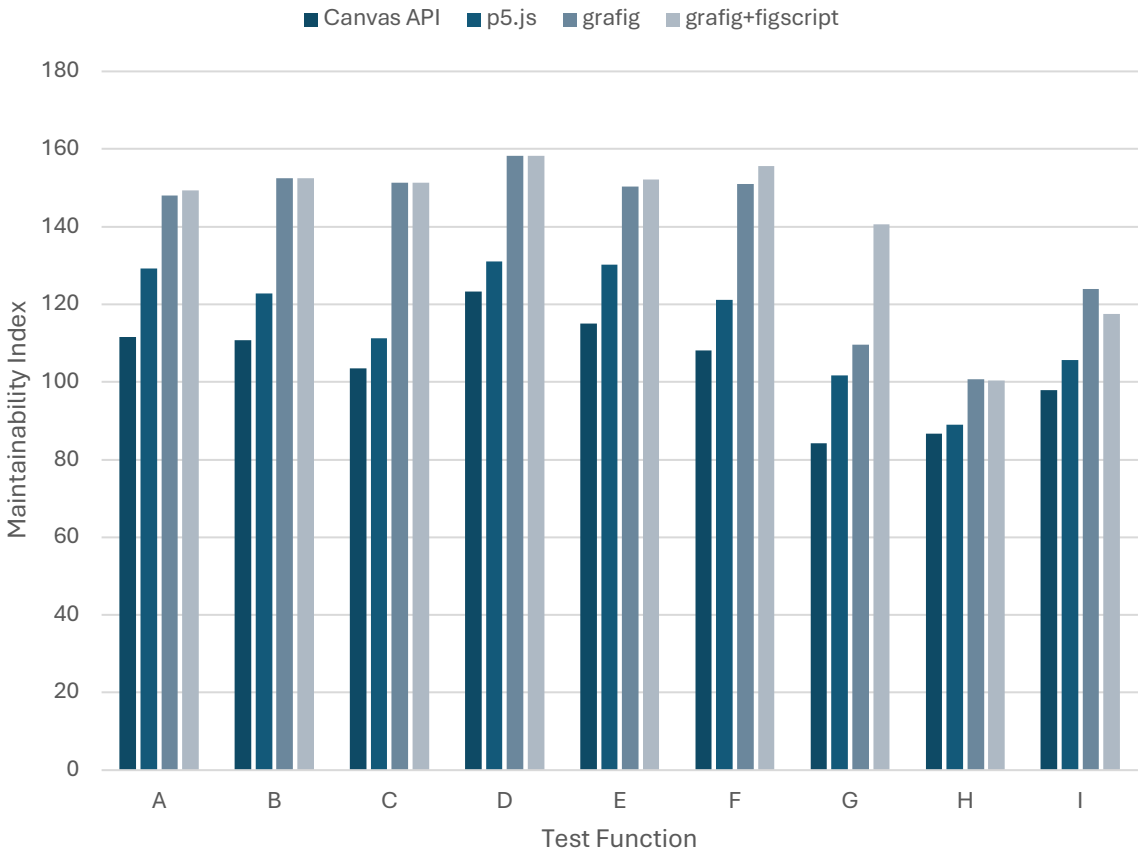


Figure 4.5 – Average Maintainability Index per test function.

Looking at the derived metrics in Figure 4.4, a significant reduction can be seen in the Halstead time, calculated using Equation 2.7, for the developed system. This is also seen for *p5.js* however, the greater maintainability index shows that the grafig options both improve upon *p5.js* as well as the Canvas API. This improvement is reiterated in Figure 4.5, whereby further investigating the MI, this time for each test function, a consistent trend of higher maintainability can be seen across each test case.

