A Graphical Playground for Haskell

Oliver Jones

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2025

Abstract

Functional programming is a fundamental paradigm in computer science education, yet it often presents a steep learning curve for students unfamiliar with its concepts. This project introduces *A Graphical Playground for Haskell*, an interactive online environment, facilitating the learning and exploration of Haskell through a custom-built graphics library, available at https://haskell-playground.co.uk. The platform allows users to write, execute and share Haskell programs, to generate images and animations which can be viewed directly in the browser.

This dissertation details the design and implementation of both the graphics library and the website. The system is structured to securely execute Haskell code on a server, and stream the results back to the client. Key challenges addressed include ensuring safe code execution and designing an intuitive user interface.

User testing was conducted to evaluate the usability and effectiveness of the tool. The results indicate that the platform can successfully facilitate the learning of Haskell, with users reporting a positive experience using the system. The project demonstrates the potential of interactive environments in enhancing the learning of functional programming languages.

Research Ethics Approval

This project obtained approval from the Informatics Research Ethics committee.
Ethics application number: 8978
Date when approval was obtained: 2025-01-08
The participants' information sheet and a consent form are included in Appendix A.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Oliver Jones*)

Acknowledgements

I would like to extend my most sincere thanks to my supervisor, Prof. Philip Wadler, for his unwavering guidance and support throughout the project. I would also like to extend my gratitude to all those who tested the software and provided feedback, particularly those who completed the user testing survey.

Table of Contents

Chapter 1

Introduction

A Graphical Playground for Haskell is a web-based Haskell editor with a built-in graphics library, allowing users to create images and animations using Haskell and render them directly in the browser. The editor provides a simple, interactive interface for writing Haskell code and viewing both textual and graphical output, without need to install Haskell on the user's computer — a particular pain point for many first-year students in Edinburgh.

The project has been successfully implemented, providing an interactive environment for functional programming and creating graphics. A custom graphics library has been developed, providing a simple interface for creating images and animations using Haskell. This is accompanied by a website that serves as an editor and rendering engine for the graphics library, allowing users to write Haskell code and view the output directly in the browser. It was well-received in user testing, with positive feedback on its usability and functionality and an average rating of 4 out of 5.

1.1 Aims and Motivations

There were a variety of motivations for creating this project, including educational benefits for students learning Haskell, as well as practical applications for users interested in creating graphics and animations.

The project aims to provide an experience that is both beginner-friendly and engaging for users of all skill levels. For beginners, the editor offers a visual, interactive environment for learning Haskell, allowing them to see how their code works more intuitively. For more experienced users, the editor provides a fun and engaging way to create graphics and animations using Haskell.

1.1.1 Educational Value

Imagine a first-year Computer Science class — a mix of students with varying levels of programming experience, from those who have never written a line of code to those who have been programming for years. Those who do have experience programming have

mostly worked with imperative languages like Python or Java, and are now learning Haskell for the first time. The students follow along, trying to understand the code and how it works, but as the program grows more complex, many start to get lost in the world of functional programming. They have difficulty seeing how different parts of the program fit together, or how the program's output changes as they modify the code.

This project aims to address this problem by providing students with an interactive, visual environment for learning Haskell, and by extension, functional programming. A graphical editor allows students to see more clearly how their code works, helping them understand how different parts of the program interact, while also providing a fun and engaging way to learn Haskell. By providing a web-based service, users never even need to install Haskell on their own computers — they can start coding right away, from any device with a web browser.

1.1.2 Practical Applications

Beyond its educational value, the project also has practical applications for users interested in creating graphics and animations using Haskell. The editor's simple interface and built-in graphics library make it easy to create images and animations, with the ability to experiment and iterate quickly. By including documentation for the graphics library and examples of how to use it, the project aims to make it easy for users to get started creating their own graphics and animations.

1.2 Dissertation Outline

Chapter 2 provides relevant background information on Haskell, functional programming, graphics, and web technologies. This is followed by an exploration of related works, placing this project in context of existing Haskell editors and graphics libraries.

Chapter 3 describes the design and implementation of the graphics library. It explores challenges and alternative design choices, explaining the rationale behind the decisions made, and the implementation of these choices.

In Chapter 4, the design and implementation of the website are detailed, illustrating technologies used and the development process, alongside the challenges faced. This includes the development and integration of the front-end and back-end components of the website. Chapters 3 and 4 can be read in any order, as they are largely independent. The order here provides a more logical progression, starting from core functionality, building to the final product.

Chapter 5 evaluates how successfully the project meets its aims and objectives, as well as how well it performs in practice. This includes a discussion of the user testing and feedback survey conducted, as well as any insights gained from the evaluation process.

Finally, Chapter 6 concludes the dissertation, summarising the project, reflecting on the process of creating it. It discusses the potential areas for future work and improvements to the project, including suggestions for how the project could be extended in the future, as well as limitations of the current implementation.

Chapter 2

Background

2.1 Functional Programming and Haskell

Functional programming is a paradigm built around the evaluation of pure mathematical functions (Hudak, 1989). Unlike the more widely used imperative programming, which utilises a series of statements to alter the state of a program and produce a result, functional programming is based on the composition and execution of functions to produce that result. The paradigm is based on lambda calculus, a formal system of computation, developed by Alonzo Church (1936) which is built around function application. Lambda calculus was proved to be equivalent to Turing machines by Alan Turing (1937).

One of the most widely used functional programming languages is Haskell, placing 31st in the TIOBE index for March 2025 (Tiobe, 2025). It is a statically typed, purely functional programming language, developed in the late 1980s, with the intention of consolidating features from existing pure lazy functional programming languages into a single, standardised language (Hudak et al., 2007). Haskell's namesake is Haskell Curry, who developed an equivalent system to lambda calculus, combinatory logic, alongside Moses Schönfinkel the 1920s and 1930s (Curry & Feys, 1958). Since its inception, there have been two major revisions of the Haskell language, Haskell 98 and Haskell 2010, with the latter being the most recent standard (Marlow, 2010).

Haskell is known for its use of type classes, introduced to the language by Phil Wadler (Wadler & Blott, 1988). These allow for ad-hoc polymorphism, where functions can be defined for a set of types which satisfy a certain interface. One of the most important type classes in Haskell is the `Monad` type class. Monads are a design pattern which provide a way to sequence computations and simulate effects in a pure functional language (Wadler, 1993).

2.2 Graphics

Computer graphics is a wide field, encompassing the use of computers for the creation, manipulation, and rendering of images, across both two and three-dimensional domains.

Although rudimentary graphics have been around longer, making use of cathode ray tube displays as far back as Braun (1897), the discipline of computer graphics was first truly established in the 1950s and 60s (Carlson, 2003), with the development of the first viable interactive graphical displays. Since then, the field has grown rapidly, resulting in a plethora of techniques and technologies for creating and rendering graphics.

Two-dimensional graphics are typically represented as raster graphics, where images are represented as a grid of pixels (Noll, 1971). This method of storage is simple, and reflects the same method that modern displays use to render images, but can result in loss of quality when images are scaled. An alternative method of storing graphics, is to use vector graphics, where shapes are described by mathematical equations, which are then converted into pixels when rendered (Sutherland, 1963). This is a more ideal solution for images which need to be scaled, as the image will not change in quality when resized. Both raster and vector graphics have their own advantages and disadvantages, and are used in different contexts depending on the requirements of the image. They are often used together, particularly on websites, with raster graphics being used for images which require fine detail, and vector graphics being used for images which need to be scaled.

2.2.1 Backends for Graphics

2.2.1.1 OpenGL

One of the most widely used graphics libraries is OpenGL, a cross-language, cross-platform library developed by Silicon Graphics in the early 1990s (Segal & Akeley, 1997). It supports both 2D and 3D graphics, and is generally used to interact with the graphics processing unit to take advantage of hardware acceleration, improving performance particularly for 3D graphics, but can also be used for central processing unit rendering. With several hundred procedures available within the API, OpenGL is a both a powerful and versatile library, making it ideal for a wide range of applications, from video games to scientific visualisation.

2.2.1.2 Cairo

Cairo is a 2D graphics library, designed to be both fast and portable (Worth & Packard, 2005). It is designed to produce consistent results across different platforms, and supports a wide range of output formats, including raster graphics, vector graphics, and PDFs. Although implemented in C, Cairo has bindings available for a number of languages, including Haskell, making it a versatile library for generating graphics.

2.2.2 Graphics in Haskell

2.2.2.1 Gloss

There are a number of libraries and bindings available for Haskell which aid in generating graphics. One of the most popular of these is the Gloss library, with a rating on Hackage of 2.75 out of 3 and 69,490 downloads. Gloss provides a simple interface for creating graphics in Haskell (Lippmeier, 2022). As with many graphics libraries,

Gloss uses OpenGL as its rendering engine, but provides a simpler and higher-level interface, making it ideal for beginners or for quick sketches. Due to the simplicity of Gloss, there are a limited number of functions available, which can make it difficult to create more complex graphics.

2.2.2.2 Diagrams

Another popular Haskell graphics library is Diagrams (Yorgey, 2023), which also has a Hackage rating of 2.75, but has fewer downloads than Gloss, at 34,898 (Yorgey, 2023). This provides a more powerful and flexible interface than Gloss, by allowing the user to choose from one of six backends, including Cairo, but not OpenGL. Diagrams' larger API makes it less beginner-friendly but more versatile due to its multiple backends.

2.2.3 Graphics in Web Browsers

The first web browser, originally named WorldWideWeb before becoming Nexus, was developed by Sir Tim Berners-Lee (1990b), alongside the first web server, and a basic version of HyperText Markup Language (HTML) (Berners-Lee, 1990a). Web browsers are inherently designed for rendering structured graphical content. While the first browsers were limited to displaying just HTML, developments in the last three decades, including the introduction of Cascading Style Sheets (CSS) (Lie, 1996), JavaScript (ECMA TC39 committee, 2024), and HTML5 (World Wide Web Consortium, 2011), have allowed for far more complex graphics.

2.2.3.1 Scalable Vector Graphics (SVGs)

Scalable Vector Graphics (SVGs) use an XML-based format, similar to HTML, to describe two-dimensional vector graphics (World Wide Web Consortium, 2014). Each object in an SVG is appended to the Document Object Model (DOM) of the web page, allowing the browser to automatically re-render the image if any changes are made to the SVG.

2.2.3.2 HTML5 Canvas API

The `canvas` element was first introduced by Apple in 2004 (Hixie, 2004), before being later standardised in the HTML5 specification (Mozilla Developer Connection, 2011) in 2011. By itself, the `canvas` element is nothing but a plain bitmap image. Where it becomes interesting, however, is when it is combined with JavaScript, allowing for the dynamic generation and rendering of graphics. Unlike SVGs, each component that makes up the image is not stored in the DOM, but is rendered directly to the canvas, which can result in better performance once the image is rendered, but requires the whole canvas to be redrawn if any changes are made.

2.2.3.3 WebGL

WebGL brings the capabilities of OpenGL to web browsers, allowing for hardware-accelerated 2D and 3D graphics to be rendered without the need for any plugins

(Khronos Group, 2011). Before being standardised by the Khronos Group, WebGL started out as two independent experiments by Mozilla and Opera (Johansson, 2007; Vukićević, 2007). WebGL renders inside the `canvas` element, and is based on OpenGL ES 2.0, a subset of OpenGL which is designed for embedded systems (Munshi & Leech, 2010).

2.3 Existing Work

This section will discuss two existing tools which provide web-based environments for creating graphics. The first, P5.js, is a JavaScript library which provides a simple interface for creating graphics in web browsers, and has its own web-based editor. The second, CodeWorld, is a tool which provides both a block-based and a Haskell-like interface for creating graphics in Haskell. These served as bases for the development of the tool described in this report.

2.3.1 Processing and P5.js

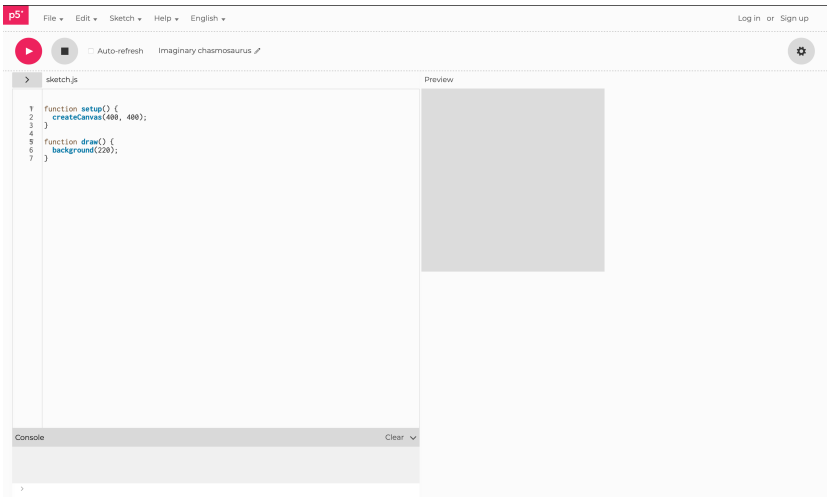

Figure 2.1: The P5.js editor.

Processing is a programming language and integrated development environment (IDE) based on Java (with bindings available for JavaScript, Python and Ruby), allowing for the creation of graphics and animations (Processing Foundation, 2018). P5.js is the official JavaScript binding for Processing, which uses the web browser to display graphics via the HTML `canvas` element (McCarthy, 2013a). Processing and P5.js sketches are built around two core functions: `setup()` which runs once at the start, and `draw()` which executes repeatedly to update the canvas. While Processing uses OpenGL as its rendering engine, P5.js uses the HTML5 Canvas API, with WebGL support available as an opt-in feature (McCarthy, 2013b) to improve performance and enable 3D graphics.

While there is no official support for Haskell from the Processing Foundation, there have been some unofficial bindings published on Hackage. One such package is processing

by Díaz (2016), which uses p5.js as a backend, producing its output using the HTML5 `canvas` element, but this was last updated in 2016, and has seemingly been abandoned. Another package, processing-for-haskell by Kholomiov (2016), was last updated in 2022, and implements almost the entire Processing API in Haskell, using OpenGL as the backend, just as the original Processing does, but does not support generating graphics in web browsers.

2.3.2 CodeWorld

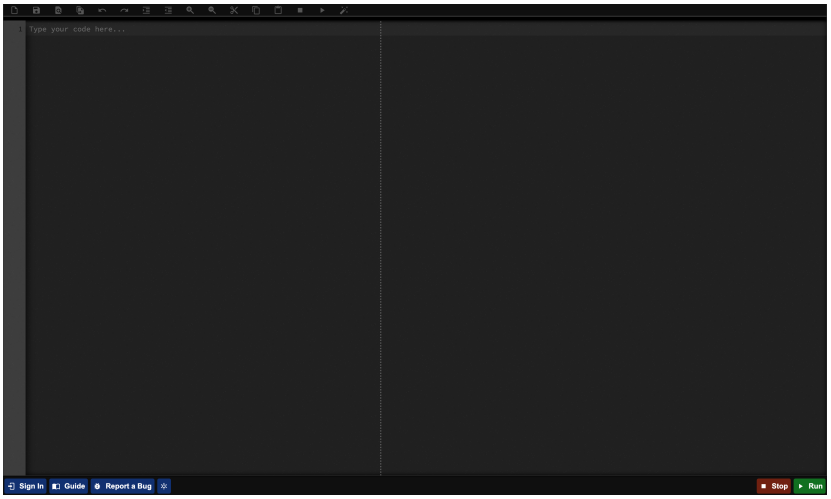

Figure 2.2: The CodeWorld editor.

CodeWorld is an educational website distributed by Google and created by Chris Smith (2014), which provides an environment for creating simple graphics using Haskell. It uses GHCJS, a Haskell to JavaScript compiler, to compile Haskell code into JavaScript, which is then executed in the browser.

There are three variants of CodeWorld available:

- The first (https://code.world) uses an educational, simplified version of Haskell 98, with some features removed to make it easier for beginners to learn.
- The second (https://code.world/haskell) uses a full version of Haskell 2010.
- The third (https://code.world/blocks) uses a drag and drop block editor, designed for beginners who are not yet comfortable writing code. According to their documentation, this version is not yet stable, so is not recommended for use.

Chapter 3

Designing and Implementing the Graphics Library

The graphics library is a key component of the system, as it provides the functionality for creating images and animations using Haskell. The implementation of the graphics library is largely independent of the rest of the system. It is responsible for generating the necessary data to render images and animations, rather than rendering them itself. The website described in Chapter 4 is responsible for rendering the images and animations at the correct frame rate, using the data generated by the graphics library.

Throughout this chapter, there are several code snippets taken from the implementation of the graphics library. In most cases, the implementations are trivial, so have been omitted for brevity, and in some cases, verbose type signatures have been replaced with ellipses. The full implementation of the graphics library can be found in Appendix C.

3.1 Library Overview

There were several sources of inspiration for the graphics library, including CodeWorld and P5.js. P5.js, as a JavaScript library, is inherently imperative, relying heavily on side effects and global state. Despite that, it provides a straightforward API, with a nice selection of functions, many of which can be adapted to fit the functional paradigm. CodeWorld, on the other hand, uses Haskell, so is purely functional, with no side effects or global state, relying instead on function composition and recursion. While CodeWorld's API provides a good example of a functional graphics library, it is less straightforward than P5.js. The aim for our library is to find a balance between the simplicity of P5.js, and the functional design of CodeWorld.

3.1.1 Required Modules

The first step in designing the graphics library was to consider what it needed to be able to do, from the perspective of a user:

- Represent a two-dimensional space to draw on, i.e. the canvas.

- Represent graphical primitives that can be drawn in that space, i.e. shapes.
- Apply transformations and colours to these shapes.
- Generate the data to allow the website to render the image or animation.

The graphics library was split into six modules:

- `Lib` — the main module, which exports the public API for the library. This has one function, `render`, which takes a `Canvas` and prints its JSON representation to the standard output. It is the responsibility of the website to render this as an image or animation. This function makes use of Haskell's `IO` monad, which allows for side effects, such as printing to the standard output.

```
1 render :: Canvas -> IO ()
```

Listing 3.1: The `render` function. The `IO ()` return type means that the function has side effects, specifically printing to the standard output, but no value is returned.

- `Internal` — an internal module, containing the types and functions used by the library, which are not exposed to the user. This includes the functions to convert each type to their JSON representation.
- `Canvas` — the module for creating and modifying the canvas.
- `Shape` — the module for creating and modifying graphical primitives.
- `Maths` — the module for providing various mathematical functions and types, such as vectors and angles.
- `Color` — the module for providing several representations for colours.

The following sections will provide an overview of these last four modules. This will start with the `Color` and `Maths` modules, as they are the simplest, and are required by the `Shape` and `Canvas` modules.

3.2 The `Color` Module

3.2.1 Common Digital Colour Representations

There are several ways to represent colours digitally, each with their own advantages and disadvantages. Some common representations are hexadecimal, RGB, HSL and HSV.

3.2.1.1 RGB and Hexadecimal

The hexadecimal and RGB (red, green, blue) representations are two ways of representing the same thing, with the former being more concise, and the latter more readable. They both represent colours as a combination of red, green and blue, with each component being an integer between 0 and 255. This is naturally suited to digital displays, which use red, green and blue light to create colours.

RGB uses a tuple of three base ten integers to represent the red, green and blue components, respectively. This is commonly written as a comma-separated list of integers, enclosed in parentheses, prefixed with `RGB`, e.g. `RGB(255, 0, 0)` for red.

Hexadecimal colours combine the red, green and blue components into a single base sixteen number. This is written as a string, starting with a hash symbol, followed by three pairs of hexadecimal digits, representing the red, green and blue components, e.g. `"#FF0000"` for red.

These representations can be visualised as a cube, with the x, y and z axes representing the red, green and blue components, (see Figure 3.1).

Both RGB and hexadecimal colours can be extended to support an alpha channel, allowing for transparency. RGB becomes RGBA, with the alpha channel represented as a floating point number between 0.0 and 1.0, where 0.0 is transparent, and 1.0 is opaque. The alpha channel in hexadecimal colours is represented as another pair of hexadecimal digits, with 00 being transparent, and FF being opaque.

3.2.1.2 HSL and HSV

HSL and HSV (also known as HSB) are almost identical, both having hue and saturation components. They differ in their final component. HSL use lightness, while HSV uses brightness (or value). These representations are visualised as a cylinder, with the hue, saturation and lightness/value components representing the angle, radius and height, respectively (see Figure 3.1).

The hue is an angle between 0° and 360°, with 0° and 360° representing red, 120° representing green, 240° representing blue. The saturation is a percentage between 0% and 100%, with 0% being grey, and 100% representing a fully saturated colour. The lightness/value component is also a percentage between 0% and 100%. For HSL, 0% represents black, 100% represents white, and 50% represents the colour itself. For HSV, 0% represents black, 100% represents the colour itself, and 50% represents white.

As with RGB and hexadecimal, HSL and HSV can also be extended to support an alpha channel, becoming HSLA and HSVA, with the alpha channel being the same as in RGBA.

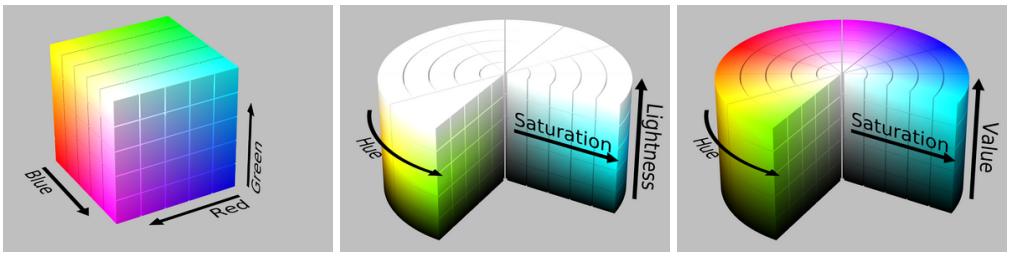

Figure 3.1: The RGB (left), HSL (centre) and HSV (right) colour spaces. Images taken from Wikipedia: HSL and HSV.

3.2.2 The `Color` Type

For the purposes of this library, it made sense to consider what other libraries used for colour representation. CodeWorld offers both RGB and HSL, as well as a limited selection of named colours. P5.js, uses any representation supported by CSS. Given that this library is designed to be used in a web browser, it made sense to follow P5.js' lead, and support all colour representations that CSS supports: hexadecimal, RGB, RGBA, HSL and HSLA, as well as a selection of named colours (World Wide Web Consortium, 2022).

The `Color` type has 153 constructors. Named colours comprise 148 of these constructors, with the remaining five constructors representing the other supported colour representations.

```
data Color
    = RGB Int Int Int -- Red, Green, Blue (0-255)
    | RGBA Int Int Int Float -- RGB + Alpha (0.0-1.0)
    | Hex String -- Hexadecimal, # optional, case insensitive
    | HSL Int Int Int -- Hue (0-360), Saturation + Lightness (0-100)
    | HSLA Int Int Int Float -- HSL + Alpha (0.0-1.0)
    | ... -- Type constructors for named colours omitted
```

Listing 3.2: The `Color` type definition. Named colours have been omitted, but are included in the actual implementation (see Appendix C).

3.3 The `Maths` Module

The majority of the mathematical functions a user would need are provided by Haskell's standard library. However, there are a few types and functions that are not provided, but are useful for a graphics library.

3.3.1 The `Length` Type

The first of these is the `Length` type, which is used to represent lengths, such as the width and height of the canvas, the radius of a circle or the side length of a square. It is a type alias for `Float`, providing a level of semantic clarity, particularly in the documentation.

```
type Length = Float
```

Listing 3.3: The `Length` type definition.

3.3.2 Angles

There are several units for measuring angles. The two most common are degrees and radians. Degrees divide a circle into 360 parts. This makes it easily divisible by many numbers, but is otherwise arbitrary. Radians divide it into 2π parts, such that one radian is the angle subtended at the centre of a circle by an arc equal in length to the radius. It

made sense to provide users with the ability to work with both of these, and functions for converting between them.