Finally, by comparing each solution against the Canvas API, the percentage increase for MI could be calculated, as is shown in Figure 4.6. This further exemplifies that the system developed, even with its small scope, makes keen improvements upon both the existing Canvas API and does so more than *p5.js* for the analysed test cases. In the following section, these improvements will be discussed further, seeing how these findings address the core motivation of this project and how effectively the developed framework fulfils the initial outlined proposal.

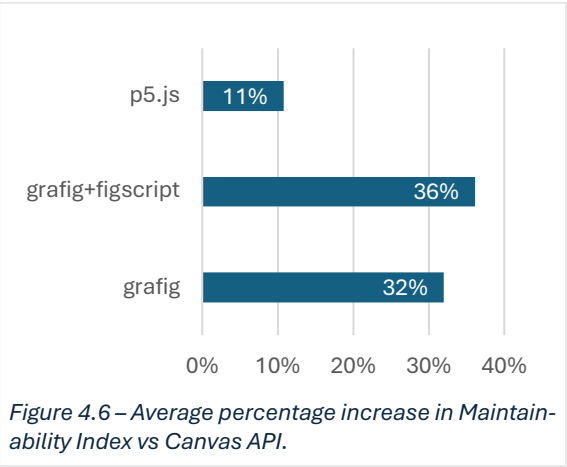


Figure 4.6 – Average percentage increase in Maintainability Index vs Canvas API.

5 Discussion

This project aimed to investigate the complexity of code inherent in web-based vector graphics applications with the view to create a framework able to provide a library of functionality and domain-specific syntax that support their development. Using the data gathered in the previous section, this section will critically assess the extent to which the developed system achieves this claim, before going on to explore how it could be further expanded with future iterations.

5.1 Comparative Analysis

As was shown by the results collected from the complexity reporter, both the library and extended syntax greatly reduced the overall complexity of tested code by over 30% when compared to the Canvas API, making a further 20% improvement over the *p5.js* library. This is a considerable improvement and indicates that code complexity could be further reduced with subsequent additions to the developed framework. This trend is unlikely to continue in a linear fashion, however, and it is important to note that the test cases used for analysis were curated to exemplify the novel features introduced. As touched upon in section 4.3, this implies that the improvements seen for the developed system should be seen as best-case scenarios and would, naturally, not be reflective of projects where the new functionality is seldom utilised.

In terms of development of vector graphics, the research discussed in section 2.2 shows that complexity analysis remains a crucial tool in modern development and can provide essential insight into system maintainability. Given this, it can be assumed that larger projects containing vector graphics may well benefit from such a framework to simplify development and improve productivity. As mentioned in section 3.1.1, the library can be used as a standalone asset with minimal impact to the overall complexity compared to the extended syntax. This provides a simpler migration path from existing libraries, which is supported by the common design patterns used throughout, allowing projects to then introduce the extended syntax when, and where, it proves beneficial.

5.2 System Appraisal

Looking back at the deliverables outlined in section 1.3, the project aimed to develop a vector graphics library, a transpiler and complexity reporter for the extended syntax, as well as an example application using the developed framework. Not only does the final system contain these individual components, but also effectively documents the design and usage of each, as well as incorporating development pipelines to run comprehensive test suites and other static analysis tooling. The components were each able to seamlessly integrate into the wider project, following common design patterns for their implementations and usage that allowed interactive debugging, in-editor linting, and other productivity tools through configuration of the IDE. Alongside the development pipelines mentioned above, this greatly improved the development experience throughout the project, specifically that of the example application.

To maintain a reasonably bounded scope for the project and support an iterative style of development, features were introduced where needed before expanding their functionality, as mentioned in section 3.2. This led the developed components, specifically the library and syntax, to be less comprehensive than their equivalent counterparts, offering only a fraction of their capability. However, as discussed above, the comparative analysis shows a marked quantitative improvement in complexity and how the developed framework, alongside its use of established design and architecture standards, provides a solid foundation for its future expansion.

5.3 Potential Improvements

To attempt to match the proficiency of existing graphics libraries, the *@grafig/lib* package could begin introducing further functionality to mirror that of other solutions. Features such as hierarchical rendering and image manipulation are both two features that would greatly improve the framework's utility. The library could implement a matrix module that builds upon the vector module, which could itself be extended with many more methods such as `dot()`, `cross()`, `mag()`, etc. Building on these future iterations of the library, the *@grafig/syntax* package could then introduce further extensions to vanilla JavaScript with common syntax such as named arguments, matrix literals and operator overloads, as well as fully implementing Kotlin's trailing lambda syntax.

Beyond additions to the new syntax itself, future versions could easily adapt it to allow transpilation to TypeScript, given that it also extends vanilla JavaScript. This could then introduce typing into the extended syntax which could further improve the development of vector graphics. Furthermore, as *React* is a powerful and widely used tool for web app development, extending both JSX and TSX could combine the benefits of the developed framework with those from React for a greatly simplified development experience when compared to using the native Canvas API in isolation.

5.4 Reflective Analysis

Looking back at the personal objectives in section 1.4, the project has greatly improved the appreciation and understanding of how code complexity can impact both the initial development, and future maintenance, of software. Having explored the concept of code complexity at length and applied it to a domain-specific framework, this project broadened awareness of the role that syntax plays in a programming language. Additionally, it has furthered understanding of how transpilation and compilation work, as well as the benefits it can bring to application development.

The development itself progressed smoothly, and all initial objectives were achieved to a suitable extent. By using the Kanban methodology, grouping tasks into fully complete iterations, the potential scope of the project was able to grow naturally whilst maintaining a controlled cadence of feature development. In doing this, along with the employed branching strategy discussed in section 3.2.1, the project increased understanding of project planning and development. This was further supported by the extensive use of tooling, such as code linters, testing frameworks, and documentation generators, which all significantly improved productivity and consistency during development. Moreover, incorporating these tools into package scripts, yarn workspaces, and IDE configuration greatly enhanced understanding of system design and architecture patterns as whole.

6 Conclusion

This project has thoroughly explored the importance of code complexity as well as how effective syntax and design can improve the maintainability of code. By applying this knowledge and augmenting the vanilla JavaScript syntax with domain-specific features, a cohesive framework was successfully developed. As seen in the results gathered from the complexity analysis, this framework made a promising start toward enabling low complexity development of vector graphics in web applications.

While only a personal testament, the benefits of the developed system were confirmed during the development of the example application, such that the previously mentioned chess board demo was able to be developed far quicker than anticipated. This indicates that the framework can positively impact development experience, and with additional research, could be further extended to broaden its utility within graphics development.

If this project were to be continued or undertaken anew, with more resources and time, the developed system provides an effective foundation of understanding for a far more comprehensive framework that rivals the comparable alternatives. This would also be supported by wider research into domain-specific syntaxes used for graphics rendering, exploring how their usage impacts code complexity and maintainability, and potentially expanding beyond 2D vector graphics and facilitating development of 3D graphics that leverage shaders. Integrating this framework with other established web-development frameworks, such as React, could provide improved canvas manipulation that leverages the exiting benefits of those frameworks. With all of this, a framework that expands upon the one developed, could greatly improve development and maintenance of websites and applications that include bespoke or complicated graphics.

In review of the initial project contract, the developed framework and its subsequent analysis achieves each aim and objective, while incorporating an iterative and flexible development methodology that leverages modular and productivity conscious design patterns. The project allowed for an expanding scope, continuously recording potential improvements, while maintaining a well-structured development cycle that encouraged validation of existing features before the introduction of new functionality. From this, an encouraging insight was gained into the development of an effective yet approachable graphics rendering framework for client-side web applications.

7 References

ABRAMOV, D. (2019) *React v16.8: The One With Hooks – React Blog*. [Online] Available from: https://legacy.reactjs.org/blog/2019/02/06/react-v16.8.0.html [Accessed Apr 4, 2025].