Two more type aliases for `Float`, `Degrees` and `Radians`, were defined to provide semantic clarity, as with the `Length` type. Functions for converting between degrees and radians were also provided.

```
type Degrees = Float
type Radians = Float
radians :: Degrees -> Radians
degrees :: Radians -> Degrees
```

Listing 3.4: The angle functions.

3.3.3 The `Vector` Type

Vectors are a fundamental part of any graphics library, as they can represent points, directions, velocities, forces, etc. As this library works with just two dimensions, our vectors only need two components, x and y. Both P5.js and CodeWorld provide a `Vector` type, with the latter also providing a `Point` type, which is identical to a `Vector`. For simplicity, we use a single `Vector` type to represent points, directions, and anything else the user may need. The `Vector` type has a single constructor with two type parameters, both of type `Float`. Note the use of the `Float` over `Length`, as vectors are often used to represent several concepts, as mentioned above. An alternative to using a custom `Vector` type would have been to use a tuple of `Floats`, but this would not provide the same level of type safety.

```
data Vector = Vector Float Float
```

Listing 3.5: The `Vector` type definition.

3.3.3.1 Vector Operations

Alongside the `Vector` type, it is useful to provide a simple set of functions for working with vectors. These include addition, subtraction, scalar multiplication, scalar division, dot product, magnitude, argument and normalisation. For the first four functions, the `Vector` type could have been made an instance of the `Num` type class. This would have required us to define all the functions in `Num`, which is unnecessary, as we only need a subset of them. Moreover, there is no defined division or multiplication for vectors, so it would be misleading to make `Vector` an instance of `Num`. We can instead define the operators `^+^`, `^-^`, `^*^` and `^/^`. These names reflect the standard `+`, `-`, `*` and `/` operators, but wrapped by carets (`^`) to indicate that they are for vectors, based on the standard mathematical notation for vectors, which uses a caret above the vector symbol.

```
(^+^) :: Vector -> Vector -> Vector -- Vector addition
(^-^) :: Vector -> Vector -> Vector -- Vector subtraction
(^*^) :: Vector -> Float -> Vector  -- Scalar multiplication
(^/^) :: Vector -> Float -> Vector  -- Scalar division
```

Listing 3.6: The vector operators.

The remaining functions are equally simple to define, and make use of several functions from the `Prelude` module. Note that there is no cross product function, as this is only defined for three-dimensional vectors.

```
mag :: Vector -> Length  -- The magnitude (length) of a vector
arg :: Vector -> Radians -- The argument (polar angle) of a vector
norm :: Vector -> Vector -- A vector with same arg, but mag = 1
dot :: Vector -> Vector -> Float -- The dot product of two vectors
```

Listing 3.7: The remaining vector functions.

3.3.4 Random Numbers

Finally, a set of functions for generating random numbers can be useful for creating more interesting animations, allowing for a level of unpredictability. As users are not able to access Haskell's `System` module with the online editor, they cannot use the `System.Random` functions for generating random numbers. Instead, a simple pseudo-random number generator was implemented using a linear congruential generator, which is a simple algorithm for generating pseudo-random numbers. This algorithm is defined by the recurrence relation

$$X_{n+1} = (aX_n + c) \bmod m$$

where X_0 is the seed, a is the multiplier, c is the increment, and m is the modulus.

```
randoms :: Int -> [Double]
randoms seed = map fst (iterate (lcg . snd) (lcg seed))
  where
    lcg :: Int -> (Double, Int)
    lcg s = (fromIntegral s' / fromIntegral (2 ^ 32), s')
      where
        s' = (1664525 * s + 1013904223) `mod` 2 ^ 32
```

Listing 3.8: .]The random number generator which uses a linear congruential generator to generate an infinite list of pseudo-random numbers, mapped to the range [0, 1].

The values of the multiplier ($a = 1664525$), the increment ($c = 1013904223$) and the modulus ($m = 2^{32}$) are taken from Equation 7.1.6, An Even Quicker Generator, in Chapter 7.1 of Numerical Recipes (Press et al., 1992).

A simple function to generate a seed was also implemented, using the current time from the `Data.Time.Clock.POSIX` module's `getPOSIXTime` function. Although the `getPOSIXTime` function does not come from Haskell's `Prelude` module, it is still safe to use, as it has no side effects, and the `System` module is not exposed to the user.

```
seed :: IO Int
seed = do
  time <- getPOSIXTime
  return (floor (time * 1000000))
```

Listing 3.9: The `seed` function.

The `seed` function returns an `IO Int`, as it requires the current time to generate the seed, which is not known until runtime. The IO monad manages side effects in Haskell in a purely functional manner.

3.4 The `Shape` Module

The `Shape` module is the most complex, and most important. It is responsible for creating and modifying shapes that can be drawn on the canvas. The module is split into two parts: graphical primitives and transformations.

3.4.1 Graphical Primitives

Graphical primitives are the basic building blocks for creating images and animations.

3.4.1.1 Examples from P5.js and CodeWorld

Once again, it made sense to use a similar set of primitives to those found in P5.js and CodeWorld, as they are simple and easy to use, making them accessible for beginners. An initial idea was to more or less translate the P5.js primitives to Haskell. This proved to be an inelegant solution, as many functions in P5.js have side effects, which would not work in a purely functional library. Instead, it seemed more appropriate to look at the names of these functions, but alter their parameters, and in some cases behaviours, to better suit our library.

P5.js provides nine primitives: `point`, `line`, `triangle`, `quad`, `rect`, `square`, `ellipse`, `circle` and `arc`. It also provides two functions for creating curves, which it does not describe as primitives: `bezier` and `curve`, the former for both quadratic and cubic Bézier curves, and the latter for Catmull-Rom spline curves. CodeWorld also provides nine primitives, along with variations to differentiate between certain properties such as filled and outlined shapes, which are omitted here: `blank`, `polyline`, `polygon`, `curve`, `rectangle`, `circle`, `arc`, `sector` and `lettering`.

3.4.1.2 Our Primitives

The primitives provided in this library needed to be simple, easy to use, and cover a wide range of use cases. All of our primitives are represented by one `Shape` type, with eight constructors. Each of these, except for `Empty` and `Group`, take a parameter of type `ShapeOptions` to represent the position (`Vector`), angle (`Radians`), fill colour (`Color`), stroke colour (`Color`) and stroke weight (`Float`). The remaining constructors and their parameters are as follows:

- `Empty`, representing the empty shape. This has no type parameters, and drawing it has no effect.
- `Group`, representing a group of shapes. This has a single type parameter of type `[Shape]`.

- `Line`, representing a straight line. This has just one type parameter, of type `Length`, representing the line's length.
- `Ellipse`, representing an ellipse. This has two type parameters, both of type `Length`, representing the ellipse's horizontal and vertical radii.
- `Rect`, representing a rectangle. This has two type parameters, both of type `Length`, representing the rectangle's width and height.
- `Polygon`, representing any polygon. This has one type parameter, of type `[Vector]`, representing the polygon's vertices.
- `Curve`, representing both quadratic and cubic Bézier curves. This has one type parameter, of type `[Vector]`, representing the curve's control points.
- `Arc`, representing an arc. This has five type parameters. Two of type `Length`, representing the horizontal and vertical radii of the arc, two of type `Radians`, representing the start and end angles of the arc, and one of type `Connection`, representing how the arc closes (either `Open`, `Chord` or `Pie`).

```
data Connection = Open | Chord | Pie
data ShapeOptions = ShapeOptions
    { _position :: Vector
    , _angle :: Radians
    , _fill :: Color
    , _stroke :: Color
    , _strokeWeight :: Float
    }
data Shape
    = Empty
    | Group [Shape]
    | Line
        { _length :: Length
        , _options :: ShapeOptions
        }
    | Ellipse
        { _horizontalAxis :: Length
        , _verticalAxis :: Length
        , _options :: ShapeOptions
        }
    | Rect
        { _width :: Length
        , _height :: Length
        , _options :: ShapeOptions
        }
    | Polygon
        { _points :: [Vector]
        , _options :: ShapeOptions
        }
    | Curve
        { _points :: [Vector]
        , _options :: ShapeOptions
        }
    | Arc
        { _horizontalAxis :: Length
        , _verticalAxis :: Length
```

```
        , _startAngle :: Radians
        , _endAngle :: Radians
        , _connect :: Connection
        , _options :: ShapeOptions
        }
```

Listing 3.10: The `Shape` type definition.

The `Shape` type is an abstract data type — it is exposed to the user, but its constructors are not. The user interacts with the library through the functions in Listing 3.11, which construct the required shapes. This allows the user to create shapes with default options, and then modify them as needed. For every shape, the default options are as follows:

- The position is the canvas origin, i.e. `Vector 0 0`.
- The angle is 0 radians.
- The fill colour is `Transparent`.
- The stroke colour is `Black`.
- The stroke weight is 1.

```
empty :: Shape
line :: Length -> Shape
ellipse :: Length -> Length -> Shape
circle :: Length -> Shape
rect :: Length -> Length -> Shape
square :: Length -> Shape
polygon :: [Vector] -> Shape
regular :: Int -> Length -> Shape
bezier2 :: Vector -> Vector -> Shape
bezier3 :: Vector -> Vector -> Vector -> Shape
arc :: Length -> Length -> Radians -> Radians -> Shape
segment :: Length -> Length -> Radians -> Radians -> Shape
pie :: Length -> Length -> Radians -> Radians -> Shape
```

Listing 3.11: The functions to create shapes.

Many of these functions directly correspond to the constructors of the `Shape` type. The `circle` and `square` functions are simply special cases of the `ellipse` and `rect` functions, respectively. The `regular` function is used to create regular polygons, and takes two arguments: the number of sides, and the radius of the circumcircle. The `bezier2` and `bezier3` functions are used to create quadratic and cubic Bézier curves, and take two and three control points, respectively. They use the `Curve` constructor, but with a different number of control points. The `arc`, `segment` and `pie` functions all use the `Arc` constructor, but with different values for the `Connection` parameter; `Open`, `Chord` and `Pie`, respectively.

The final constructor is the `Group` constructor, which is used to group shapes together. This is useful for creating more complex shapes, as well as for applying transformations to multiple shapes at once. To create a group of shapes, the user can use the `&` operator. This is a simple infix operator that takes two shapes, and returns a group of those shapes.

The name of the operator is taken directly from CodeWorld, where it is also used to group shapes together.

```
(&) :: Shape -> Shape -> Shape
```

Listing 3.12: The group (`&`) operator.

3.4.2 Transformations

There are broadly speaking two categories of transformations to consider: colour transformations and geometric transformations.

3.4.2.1 Colour Transformations

Colour transformations change the colour of a shape. Several colour transformations are commonly used in graphics libraries, including setting the fill colour, setting the outline colour, and setting the outline thickness. In P5.js, this is done using the `fill`, `stroke` and `strokeWeight` functions, respectively. These alter a global state, so that all subsequent shapes are drawn with the specified properties. In CodeWorld, setting the stroke colour is done using the `colored` function and the regular variant of the shape function (e.g. `circle`), while setting the fill colour is done using the `colored` function and the solid variant of the shape function (e.g. `solidCircle`). The stroke weight can be increased by using the thick variant of the shape function (e.g. `thickCircle`).

P5.js' approach is more user-friendly, as it allows the user to set the fill, stroke and stroke weight for each shape more easily, but CodeWorld's approach is more functional, as it avoids global state. Our design takes the best of both, by retargeting the P5.js functions to transform a given shape, rather than to set a global variable.

```
fill :: Color -> Shape -> Shape -- Set shape's fill colour
stroke :: Color -> Shape -> Shape -- Set shape's stroke colour
strokeWeight :: Float -> Shape -> Shape -- Set shape's stroke weight
noFill :: Shape -> Shape -- fill Transparent
noStroke :: Shape -> Shape -- stroke Transparent
```

Listing 3.13: The colour transformation functions.

3.4.2.2 Geometric Transformations

Several geometric transformations are commonly used in graphics libraries, including translations, rotations and scaling. In P5.js, individual shapes do not need to be translated, as the user specifies the position of each shape when creating it. Instead, P5.js' `translate` function moves the canvas' origin, so that all subsequent shapes are drawn relative to the new origin. This can be useful but also confusing, particularly for beginners. The `rotate` and `scale` functions behave similarly, acting on the canvas coordinates rather than individual shapes. CodeWorld takes a more intuitive approach, providing functions for translating, rotating and scaling individual shapes. This library followed CodeWorld's approach, as it fits better with the functional programming paradigm, and is more intuitive for beginners.

```
translate :: Vector -> Shape -> Shape -- Move shape by given vector
rotate :: Radians -> Shape -> Shape -- Rotate shape by given angle
scale :: Float -> Shape -> Shape -- Scale shape by given factor
translateX :: Length -> Shape -> Shape -- translate (Vector x 0)
translateY :: Length -> Shape -> Shape -- translate (Vector 0 y)
```

Listing 3.14: The geometric transformation functions.

3.4.2.3 Applying Transformations

Transformations can be applied to a shape by directly calling the appropriate function.

```
translate (Vector 5 5) (rotate pi (scale 2 (rect 10 50)))
```

This is hard to read when applying multiple transformations to a shape. Another approach is to compose the functions, applying the resulting function to the shape.

```
(translate (Vector 5 5) . rotate pi . scale 2) $ rect 10 50
```

This is more readable, but as the order of the transformations is reversed, it could still be confusing for beginners. To solve this, an operator was implemented to apply transformations to shapes.

```
(>>>) :: Shape -> (Shape -> Shape) -> Shape
```

Listing 3.15: The transformation application (>>>) operator.

Now the user can apply transformations to shapes in a readable and intuitive fashion.

```
rect 10 50 >>> scale 2 >>> rotate pi >>> translate (Vector 5 5)
```

3.5 The `Canvas` Module

This is responsible for creating and modifying the canvas. The `Canvas` type has five type parameters:

- Two `Lengths` to represent the width and height.
- An `Int` to represent the frame rate at which to render the animation.
- A `Color` to represent the background colour to apply to each frame. There is an argument for omitting this, requiring users to draw a rectangle filling the whole canvas, and applying the background colour there. However, as most animations are likely to use the same background colour for each frame, this provides a more elegant solution.
- A `[Shape]` to represent the frames of the animation.

```
data Canvas = Canvas
    { _width :: Length
    , _height :: Length
    , _fps :: Int
```

```
    , _backgroundColor :: Color
    , _frames :: [Shape]
    }
```

Listing 3.16: The `Canvas` type definition.

Once the `Canvas` type was defined, we needed an interface for users to interact with it, starting with a function to create a canvas. One option was to take all the canvas parameters as arguments. However, this would require the user to write out each parameter in every program they write, even if these values are not important to them. To provide a more user-friendly interface, the function provides default values for certain parameters, requiring the user to only specify the width and height. Specifically, the default values for the frame rate and background colour are 24 and `Transparent`, respectively, while the default value for the frames is an empty list. To modify the frame rate and background colour, the `fps` and `background` functions were provided.

To allow the user to draw shapes on the canvas, we needed a way to append shapes to the list of frames. This was handled by two operators, `(<<<)`, which adds a single `Shape`, and `(<<<:)`, which appends a list of `Shapes`. The `(<<<:)` operator was not strictly necessary, as users could instead use `foldl (<<<) canvas shapes`, but it provided a more elegant solution. Using `foldl` with `(<<<)` would not allow the user to take advantage of Haskell's lazy evaluation, as the list of shapes would be fully evaluated due to the tail-recursive nature of `foldl`.

```
createCanvas :: Length -> Length -> Canvas
fps :: Int -> Canvas -> Canvas
background :: Color -> Canvas -> Canvas
(<<<) :: Canvas -> Shape -> Canvas
(<<<:) :: Canvas -> [Shape] -> Canvas
```

Listing 3.17: The `createCanvas`, `fps` and `background` functions and the `(<<<)` and `(<<<:)` operators.

A final function was defined within the `Canvas` module, to translate shapes to the centre of the canvas. It was defined in this module because it requires the canvas as an argument, as well as the shape. Defining it in the `Shape` module would require importing the `Canvas` module, which would create a circular dependency.

The implementation of this translation was dependent on the shape's origin. For shapes whose origin is at their centre, such as circles and ellipses, centring the shape was as simple as translating it by half the width and height of the canvas. For shapes whose origin is at their top-left corner, such as rectangles and polygons, centring the shape was more complex. It required translating the shape by half the width and height of the canvas, then translating it back by half the width and height of the shape. These shapes also needed to be translated to account for their rotation.

```
center :: Canvas -> Shape -> Shape
```

Listing 3.18: The `center` function.

Chapter 4

The Website — Designing and Implementing an Elegant Interface

There were two options for how to implement the user interface: a desktop application or a website. A desktop application would have allowed users to run their programs locally, safely using any Haskell library they wanted. However, this would have required users to install the application, and Haskell itself, which could alienate users who are unfamiliar with installing software. A website would allow users to run their programs in the browser, without need to install anything. This would make the system more accessible to users, and compatible across multiple platforms. For these reasons, the system was implemented as a website.

The website is the interface for users to interact with the system, so it was important that the website be user-friendly and easy to use. This chapter discusses its design, implementation, and the technologies used to build it.

4.1 Design Principles

Websites have been at the core of people's lives for the past two decades. They are used for a variety of activities, from shopping to socialising to filing taxes. It is thus essential that websites meet certain design standards, to ensure that users can easily navigate and interact with them. It is also important that websites are visually appealing, to keep users engaged. Due to these requirements, many standard design practices have emerged, which are important to consider when designing a website.

4.1.1 Purpose — What is the site for and how do we achieve that?

The first thing to consider when designing a website is its purpose. For this project, the website was designed to allow users to write and run programs in Haskell. It had to integrate seamlessly with the graphics library discussed in Chapter 3, and render both the textual and graphical results of programs.

It would have been easy to say that an editor for Haskell, integrated with the graphics

library, and a way to execute programs would be sufficient, and to ignore any further functionality. However, this would have been a mistake. The website had to be designed with user-friendliness in mind, and to provide users with all the tools and information they might need. This meant that the website would need four main pages:

- The editor, integrated with the graphics library, for users write, run, save and share their programs.
- The reference page, for users to learn how to use the graphics library.
- The homepage, to showcase the capabilities of the library and encourage users to try it out.
- The account page, for users to view and manage their saved programs and their account settings.

There were a few additional pages, such as a login page, a registration page, and a privacy policy page.

4.1.2 Key Considerations

4.1.2.1 User Experience — How can we make the website easy to use?

A website should be easy to use, with a clear and intuitive interface that guides users through the various features and functions of the site. Most websites follow a standard layout (see Figure 4.1), with a header at the top of the page containing the site's logo and navigation links, a main content area in the centre of the page, and a footer at the bottom of the page for additional links and information. It is also recommended limiting text to between 45 and 75 characters wide, as longer or shorter lines make it difficult for users to read (Rutter, 2005).

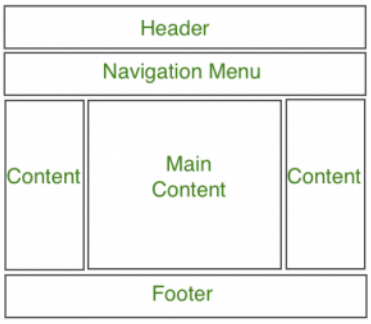

Figure 4.1: Standard website layout. Image taken from Geek for Geeks: CSS Website Layout.

4.1.2.2 Visual Design — How can we make the website visually appealing?

The website should be visually appealing, with a clean and modern design to reflect the brand and purpose of the site. It should use a consistent colour scheme, and typography with clear and legible text. Images and graphics can enhance the visual appeal of the site, but should be used sparingly and to complement the overall design (Krause, 2023).

For this site, it made sense to use a colour scheme based on the Haskell logo (see Figure 4.2) to reflect the use of Haskell.

Figure 4.2: Coloured Haskell logo (left) and monochrome Haskell logo (right). Images taken from Haskell.org and Haskell Wiki.

4.1.2.3 Accessibility — How can we make the website accessible to all users?

Accessibility is essential, and often overlooked, when designing a website. Websites should be accessible to all users, including those with disabilities, or visual impairments such as colour blindness. To keep accessibility in mind, basic features such as alt text for images, high contrast text, and semantic HTML for screen readers should be used. Websites should also be responsive, adapting to different screen sizes and devices.

4.2 Technologies

There were a number of technologies used to implement the website. While all websites are built with some combination of HTML, CSS, and JavaScript, there are a number of frameworks, libraries and tools to make the development process easier and more efficient.

TypeScript TypeScript is a superset of JavaScript that adds static typing to the language, and is maintained by Microsoft. TypeScript compiles to plain JavaScript, so can be run in any web browser. The addition of static typing makes it easier to catch errors at compile-time, preventing many runtime bugs, making the codebase more maintainable.

React React is a JavaScript library for building user interfaces. It is maintained by Facebook and is the most widely used front-end framework according to the Stack Overflow Developer Survey 2024 (StackOverflow, 2024). React allows developers to build reusable components that can be combined to create complex user interfaces. Components are written in JSX, a syntax extension for JavaScript to allow developers to write HTML-like code in JavaScript files. Similarly to TypeScript, React compiles to plain JavaScript, so it can be run in any browser. React provides full support for building single-page applications, which load a single HTML document and dynamically update the content of the page using JavaScript. This can improve performance, as it reduces the number of requests that need to be made to the server, as well as providing a smoother user experience.

Next.js Next.js is a framework built on top of React, maintained by Vercel. It is one of the most popular frameworks for building websites, and the decision to use it here

was based largely on familiarity with the framework. Next.js provides many built-in optimizations for performance and search engine optimisation, and makes it easy to communicate between the client and server. It also provides several tools for handling routing, data fetching, and other common tasks in web development. Next.js uses a file system-based routing system, where each page is represented by a file in the project directory. This makes it easy to add new pages to the website by creating a new file in the correct location.

Material UI There are a number of user interface libraries available, which provide pre-built components that can be used to build websites. Material UI is library of highly customisable React components, implementing Google's Material Design guidelines. This saves time in development, as developers do not need to spend time building components from scratch. It also provides a consistent look and feel across the website, which can help to improve the user experience.

MongoDB MongoDB is a NoSQL database that stores data in a flexible, JSON-like format. It is widely used in the industry for its scalability, flexibility, and ease of use. Any database could have been used for this project, but as with Next.js, MongoDB was chosen for its ease of use and familiarity.

Docker Docker is a tool that allows developers to package their applications into containers, which can then be run on any machine that has Docker installed. Docker containers are isolated from the host, preventing access to its file system or unauthorised network.

4.3 Tackling the Editor

The Haskell editor is the core of the website, and was the most complex part to implement. It was prudent to tackle this part first, to ensure that it could be implemented effectively before moving on to the other parts of the website.

4.3.1 Running Haskell from the Browser

Web browsers are designed to do one thing, and do it well: display web pages. They can display content formatted with HTML, style it with CSS, and add interactivity with JavaScript. They cannot run Haskell. To run Haskell from the browser, there were two options: compile the Haskell to JavaScript using GHCJS, or run Haskell on a server and send the output to the browser. While the former would have kept the system more self-contained, GHCJS does not currently support the latest version of Haskell. At the time of writing, GHCJS only supports up to GHC 8.10, while the latest version of GHC is 9.12.1, suggesting that it is not actively maintained. For this reason, the latter option was chosen.

4.3.1.1 Running Haskell Code Securely

Running arbitrary, potentially malicious code on the server is a security risk. To mitigate this, the Haskell code was run in a sandboxed environment, using Docker containers. As an extra layer of security, the only Haskell libraries made available to the user were our custom graphics library and Haskell's `Prelude` library.

Next.js server actions were used to communicate between the client and server. These are functions which can be called by the client to perform server-side tasks. The server action executes the user's Haskell code in a Docker container and streams the output back to the client.

This, utilised the `exec` function from the `child_process` module in Node.js to spawn a shell and run the Docker container (see Listing 4.1). The Docker container uses the `haskell:latest` image, which includes the Haskell compiler.

```
1 docker run --rm -m 128m --cpus=0.5 haskell:latest bash -c "..."
```

Listing 4.1: The command used to run the Haskell code in a Docker container. The `--rm` flag ensures the container is properly disposed of after running, the `-m 128m` flag limits the container to 128 MB of memory, and the `--cpus=0.5` flag limits the container to half a CPU core. The `bash -c "..."` part executes a command inside the container. This command is a simple script to write the user's program and the graphics library to appropriate files, compile them using GHC, and run the resulting executable. All code was sent as a base64-encoded string to prevent special characters from escaping the command, posing a security risk.