ABRAMOV, D., ALPERT, S. and FLORENCE, R. (2018) React Today and Tomorrow and 90% Cleaner React With Hooks. In: *React Conf 2018, Henderson, NV, Oct 25-26, 2018*.

BASILI, V.R. and PERRICONE, B.T. (1983) Software errors and complexity: An empirical investigation. *NASA Goddard Space Flight Center Collected Software Engineering Papers,* 2.

BIERSDORFER, J.D. (ed.) (2017) When Dinosaurs Roam in Chrome. *The New York Times,* Nov 14 retrieved from: https://www.nytimes.com/2017/11/14/technology/personaltech/chrome-dinosaur-internet-connection.html.

CORNELL UNIVERSITY (1998) *What is Computer Graphics?*. [Online] Available from: https://www.graphics.cornell.edu/online/tutorial/ [Accessed Apr 3, 2025].

D. COLEMAN et al. (1994) Using metrics to evaluate software system maintainability. *Computer,* 27 (8), pp. 44–49.

DEMARCO, T. (1982) *Controlling software projects : management, measurement & estimation*. Englewood Cliffs, NJ: Yourdon Press.

FELICIAN, L. and ZALATEU, G. (1989) Validating Halstead's theory for Pascal programs. *IEEE Transactions on Software Engineering,* 15 (12), pp. 1630–1632.

FENTON, N.E. and NIEL, M. (1999) A Critique of Software Defect Prediction Models. *IEEE Transactions on Software Engineering,* 25 (3).

FLATER, D. (2018) *'Software Science' revisited: rationalizing Halstead's system using dimensionless units*.

HAO, G. et al. (2023) On the accuracy of code complexity metrics: A neuroscience-based guideline for improvement. *Frontiers in Neuroscience,* 16.

HATTON, L. (1993) The automation of software process and product quality. *Transactions on Information and Communications Technologies,* 4.

HUMPHREY, W.S. (1997) *Introduction to the personal software process*. Reading, Mass.: Addison-Wesley Publishing.

MCCABE, T.J. (1976) A Complexity Measure. *IEEE Transactions on Software Engineering,* SE-2 (4), pp. 308–320.

MDN (2025) *Canvas API*. [Online] Available from: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API [Accessed Apr 4, 2025].

P5.JS (2025) *About*. [Online] Available from: https://p5js.org/about/ [Accessed Apr 3, 2025].

SUTHERLAND, I.E., RODDEN, K. and KUHN, M. (2003) *Sketchpad: A man-machine graphical communication system*.

T. HARIPRASAD et al. (2017) Software complexity analysis using halstead metrics. In: *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, pp. 1109–1113.

TAGPRO LEAGUE (2025) *TagPro League*. [Online] Available from: https://www.tagpro-league.com/ [Accessed Apr 3, 2025].

V. Y. SHEN et al. (1985) Identifying Error-Prone Software—An Empirical Study. *IEEE Transactions on Software Engineering,* SE-11 (4), pp. 317–324.

W3TECHS (2025a) *Usage Statistics and Market Share of HTML5 for Websites*. [Online] Available from: https://w3techs.com/technologies/details/ml-html5 [Accessed Apr 3, 2025].

W3TECHS (2025b) *Web Technologies Statistics and Trends*. [Online] Available from: https://w3techs.com/technologies [Accessed Apr 3, 2025].

WIKIPEDIA (2023) *Template:Should be SVG*. [Online] Available from: https://en.wikipedia.org/w/index.php?title=Template:Should_be_SVG&oldid=1190379749 [Accessed Apr 3, 2025].