As Haskell programs can produce an infinite output, the server action created a `ReadableStream` object, which was updated with new output as it was produced. To prevent user programs from running indefinitely, both the stream's listening time and the program's execution time were limited to five minutes. The listener also had a timeout of two and a half seconds, terminating it if no data was received in that time, as the program was presumed to have finished running. These numbers were chosen to suit the performance of the server being used. Initially, a timeout of one second was used, which proved to be too short, leading to the stream occasionally being killed prematurely. If the system were to be deployed to a different server, these numbers would need to be adjusted accordingly.

4.3.1.2 Reading the Stream

A custom React hook was created to read the output stream on the client side. Using this hook in a component allows that component to control the execution and termination of the stream, and display and clear the output. When the stream is running, the component automatically re-renders as new data is received.

The hook takes a single parameter: a server action which returns a `ReadableStream` object. The stream was stored using React's `useRef` hook, allowing it to persist between renders. The hook defines a new state variable, `data`, to store the output of the stream using React's `useState` hook. This provides a function, `setData`, to update the state variable, triggering the component to re-render. The hook then defines three

functions: `executeStream`, to run the server action and populate the `data` variable until the stream is empty, `terminateStream`, to cancel the stream, and `clearStream`, to clear the output data. The hook returns an array containing `data`, `executeStream`, `terminateStream`, and `clearStream`.

4.3.2 Editor Components

The editor page needed four main components:

- A code editor, for users to write their Haskell programs.
- A graphics display, for users to view the output of their programs.
- A console, for users to view any errors, warnings or debugging statements generated by their programs.
- A toolbar, for users to run, stop, save or share their programs.

These components were contained within a parent component which manages the state of the page, keeping track of the user's code and managing the execution of the program.

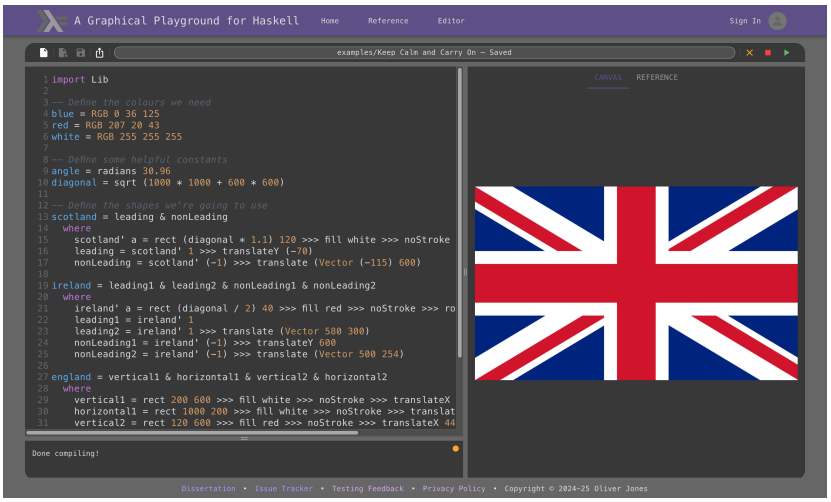

Figure 4.3: The editor page. At the top is website's header, containing the monochrome Haskell logo to make it visible against the background, the website's name, the navigation links, and a sign-in button. Below are the four editor components outlined above. The toolbar is at the top, the code editor on the left, the graphics display on the right, and the console below the code editor.

4.3.2.1 The Code Editor

There are a wide variety of elements built-in to HTML. One of these is the `<textarea>` element, which creates a multi-line text input. This can be used to create a simple code editor. Unfortunately, you cannot change the colour of individual parts of the text in a `<textarea>` element, so it is not suitable for syntax highlighting.

This posed a challenge, as a good code editor should provide syntax highlighting to help users identify parts of their code. The solution was to layer another element, in this

case the `<pre>` element, behind the `<textarea>` element, then setting the background and text colours of the `<textarea>` to `transparent`. The `<pre>` element preserves whitespace and line breaks, making it ideal for displaying code.

An event listener was added to the `<textarea>` element, which listens for changes to the text. The content of the `<pre>` element was then updated with the syntax-highlighted version of the code whenever the user types. The `highlight.js` library was used to provide syntax highlighting. This supports multiple languages, and colour themes. It wraps keywords in `<span>` elements with a class corresponding to the type of keyword, which can then be styled using CSS.

A second event listener was added the `<textarea>` element to listen for changes to the scroll position, and update the scroll position of the `<pre>` element accordingly. By setting the `font-size`, `line-height` and `letter-spacing` CSS properties of both the `<textarea>` and `<pre>` elements to the same values, the two elements lined up perfectly. This created the illusion that the code is typed directly into the `<pre>` element.

Another useful feature of a code editor is automatic line numbering, which can help users keep track of where they are in their code. To achieve this, each line was split into a separate `<code>` element. The line number was then rendered using the `::before` pseudo-element in CSS, which can insert extra content before an element. CSS also provides a built-in `counter()` function, which can be used to automatically number the lines of code.

4.3.2.2 The Console

This was the simplest component to implement as it is just displays the output of the user's program as it runs. A simple React component which receives a string as an input and renders it as a block of text was all that was needed initially. Later, this component was expanded to include a status indicator, which displays whether the program is currently running, has finished, or raised an error. Originally, this was indicated with extra text in the console, but this proved to be confusing for users.

4.3.2.3 The Graphics Display

The original plan for this component was quite simple, just a canvas element, with a function to parse the JSON output of the user's program and draw the graphics on the canvas. However, this was more complex than anticipated. Due to the nature of React, when a component's state updates, the component and its children are re-rendered. With the user's code stored in the state of the editor component, changes to the code caused the editor component to re-render, triggering all of its children to re-render as well. This posed an issue for the graphics component, as every re-render would, at best, cause a small flicker as the rendered image is re-rendered, and at worst, cause an animation to restart. Fortunately, React provides a solution to this problem. The graphics component was memoised using the `React.memo` function, meaning that it would only re-render when its props change, rather than when its parent component re-renders.

With this solution, it was straightforward to wait for the user's program to finish running, then parse the output and draw the graphics on the canvas. However, this meant that the

user had to wait for the program to finish running before they could see any graphics. By implementing an intermediate controller component, the graphics could be drawn as the program runs, allowing the user to see the graphics being drawn in real-time. This intermediate component was also memoised, so it only re-rendered as more data was received from the server. It was then responsible for parsing the JSON data, and passing it to the graphics component to be drawn on the canvas at the correct time, based on the frame rate of the animation.

4.3.2.4 The Toolbar

Figure 4.4: The toolbar component.

On the left are four buttons: new, save, open and share. As the names suggest, these buttons allow users to create a new program, replacing the content of the code editor with some boilerplate code, save their current program, open a saved program, and share their program. Saving and opening programs are only available to users who are logged in. Sharing programs is available to all users. Sharing gives the user the option of copying their program to the clipboard, generating and copying a URL which links directly to the program, copying the image output of the program to the clipboard, or downloading the image output of the program. The name of the program is displayed in the centre of the toolbar, along with the name of the user who wrote it. To the right are three buttons: clear, run and stop. The clear button clears the console and graphics display, the run button runs the user's program, and the stop button stops the program if it is currently running.

4.3.3 Making the Components Resizable

Making the components resizable was relatively straightforward. A custom `SplitView` component was created, which takes two child components, and creates a draggable border between them. Using this component twice created a three-way split view. A simplified version of the layout is shown in Listing 4.2.

```
<SplitView>
    <SplitView vertical>
        <CodeEditor />
        <Console />
    </SplitView>
    <GraphicsController />
</SplitView>
```

Listing 4.2: A simplified version of the editor page layout.

While React allows any HTML element to be marked as `draggable`, this causes the element to be movable in both the x and y directions, which can cause strangle visual artefacts when dragging the border. To prevent this, a React state variable was used to store whether the user is currently dragging the border. When the user clicks on the

border, this variable was set to `true`, and when the user releases the mouse button, it is set back to `false`. An event listener was then added to check for mouse movements. If the dragging variable is `true`, we would update the sizes of the two child components based on the mouse position. This produced two resizable components without any visual artefacts.

4.4 Making a Reference Page

The reference page was considerably simpler to implement than the editor page. It is simply a long page of text, with a table of contents at the top, linking to the various sections of the page. The page was divided into sections, with alternating coloured backgrounds to make it easier to read. Each section detailed a different aspect of the graphics library:

- "Haskell", with a brief explanation of Haskell's `Prelude` module, a link to its full documentation, and an explanation of Haskell's type signature syntax.
- "Canvas", to explain the concept of the canvas, and how to create it.
- "Images and Animations", to explain how to draw images and animations on the canvas and modify the frame rate and background colour.
- "Vectors", to explain how to create and manipulate vectors.
- "Shapes", to explain how to create the various shapes available in the graphics library, with pictures of each shape.
- "Transformations", to explain how to manipulate these shapes, with pictures of each transformation.
- "Colors", to explain how to create colours using the various colour spaces available in the graphics library.
- "Other", to explain how to use the other functions available in the graphics library.

Throughout the page are a series of examples, demonstrating how to use the functions available in the graphics library. These are written in both inline and block code, which are styled accordingly to make them clear.

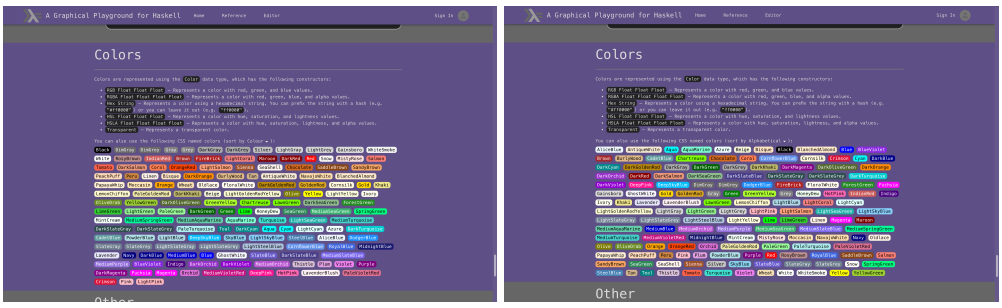

Figure 4.5: The "Colors" section of the reference page. On the left, the colours are sorted by hue, saturation, and lightness. On the right, the colours are sorted alphabetically.

The "Color" section also includes a list of all the named colours available in the graphics library (see Figure 4.5). Each colour is displayed as its name, highlighted in that colour. Hovering over these displays a tooltip with the hexadecimal, RGB and HSL values of the colour. The list can be sorted either by colour, a combination of hue, then saturation, then lightness, or alphabetically by name, to accommodate users who are colour-blind.

4.5 Creating a User Account System

User accounts are nothing new to the internet. They are used for storing user data, and to provide a more personalised experience for the user. For this website, user accounts were used to store the user's saved programs.

Next.js provides built-in support for middleware, which can be used to direct users to the correct page based on their authentication status. Users who are not logged in, who try to access the account page, are redirected to the login page. Users who are logged in, who try to access the login page, are redirected to the account page. This ensures that users are always directed to the correct page, based on their authentication status.

The user account system is relatively simple. Users can create an account by providing an email address and password. The password is hashed using the `bcrypt` library, and the email address is stored in plain text. When a user logs in, the server checks the email address and password against the database, and if they match, the user is logged in. A unique identifier is generated for the user, and stored in a cookie, which is sent to the client. This cookie is used to authenticate the user on subsequent requests, so they do not need to log in every time they try to access a protected endpoint.

The user account system also allows users to save their programs. When a user saves a program, it is stored in the database, along with the user's unique identifier. When a user logs in, the server retrieves all the programs associated with that user's unique identifier, and displays them on the account page.

4.6 Landing on the Homepage

The homepage is the first page users see when they visit the website. It is designed to be engaging and informative, showcasing the capabilities of the graphics library, and encouraging users to try it out.

The top of the homepage features a large banner, indicating the name of the website. Below is a brief welcome message, encouraging users to try out the graphics library. Below this is a series of examples, showcasing the capabilities of the graphics library — both static images and animations. Finally, at the bottom of the page is a help section, directing users to the reference page for more information about the graphics library, and to the issue tracker if they encounter any problems.

Figure 4.6: The homepage.

4.7 Quality-of-Life and Legal Requirements

There were a few features that were added to the website to make it more user-friendly, and to comply with legal requirements. These included:

- A notification banner which appears at the bottom corner of the screen whenever a user carries out certain actions such as saving a program, copying their program or URL via the share, etc. The banner automatically fades after a few seconds, but does not disappear entirely until the user dismisses it.
- A notification informing the user that the website uses cookies, as is legally required by GDPR regulations. This makes use of the same notification banner as the other notifications, and includes a link to the privacy policy.
- The footer includes links to this dissertation, the GitHub issue tracker, the user testing and feedback form, and the privacy policy. The privacy policy is a legal requirement, and outlines how the website collects, stores and uses user data. It also includes a copyright notice. While this is not strictly necessary, as UK copyright law protects the author's rights to their work, regardless of whether they include a copyright notice, it is a good practice to include one, as many users may not be aware of this.
- A saved state indicator appears in the menu bar, informing a user of whether they have an open program, and if it is saved. Hovering over this indicator displays a tooltip with the name of the program.
- Confirmation menus appear when a user tries to open a program while they have an unsaved program open, preventing them from losing their work. The same is true when a user tries to delete one of their saved programs.

Chapter 5

Testing and Evaluation

Testing a website is not quite as simple as testing other software. Websites are not standalone applications, but rather a collection of files that are interpreted by a web browser. These browsers can vary greatly in their implementation of web standards, and the same website can look and behave differently in different browsers. In particular, CSS can be a source of many problems. This chapter will discuss the testing methodology used to ensure that the website is functional and accessible to all users.

5.1 Testing Methodology

5.1.1 Browser and Device Compatibility

To ensure compatibility between browsers, the website was tested in three of the four most popular web browsers (StatCounter, 2025): Google Chrome, Safari and Mozilla Firefox. While Microsoft Edge is more popular web browser than Firefox, it is based on Chromium, the same engine as Google Chrome, and so produces largely the same results.

The website was also tested on devices with different screen sizes to ensure that the website is responsive. A 28-inch monitor, a 14-inch laptop, an 11-inch tablet, a 6.7-inch smartphone and a 5.85-inch smartphone. To test the transitions between these sizes, the website was tested using Safari's "Responsive Design Mode". In particular, Safari and Chrome were tested on all devices, as these browsers are the most popular on mobile devices, while Firefox was only tested on the 28-inch monitor and the 14-inch laptop.

5.1.2 CSS and JavaScript Robustness

As CSS and JavaScript are the backbone of the website, it is important to ensure that they are robust. Fortunately, the Material UI library is well-tested and widely used, so many components are already tested. However, the website also includes custom CSS and JavaScript, which needed to be tested. Components which display variable amounts of data, such as the code editor and the console output, were tested with a range of data

to ensure that they behave as expected. This included short strings, long strings which exceeded the size of their container, and strings with special characters.

5.1.3 The Graphics Library

The graphics library was tested by creating a series of programs, covering every feature of the library. This included testing every primitive shape, in combination with every transformation function, to ensure that they all worked as expected. This came to a total of ten shapes and seven transformations (including the `center` function, which uses the `translate` function, but behaves differently for different shapes) which resulted in 70 combinations. These programs were then run on the website to ensure that they worked as expected. The results of these tests were then compared to the expected behaviour, as documented in the reference page, to ensure that these match.

5.2 Testing Results

On the whole, the website performed well in testing. The website was responsive on all devices tested, and the transitions between different screen sizes were smooth. The majority of the web browsers tested, displayed the website correctly.

There were some exceptions, however.

5.2.1 Resizable Areas

When the console output exceeds the size of the console, the console becomes scrollable, and the page should not resize. This worked as expected, until the content of the console area became taller than the height of its parent container (that being the total height of both the code editor and the console). At this point, the entire page would resize to accommodate the console, which was not the desired behaviour.

This was caused by an error in the calculation of the size of each area. This calculation was done using pixel values, which are fixed, rather than percentages, which are relative. Changing the calculation to use percentages fixed this issue.

5.2.2 Scrolling in Firefox

The program examples on the homepage were designed to scroll to the middle automatically when the page is loaded. While this worked perfectly on other browsers, allowing users to scroll in both directions, Firefox did not allow users to scroll left. Instead, the contents of the scrollable area were correctly centred, but the elements which were now off-screen to the left could not be accessed. This particular error was rectified by using some extra JavaScript to set the correct scroll position after the page had loaded.

5.2.3 The Embedded Reference Page

The embedded reference page on the editor page caused two issues, one in all browsers, and one in Firefox.

The Firefox error was once again related to scrolling. Switching to the reference tab would resize the page to accommodate the reference page, instead of making just the right column scrollable. This was fixed by adding a calculation in CSS to ensure that the reference page could not exceed the height of the parent container.

The second issue allowed the components of the reference page to exceed the width of their parent container. A simple fix for this was to add a new `<div>` element around the reference page, and set the width of this element to 100%. This ensured that the reference page could not exceed the width of its parent container, and would resize correctly when the page was resized.

5.2.4 The Graphics Library

For the most part, the graphics library worked as expected. One issue that was encountered, however, was a limitation of working with infinite lists. If the user tries to render an animation with an infinite number of frames, the browser quickly drops in performance, becoming unresponsive unless refreshed. This is a limitation of how the website is implemented, and not a limitation of the graphics library itself. This limitation was documented in the reference page. The site will automatically loop the animation, so users can still create the appearance of infinite animations by using a finite number of frames, and having the first and last frames of the animation be the same.

5.3 User Testing and Feedback

A survey was created to test how users interacted with the website. The survey asked users to complete a short series of tasks including:

- Writing and running a basic Haskell program, without graphics.
- Learning how to produce a basic graphic by using the reference page.
- Altering a pre-existing program to produce a different image.
- Altering a pre-existing program to produce a different animation.

Participants were asked to rate the ease of each task from one to ten, with one being very difficult and ten being very easy, and to state roughly how long each task took. They were also asked to provide any feedback they had on the website, both positive and negative, and to identify any bugs they encountered. Participants were kept anonymous, but were asked to describe their qualifications in Computer Science, and their experience with Haskell.

5.3.1 Survey Responses

The survey was shared among Informatics students at the University of Edinburgh, and was posted on the Haskell subreddit. The survey received a total of 11 responses, alongside additional feedback from members of the subreddit.

5.3.1.1 User Experience and Qualifications

Three of the participants have the equivalent of (or are currently studying for) an A-Level in Computer Science. Two of these participants described their experience with Haskell as “beginner”, while the third described their experience as “intermediate”.

Five of the participants have the equivalent of (or are currently studying for) a bachelor’s degree in Computer Science. Four of these participants described their experience with Haskell as “intermediate”, while the fifth described their experience as “beginner”.

Two participants have the equivalent of (or are currently studying for) a postgraduate degree in Computer Science, both of whom described their experience with Haskell as “advanced”.

The final participant described their Computer Science qualifications as “other”, saying “Every day is a school day”, and their experience with Haskell as “intermediate”.

Fully tabulated results for all quantitative data can be found in Appendix B. A summary of the results is as follows:

5.3.1.2 Writing a Basic Haskell Program

Category	Average Ease of Task	Average Time Taken
A-Level	2	15 minutes
Bachelor’s	7.4	4 minutes
Postgraduate	10	less than a minute
Other	10	3 minutes
Beginner	2.67	13 minutes
Intermediate	7.5	5 minutes
Advanced	10	less than a minute
Overall	6.64	6 minutes

Table 5.1: The average results of the first task.

These results indicate show that the writing and executing a basic Haskell program through the website was straightforward for those with experience in Haskell, but quite difficult for those without. This indicates that the website is quite intuitive to use, but that it could provide more documentation for Haskell itself, rather than giving a brief summary, and linking to external resources.

5.3.1.3 Producing a Simple Graphic Using the Reference Page

Category	Average Ease of Task	Average Quality of Docs	Average Time Taken
A-Level	2.33	3.33	11 minutes
Bachelor's	6	6.6	10 minutes
Postgraduate	7.5	7.5	8 minutes
Other	8	7	3 minutes
Beginner	3.33	5	13 minutes
Intermediate	5.83	5.83	8 minutes
Advanced	7.5	7.5	8 minutes
Overall	5.45	5.91	9 minutes

Table 5.2: The average results of the second task.

These results demonstrate that those with more experience both in the field of Computer Science and in Haskell found the documentation to be of reasonable high quality, while those with less experience did not. This could be a result of participants with less experience having less experience with reading documentation in general, but would also suggest that, as before, the website could benefit from more detailed documentation for beginners.

As to be expected, more experienced users once again found this task easier than those with less experience. Interestingly though, users with less experience found this task easier than the previous task, whereas more experienced users found this task harder. As users should have learnt how to write basic Haskell programs in the previous task, they should no longer be at a disadvantage in this task. This would suggest, that once users have worked out how to write a basic Haskell program, they are able to use the reference page to produce simple graphics with relative ease, albeit still with some difficulty.

5.3.1.4 Altering a Pre-existing Program to Produce a Different Image

Category	Average Ease of Task	Average Time Taken
A-Level	9.67	2 minutes
Bachelor's	9	3 minutes
Postgraduate	5.33	6 minutes
Other	10	3 minutes
Beginner	9.67	2 minutes
Intermediate	9.17	3 minutes
Advanced	5.33	6 minutes
Overall	9.09	3 minutes

Table 5.3: The average results of the third task.

These results show that altering a pre-existing program to produce a different image was easy for most users, regardless of their experience. This suggests that the graphics

library is easy to grasp, if provided with an example to work from. Feedback from the postgraduate participant who gave a lower score for this task suggests that they accomplished the task to a degree, but were not happy with their result, as it did not match the provided example closely enough.

5.3.1.5 Altering a Pre-existing Program to Produce a Different Animation

Category	Average Ease of Task	Average Time Taken
A-Level	0.67	13 minutes
Bachelor's	7	6 minutes
Postgraduate	7.5	6 minutes
Other	10	3 minutes
Beginner	3	15 minutes
Intermediate	6.33	5 minutes
Advanced	7.5	8 minutes
Overall	5.63	6 minutes

Table 5.4: The average results of the fourth task.

The results for this task indicate a substantial leap in complexity from the previous task. A-Level and beginner participants in particular found this task to be difficult, with one A-Level participant taking over 21 minutes to complete the task. This suggests that the website could benefit from more detailed documentation on how to move from images to animations. For more experienced participants, this task was still more difficult than the previous task, but not to the same extent. These participants were able to complete the task in a reasonable amount of time, and with a reasonable level of ease.

5.3.1.6 Overall Ratings

Category	Average Rating
A-Level	3
Bachelor's	4.4
Postgraduate	4.5
Other	5
Beginner	3.33
Intermediate	4.33
Advanced	4.5
Overall	4.09

Table 5.5: The average overall ratings of the website, out of 5.

The overall ratings for the website were generally positive, with the majority of participants rating the website as above average. The website was rated most highly by those with more experience, and least highly by those with less experience. This once again indicates that the website provides a better user experience for users with at least bachelor's level experience in Computer Science, and some experience with Haskell.

5.3.1.7 Feedback

The feedback received from participants was generally positive, however there were a few areas for improvement that were highlighted. Several participants indicated initial difficulties due to their lack of experience with Haskell, but found the following tasks to be easier, once they had a better understanding of the language. A few participants found the reference page lacked clear instructions for how to render a shape once it had been defined. The documentation has since been updated to make this clearer.

Multiple users requested some way to view the reference page while editing their program, as they found it irritating to switch between the two pages. This has been implemented by embedding the reference page in the editor page. Users can switch between the canvas and the reference page by clicking on the tabs at the top of the right column.