8 Appendices

8.1 Appendix A – Implementation and usage of `vector.from()`

```
// from.ts
function from<
  C extends string,
  T extends Record<C, number> = Record<C, number>
>(obj: T, x: C, y: C): Vec2

function from<
  C extends string,
  T extends Record<C, number> = Record<C, number>
>(obj: T, x: C, y: C, z: C): Vec3

/**
 * Creates a vector from a given object and property names
 *
 * @param obj the target object
 * @param x the property key for the x-component
 * @param y the property key for the y-component
 * @param z the property key for the z-component
 * @returns the resultant vector
 */
function from<
  C extends string,
  T extends Record<C, number> = Record<C, number>
>(obj: T, x: C, y: C, z?: C): Vec {
  return {
    x: obj[x],
    y: obj[y],
    z: z != undefined ? obj[z] : z,
  }
}

// app.ts
const box = { width: 100, height: 200, outline: true }

const a = from(box, 'width', 'width')   // ok
const b = from(box, 'width', 'outline') // error ts(2345)

Argument of type '{ width: number; height: number; outline:
boolean; }' is not assignable to parameter of type 'Record<
"width" | "outline", number>'.
```

8.2 Appendix B – Code to render a circle within canvas bounds

Canvas API

```
// create canvas and context
const canvas = document.createElement('canvas')
const context = canvas.getContext('2d')
if (context == null)
  throw new Error('cannot use canvas with null context')
// get random position within bounds
const r = 10
const x = radius + Math.random() * (canvas.width - r * 2)
const y = radius + Math.random() * (canvas.height - r * 2)
// render circle
context.fillStyle = 'rgb(255, 255, 255)'
context.beginPath()
context.ellipse(x, y, r, r, 0, 0, Math.PI * 2)
context.fill()
context.closePath()
```

grafig

```
// get random position
const r = 10
const v = { x: Math.random(), y: Math.random() }
// render circle to canvas within bounds
Canvas({}, context => {
  context.circle({
    position: add(mult(sub(context.dimensions, r * 2), v), r),
    r,
    fill: 'rgb(255, 255, 255)',
  })
})
```

grafig+figscript

```
// get random position
const r = 10
const v = (Math.random(), Math.random())
// render circle to canvas within bounds
Canvas {
  circle({
    position: (it.dimensions - r * 2) * v + r,
    r,
    fill: 'rgb(255, 255, 255)',
  })
}
```

8.3 Appendix C – Help text for *figsyntax* binary

```
$ figsyntax --help
Use a subcommand with --help to see more usage information.

Usage:
  figsyntax [command]

Available Commands:
  analyse     Analyse FigScript or JavaScript file complexity.
  completion  Generate autocompletion script for the specified shell
  help        Help about any command
  parse       Parse a given figscript or JavaScript file.
  transpile   Transpile a given figscript file.

Flags:
  -h, --help   help for figsyntax

Use "figsyntax [command] --help" for more information about a command.
```

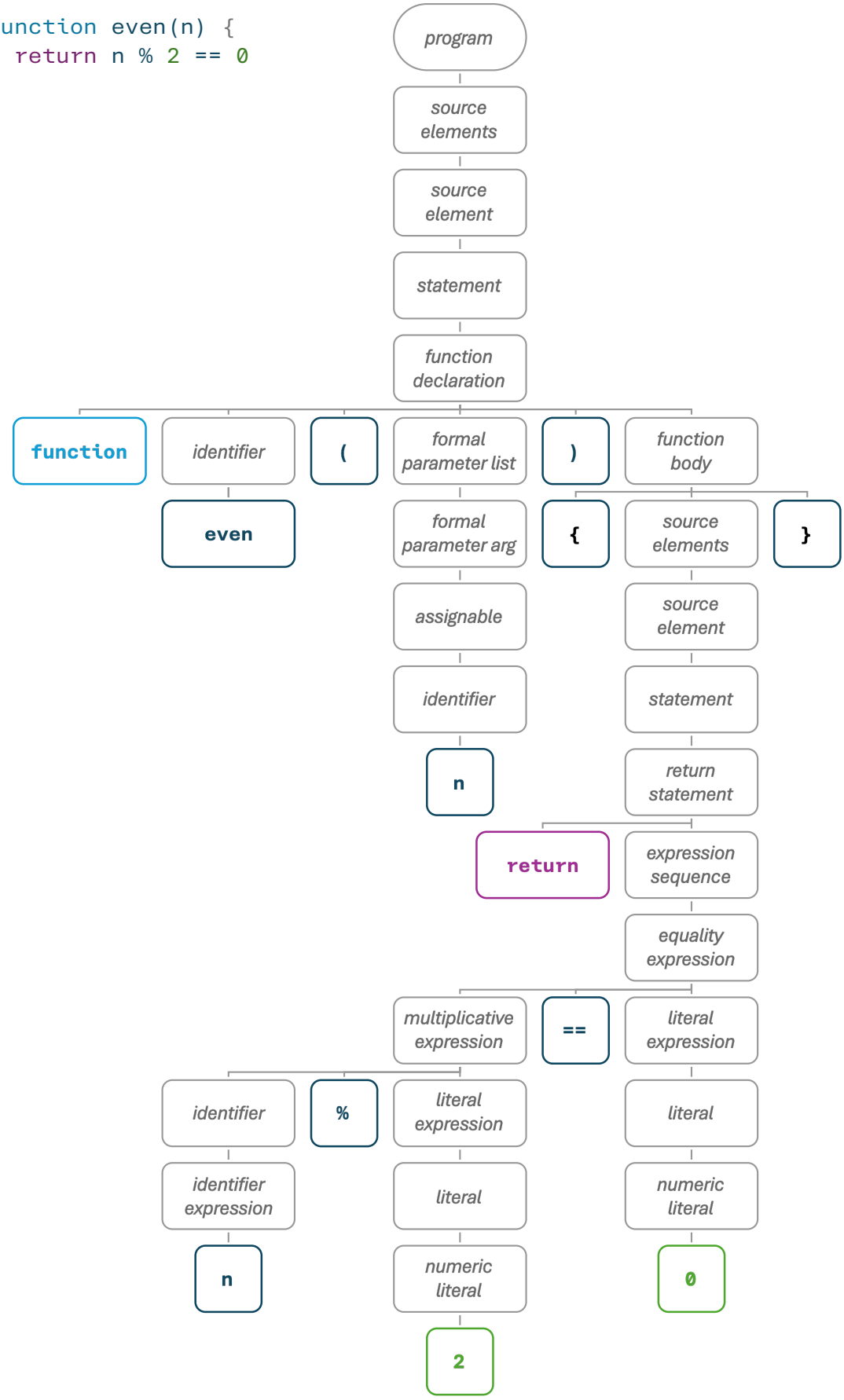
```
$ figsyntax transpile --help
Given a file containing valid figscript, the transpile command will
parse the code and generate the transpiled JavaScript.


Usage:
  figsyntax transpile <path|string> [flags]

Flags:
  -h, --help               help for transpile
      --debug              use the DEBUG logger level
      --error              use the ERROR logger level
      --info               use the INFO logger level
      --log-level string   specify a logger severity level
  -o, --output    string   specify an output dir
  -s, --silent             don't write any output to the console
      --warn               use the WARN logger level
```