Participants praised the website's clean design, ease-of-use and speed. Many spoke highly of the graphics library, and the example image provided in the reference page, which helped them to understand how to use the library. The examples on the home-page were described by multiple participants as "interesting", "useful", "informative", "entertaining" and even "incredible" by one participant.

5.3.2 Other Feedback from Users

Alongside the survey responses, feedback was also received from users on the Haskell subreddit. This feedback was extremely positive, with one user saying,

> This an important step forward in the production of educational artefacts for Haskell. (u/mlitchard)

and another saying of the reference page,

> This is the level of friendliness in API docs that would get coders to come back to Haskell. (u/CubOfJudahsLion)

This feedback is greatly encouraging, and suggests that users could find the website to be a useful tool for learning Haskell.

Several users drew comparisons to other Haskell graphics tools, including Diagrams and Haskell for Mac. One user compared this project to CodeWorld, which led to its creator, Chris Smith, commenting on the post. His assessment of the project was as follows:

- As far as overall goal, it looks like this is staking out a path that's solidly and consistently using simple but normal Haskell. This is different from CodeWorld, which offers both an educational dialect that's aggressively monomorphized, uncurried, and otherwise removes any overloading type features that cause problems for beginners, but then also offers a Haskell mode with a more conventional Haskell API.
- In terms of implementation, this project seems to run code on the server and stream frames of drawing instructions back to the client

> to be interpreted in TypeScript. CodeWorld, by contrast, compiles Haskell code into JavaScript that runs in the web browser directly.
>
> - The API here is a bit less declarative in flavor than CodeWorld's. For instance, while CodeWorld works very hard to make its `Picture` type denotationally equivalent to `Point -> Color`, the analogous `Shape` type here appears to be rather more complex, including explicit notions of path, stroke, fill, etc. that mirror JavaScript's canvas API. That does give it some more flexibility, but at the cost of a more complex abstraction.
> - There does not appear to be an API for interactive or stateful programs here. (Not surprising, since it's a much newer project).
>
> (u/cdsmith (Chris Smith))

This feedback is particularly useful, as it provides a comparison between this project and CodeWorld from the creator of CodeWorld himself, and highlights some key differences between the two projects.

Chapter 6

Conclusions

Overall, the project has been a success. The website is functional, and the graphics library is capable of producing a wide range of images and animations. The website works well on a range of devices, and web browsers, is easy to use, and provides a good user experience. The feedback from users has been positive, and the website has been well-received by members of the Haskell community.

6.1 Future Work

While the system successfully achieves its goals, there is always room for improvement. A number of potential expansions could be made to the system, both in the graphics library and the website. Such additions could allow users to make better graphics, and provide an even better user experience.

6.1.1 The Graphics Library

Firstly, the library could be expanded to include more shapes and transformations, such as shearing and reflecting, allowing for more complex graphics to be created more easily. Secondly, the library could venture into functional reactive programming (Elliott & Hudak, 1997), allowing for more complex animations, which could be controlled by the user while the animation is running. This could also prove beneficial for introducing livelits to the library, allowing users to see truly real-time changes to their programs as they write them. Interactive programs were considered too complex for the scope of this project, but could be an extremely powerful addition to both the library and the website.

6.1.2 The Website

Modern IDEs provide a plethora of features to help users write code more efficiently. While the website's editor provides syntax highlighting, it could be expanded to include more features, such as inline error checking and intellisense. These features would allow users to find bugs before executing their programs, and make it easier to write their programs, without needing to refer to the documentation as frequently.

While the system currently allows users to save their programs to a database, it could prove beneficial to allow users to connect their GitHub accounts, and save their programs to a GitHub repository. This would allow users to easily share their programs with others, while also providing incremental backups for their work. Additionally, GitHub integration could go further, allowing users with access to GitHub Copilot to use its suggestions directly in the editor.

Bibliography

Berners-Lee, S. T. (1990a). *Information management: A proposal*. https://cds.cern.ch/record/369245/files/dd-89-001.pdf

Berners-Lee, S. T. (1990b). *The worldwideweb browser*. https://www.w3.org/People/Berners-Lee/WorldWideWeb.html

Braun, F. (1897). Ueber ein verfahren zur demonstration und zum studium des zeitlichen verlaufes variabler ströme [on a process for the display and study of the course in time of variable currents]. *Annalen der Physik und Chemie*, *60*.

Carlson, W. (2003). *A critical history of computer graphics and animation*. https://web.archive.org/web/20070405181508/http://accad.osu.edu/%7Ewaynec/history/lesson2.html

Church, A. (1936). A formulation of the simple theory of types. *Journal of Symbolic Logic*, *5*. https://www.cambridge.org/core/journals/journal-of-symbolic-logic/article/abs/formulation-of-the-simple-theory-of-types/85B3666C7DD81A4F66966A399364B44B

Curry, H. B., & Feys, R. (1958). Combinatory logic. *North-Holland Publishing Company*.

Díaz, D. (2016). *Processing [haskell]*. https://hackage.haskell.org/package/processing

ECMA TC39 committee. (2024). *Ecmascript® 2024 language specification*. https://ecma-international.org/publications-and-standards/standards/ecma-262/

Elliott, C., & Hudak, P. (1997). Functional reactive animation. *International Conference on Functional Programming*. http://conal.net/papers/icfp97/

Hixie, I. (2004). *Extending html*. http://ln.hixie.ch/?start=1089635050&count=1

Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, *21*. https://dl.acm.org/doi/pdf/10.1145/72551.72554

Hudak, P., Hughes, J., Jones, S. P., & Wadler, P. (2007). A history of haskell: Being lazy with class. *In Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL-III*, 1–55. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/history.pdf

Johansson, T. (2007). *Taking the canvas to another dimension*. https://web.archive.org/web/20071117170113/http://my.opera.com/timjoh/blog/2007/11/13/taking-the-canvas-to-another-dimension

Kholomiov, A. (2016). *Processing for haskell*. https://hackage.haskell.org/package/processing-for-haskell

Khronos Group. (2011). *Final webgl 1.0 specification*. https://www.khronos.org/news/press/khronos-releases-final-webgl-1.0-specification

Krause, R. (2023, July 23). *Using imagery in visual design*. https://www.nngroup.com/articles/imagery-in-visual-design/

Lie, H. W. (1996). *Cascading html style sheets — a proposal*. https://www.w3.org/People/howcome/p/cascade.html

Lippmeier, B. (2022). *Gloss: Painless 2d vector graphics, animations and simulations*. https://hackage.haskell.org/package/gloss

Marlow, S. (2010). *Haskell 2010 language report*. https://www.haskell.org/definition/haskell2010.pdf

McCarthy, L. (2013a). *P5.js*. https://p5js.org/about/

McCarthy, L. (2013b). *P5.js reference — webgl*. https://p5js.org/reference/p5/WEBGL/

Mozilla Developer Connection. (2011). *Htmlcanvaselement*. https://web.archive.org/web/20110604062413/https://developer.mozilla.org/en/DOM/HTMLCanvasElement

Munshi, A., & Leech, J. (2010). *Opengl es common profile specification version 2.0.25 (full specification)*. https://registry.khronos.org/OpenGL/specs/es/2.0/es_full_spec_2.0.pdf

Noll, A. M. (1971). Scanned-display computer graphics. *Communications of the ACM*, *14*. https://dl.acm.org/doi/10.1145/362566.362567

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (1992). *Numerical recipes in fortran 77: The art of scientific computing*.

Processing Foundation. (2018). A modern prometheus. *Medium*. https://medium.com/processing-foundation/a-modern-prometheus-59aed94abe85

Rutter, R. (2005, December 8). *Elements of typographic style applied to the web*. https://webtypography.net/2.1.2

Segal, M., & Akeley, K. (1997). *The opengl graphics system: A specification (version 1.1)*. https://registry.khronos.org/OpenGL/specs/gl/glspec11.pdf

Smith, C. (2014). *Codeworld*. https://github.com/google/codeworld

StackOverflow. (2024). *Stackoverflow developer survey 2024*. https://survey.stackoverflow.co/2024/

StatCounter. (2025). *Browser market share worldwide*. https://gs.statcounter.com/browser-market-share

Sutherland, I. (1963). *Sketchpad: A man-machine graphical communication system* [Ph.D. Thesis]. Massachusetts Institute of Technology. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-574.pdf

Tiobe. (2025). *Tiobe index*. https://www.tiobe.com/tiobe-index/

Turing, A. (1937). Computability and λ-definability. *Journal of Symbolic Logic*, *2*. https://www.jstor.org/stable/2268280

Vukićević, V. (2007). *Canvas 3d: Gl power, web-style*. https://web.archive.org/web/20110717224855/http://blog.vlad1.com/2007/11/26/canvas-3d-gl-power-web-style/

Wadler, P. (1993). Monads for functional programming. *Program Design Calculi*, 233–264. https://link.springer.com/chapter/10.1007/978-3-662-02880-3_8

Wadler, P., & Blott, S. (1988). How to make ad-hoc polymorphism less ad hoc. https://www.researchgate.net/publication/2710954_How_to_Make_Ad-Hoc_Polymorphism_Less_Ad_Hoc

World Wide Web Consortium. (2011). *A vocabulary and associated apis for html and xhtml*. https://www.w3.org/TR/2011/WD-html5-20110405/

World Wide Web Consortium. (2014). *Secret origin of svg*. https://www.w3.org/Graphics/SVG/WG/wiki/Secret_Origin_of_SVG

World Wide Web Consortium. (2022). *Css color module level 3*. https://www.w3.org/TR/css-color-3/#colorunits

Worth, C., & Packard, K. (2005). *Cairo*. https://www.cairographics.org

Yorgey, B. (2023). *Diagrams: Embedded domain-specific language for declarative vector graphics*. https://hackage.haskell.org/package/diagrams

Appendix A

Participants' Information Sheet & Consent Form

The following pages contain the participants' information sheet and consent form, verbatim as presented to participants.

Participant Information Sheet

Project title:	A Graphical Playground for Haskell
Principal investigator:	Professor Philip Wadler
Researcher collecting data:	Oliver Jones

This study was certified according to the Informatics Research Ethics Process, reference number 102908. Please take time to read the following information carefully. You should keep this page for your records.

Who are the researchers?

Oliver Jones and Professor Philip Wadler

What is the purpose of the study?

To assess the usability of the system, allowing me to evaluate how successful the project is.

Why have I been asked to take part?

The research target group is people with varying backgrounds in computer science and functional programming.

Do I have to take part?

No – participation in this study is entirely up to you. You can withdraw from the study at any time, without giving a reason. Your rights will not be affected. If you wish to withdraw, contact the PI. We will stop using your data in any publications or presentations submitted after you have withdrawn consent. However, we will keep copies of your original consent, and of your withdrawal request.

What will happen if I decide to take part?

You will be asked to complete a series of tasks using the system. For each task, you will be asked to answer a few questions, rating various aspects of the system's usability. This will be done remotely, with questions being answered via an online survey.

Are there any risks associated with taking part?

There are no significant risks associated with participation.

Are there any benefits associated with taking part?

No

What will happen to the results of this study?

The results of this study may be summarised in published articles, reports and presentations. Quotes or key findings will be anonymized: We will remove any information that could, in our assessment, allow anyone to identify you. With your consent, information can also be used for future research. Your data may be archived for a minimum of 2 years.

Data protection and confidentiality.

Your data will be processed in accordance with Data Protection Law. All information collected about you will be kept strictly confidential. Your data will be referred to by a unique participant number rather than by name. Your data will only be viewed by Oliver Jones.

All electronic data will be stored on a password-protected encrypted computer, on the School of Informatics' secure file servers, or on the University's secure encrypted cloud storage services (DataShare, ownCloud, or Sharepoint) and all paper records will be stored in a locked filing cabinet in the PI's office. Your consent information will be kept separately from your responses in order to minimise risk.

What are my data protection rights?

The University of Edinburgh is a Data Controller for the information you provide. You have the right to access information held about you. Your right of access can be exercised in accordance Data Protection Law. You also have other rights including rights of correction, erasure and objection. For more details, including the right to lodge a complaint with the Information Commissioner's Office, please visit www.ico.org.uk. Questions, comments and requests about your personal data can also be sent to the University Data Protection Officer at dpo@ed.ac.uk.
For general information about how we use your data, go to: edin.ac/privacy-research

Who can I contact?