8.4 Appendix D – Example *JavaScript* function and *ANTLR* parse tree

```
function even(n) {
  return n % 2 == 0
}
```



8.5 Appendix E – Example application source code

Bold and *italicised* code indicates use of the library and extended syntax respectively.

```
// ulam.fs
function Spiral(size, radius) {
  const w = Math.floor(Math.max(size.x, size.y) / radius)
  const length = w ** 2
  const offset = w % 2 === 0 ? Math.floor(w / 2) : 0
  const center = Math.floor(length / 2) + offset

  const getSpiralIndices = (acc, _, n) => n === 0
    ? [[center, 0, 1, 1, 0]]
    : [...acc, nextIndex(width, ...acc[acc.length-1])]

  const toRenderFunction = ([i], n) => {
    const position = (i % w, Math.floor(i / w)) * (radius * 2)
    if (
      position.x < 0 ||
      position.x > size.x ||
      position.y < 0 ||
      position.y > size.y ||
      !isPrime(n + 1)
    ) return () => {}

    return Renderer {
      circle({ position, radius, fill: '#0F4761' })
    }
  }

  return (context) => Array(length)
    .fill(0)
    .reduce(getSpiralIndices, [])
    .map(toRenderFunction)
    .map(context.apply)
}

// app.fs
Canvas({ parent: document.body }) {
  background('white')
  apply(Spiral(it.dimensions, 4))
}()
```

8.6 Appendix F – Asset transformation and dependency resolution

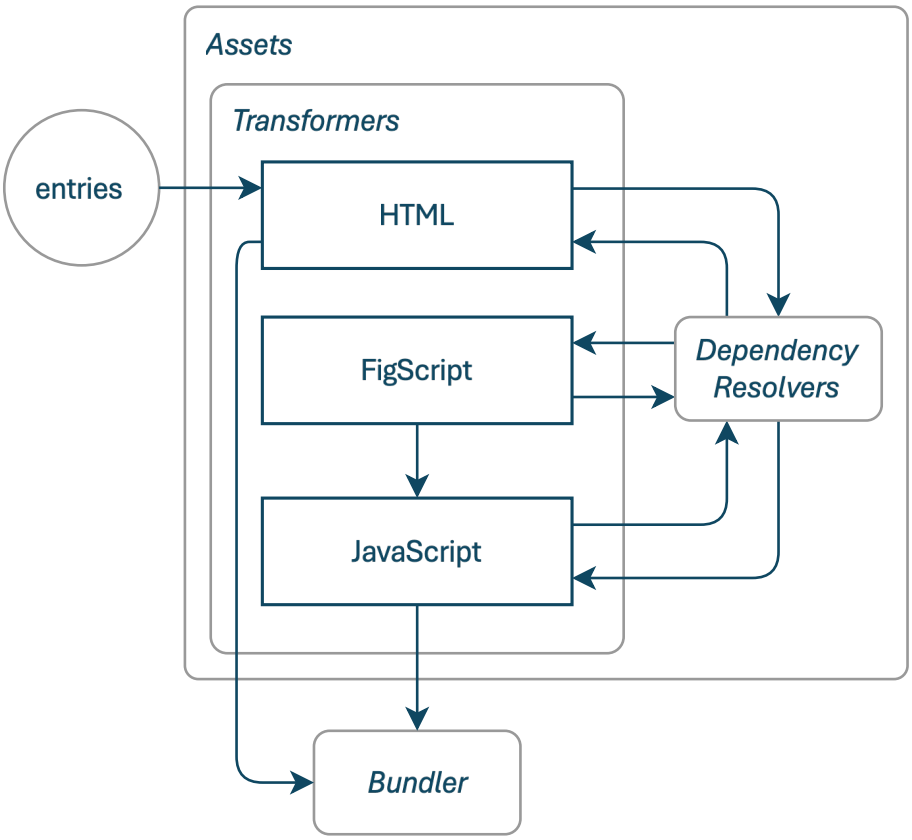
```
// @grafig/syntax/util/transformer.js
const cmd = '../syntax/bin/figsyntax transpile -s'

export default new Transformer({
  async transform({ asset }) {
    const source = await asset.getCode()
    const { stdout, stderr } = await exec(`${cmd} "${source}"`)

    if (stderr.length) {
      asset.diagnostics = parseError(asset.filePath, stderr)
      return []
    }

    asset.setCode(stdout)
    asset.type = 'js'

    return [asset]
  }
})
```



This diagram is a simplified version of the example given by the Parcel reference. Available here: https://parceljs.org/plugin-system/overview/