If you have any further questions about the study, please contact the lead researcher, Oliver Jones (s2153980@ed.ac.uk).

If you wish to make a complaint about the study, please contact inf-ethics@inf.ed.ac.uk. When you contact us, please provide the study title and detail the nature of your complaint.

Updated information.

If the research project changes in any way, an updated Participant Information Sheet will be made available on http://web.inf.ed.ac.uk/infweb/research/study-updates.

Consent

By proceeding with the study, I agree to all of the following statements:

- I have read and understood the above information.
- I understand that my participation is voluntary, and I can withdraw at any time.
- I consent to my anonymised data being used in academic publications and presentations.
- I allow my data to be used in future ethically approved research.

  Take me to the survey

Appendix B

Responses to User Testing & Feedback Survey

The following pages contain the tabulated quantitative responses to the user testing and feedback survey.

Writing a Basic Haskell Program

Qualifications	Experience	Ease of Task	Time Taken
Bachelor's Degree	Intermediate	7	1-5 minutes
Bachelor's Degree	Intermediate	10	1-5 minutes
Postgraduate Degree	Advanced	10	less than a minute
A-Level or equivalent	Intermediate	2	11-15 minutes
A-Level or equivalent	Beginner	2	11-15 minutes
A-Level or equivalent	Beginner	2	16-20 minutes
Bachelor's Degree	Intermediate	8	1-5 minutes
Postgraduate Degree	Advanced	10	less than a minute
Other	Intermediate	10	1-5 minutes
Bachelor's Degree	Intermediate	8	1-5 minutes
Bachelor's Degree	Beginner	4	6-10 minutes

Table B.1: The results of the first task

Producing a Simple Graphic Using the Reference Page

Qualifications	Experience	Ease of Task	Quality of Docs	Time Taken
Bachelor's Degree	Intermediate	4	5	6-10 minutes
Bachelor's Degree	Intermediate	10	9	11-15 minutes
Postgraduate Degree	Advanced	6	6	6-10 minutes
A-Level or equivalent	Intermediate	2	3	6-10 minutes
A-Level or equivalent	Beginner	4	3	11-15 minutes
A-Level or equivalent	Beginner	1	4	11-15 minutes
Bachelor's Degree	Intermediate	5	3	6-10 minutes
Postgraduate Degree	Advanced	9	9	6-10 minutes
Other	Intermediate	8	7	1-5 minutes
Bachelor's Degree	Intermediate	6	8	6-10 minutes
Bachelor's Degree	Beginner	5	8	11-15 minutes

Table B.2: The results of the second task, including the quality of the documentation

Altering a Pre-existing Program to Produce a Different Image

Qualifications	Experience	Ease of Task	Time Taken
Bachelor's Degree	Intermediate	9	less than a minute
Bachelor's Degree	Intermediate	9	1-5 minutes
Postgraduate Degree	Advanced	6	6-10 minutes
A-Level or equivalent	Intermediate	10	1-5 minutes
A-Level or equivalent	Beginner	10	1-5 minutes
A-Level or equivalent	Beginner	9	less than a minute
Bachelor's Degree	Intermediate	8	1-5 minutes
Postgraduate Degree	Advanced	10	1-5 minutes
Other	Intermediate	10	1-5 minutes
Bachelor's Degree	Intermediate	9	1-5 minutes
Bachelor's Degree	Beginner	10	1-5 minutes

Table B.3: The results of the third task

Altering a Pre-existing Program to Produce a Different Animation

Qualifications	Experience	Ease of Task	Time Taken
Bachelor's Degree	Intermediate	8	1-5 minutes
Bachelor's Degree	Intermediate	9	6-10 minutes
Postgraduate Degree	Advanced	7	6-10 minutes
A-Level or equivalent	Intermediate	0	1-5 minutes
A-Level or equivalent	Beginner	0	21+ minutes
A-Level or equivalent	Beginner	2	11-15 minutes
Bachelor's Degree	Intermediate	3	6-10 minutes
Postgraduate Degree	Advanced	8	1-5 minutes
Other	Intermediate	10	1-5 minutes
Bachelor's Degree	Intermediate	8	1-5 minutes
Bachelor's Degree	Beginner	7	6-10 minutes

Table B.4: The results of the fourth task

Overall Ratings

Qualifications	Experience	Rating
Bachelor's Degree	Intermediate	5
Bachelor's Degree	Intermediate	4
Postgraduate Degree	Advanced	4
A-Level or equivalent	Intermediate	3
A-Level or equivalent	Beginner	3
A-Level or equivalent	Beginner	3
Bachelor's Degree	Intermediate	4
Postgraduate Degree	Advanced	5
Other	Intermediate	5
Bachelor's Degree	Intermediate	5
Bachelor's Degree	Beginner	4

Table B.5: The overall ratings of the website, out of 5

Appendix C

Graphics Library Code Listings

The following pages contain the full code listings for the graphics library.

Lib.hs

```haskell
module Lib (
    module Canvas,
    module Color,
    module Maths,
    module Shape,
    render,
) where

import Canvas
import Color
import Internal ()
import Maths
import Shape (
    Shape,
    arc,
    bezier2,
    bezier3,
    circle,
    ellipse,
    empty,
    fill,
    line,
    noFill,
    noStroke,
    pie,
    polygon,
    rect,
    regular,
    rotate,
    scale,
    segment,
    square,
    stroke,
    strokeWeight,
    translate,
    translateX,
    translateY,
    (&),
    (>>>),
 )

-- Prints the instructions to render to the canvas
render :: Canvas -> IO ()
```

```haskell
render canvas = do
    putStrLn $ "canvas(" ++ show canvas ++ ")"
    mapM_ (putStrLn . (\x -> "frame(" ++ show x ++ ")")) (_frames canvas)
    putStrLn "done()"
```

Internal.hs

```haskell
module Internal where

import Canvas (Canvas (..))
import Color (Color (..))
import Data.List (intercalate)
import Maths (Radians, Vector (..))
import Shape (
    Connection (..),
    Shape (Arc, Curve, Ellipse, Empty, Group, Line, Polygon, Rect),
    ShapeOptions (ShapeOptions),
    defaultAngle,
    defaultFill,
    defaultStroke,
    defaultStrokeWeight,
 )

-- Remove ".0" from the end of a float
removeFloat :: Float -> String
removeFloat float
    | float == fromIntegral (round float) = show (round float)
    | otherwise = show float

-- Convert a canvas to a JSON string (excluding the frames)
instance Show Canvas where
    show :: Canvas -> String
    show (Canvas w h r b _) = "{\"w\":" ++ show w ++ ",\"h\":" ++ show h ++ ",\"r\":"
     ++ show r ++ ",\"b\":" ++ show b ++ "}"

-- Convert a vector to a JSON string
instance Show Vector where
    show :: Vector -> String
    show (Vector x y) = "[" ++ removeFloat x ++ "," ++ removeFloat y ++ "]"

-- Convert a shape to a JSON string
instance Show Shape where
    show :: Shape -> String
    show Empty = "{}"
    show (Group shapes) =
        "["
            ++ (intercalate "," [show shape | shape <- shapes])
            ++ "]"
    show (Line length options) =
        "{\"t\":0,\"l\":"
            ++ removeFloat length
            ++ jsonOptionsNoFill options
            ++ "}"
    show (Ellipse horizontalAxis verticalAxis options) =
        "{\"t\":1,\"h\":"
            ++ removeFloat horizontalAxis
            ++ ",\"v\":"
            ++ removeFloat verticalAxis
            ++ jsonOptions options
            ++ "}"
    show (Rect width height options) =
        "{\"t\":2,\"w\":"
            ++ removeFloat width
            ++ ",\"h\":"
            ++ removeFloat height
            ++ jsonOptions options
            ++ "}"
    show (Polygon points options) =
```

```
        "{\"t\":3,\"v\":"
            ++ show points
            ++ jsonOptions options
            ++ "}"
    show (Curve points options) =
        "{\"t\":4,\"v\":"
            ++ show points
            ++ jsonOptions options
            ++ "}"
    show (Arc horizontalAxis verticalAxis startAngle endAngle connect options) =
        "{\"t\":5,\"h\":"
            ++ removeFloat horizontalAxis
            ++ ",\"v\":"
            ++ removeFloat verticalAxis
            ++ ",\"b\":"
            ++ removeFloat startAngle
            ++ ",\"e\":"
            ++ removeFloat endAngle
            ++ ",\"c\":"
            ++ jsonConnection connect
            ++ jsonOptions options
            ++ "}"

-- Helper functions for converting shapes to JSON
isEmpty :: Shape -> Bool
isEmpty Empty = True
isEmpty _ = False

jsonPos :: Vector -> String
jsonPos (Vector x y)
    | x == 0 && y == 0 = ""
    | otherwise = ",\"p\":" ++ show (Vector x y)

jsonAng :: Radians -> String
jsonAng a
    | a == defaultAngle = ""
    | otherwise = ",\"a\":" ++ removeFloat a

jsonFC :: Color -> String
jsonFC c
    | c == defaultFill = ""
    | otherwise = ",\"f\":" ++ show c

jsonS :: Color -> String
jsonS c
    | c == defaultStroke = ""
    | otherwise = ",\"s\":" ++ show c

jsonSW :: Float -> String
jsonSW w
    | w == defaultStrokeWeight = ""
    | otherwise = ",\"sw\":" ++ removeFloat w

jsonConnection :: Connection -> String
jsonConnection Open = "0"
jsonConnection Chord = "1"
jsonConnection Pie = "2"

jsonOptionsNoFill :: ShapeOptions -> String
jsonOptionsNoFill (ShapeOptions pos ang fc sc sw) =
    jsonPos pos
        ++ jsonAng ang
        ++ jsonS sc
        ++ jsonSW sw

jsonOptions :: ShapeOptions -> String
jsonOptions (ShapeOptions pos ang fc sc sw) =
    jsonPos pos
        ++ jsonAng ang
        ++ jsonFC fc
```

```
        ++ jsonS sc
        ++ jsonSW sw

-- Convert a color to a JSON string
instance Show Color where
    show :: Color -> String
    show (RGB r g b) =
        "\"rgb("
            ++ show r
            ++ ","
            ++ show g
            ++ ","
            ++ show b
            ++ ")\""
    show (RGBA r g b a) =
        "\"rgba("
            ++ show r
            ++ ","
            ++ show g
            ++ ","
            ++ show b
            ++ ","
            ++ removeFloat a
            ++ ")\""
    show (Hex ('#' : hex)) =
        "\"#"
            ++ hex
            ++ "\""
    show (Hex hex) =
        "\"#"
            ++ hex
            ++ "\""
    show (HSL h s l) =
        "\"hsl("
            ++ show h
            ++ ","
            ++ show s
            ++ "%,"
            ++ show l
            ++ "%)\""
    show (HSLA h s l a) =
        "\"hsla("
            ++ show h
            ++ ","
            ++ show s
            ++ "%,"
            ++ show l
            ++ "%,"
            ++ removeFloat a
            ++ ")\""
    show Transparent = "0"
    show AliceBlue = "1"
    show AntiqueWhite = "2"
    show Aqua = "3"
    show AquaMarine = "4"
    show Azure = "5"
    show Beige = "6"
    show Bisque = "7"
    show Black = "8"
    show BlanchedAlmond = "9"
    show Blue = "10"
    show BlueViolet = "11"
    show Brown = "12"
    show BurlyWood = "13"
    show CadetBlue = "14"
    show Chartreuse = "15"
    show Chocolate = "16"
    show Coral = "17"
    show CornflowerBlue = "18"
    show Cornsilk = "19"
```

```
    show Crimson = "20"
    show Cyan = "21"
    show DarkBlue = "22"
    show DarkCyan = "23"
    show DarkGoldenRod = "24"
    show DarkGray = "25"
    show DarkGreen = "26"
    show DarkGrey = "27"
    show DarkKhaki = "28"
    show DarkMagenta = "29"
    show DarkOliveGreen = "30"
    show DarkOrange = "31"
    show DarkOrchid = "32"
    show DarkRed = "33"
    show DarkSalmon = "34"
    show DarkSeaGreen = "35"
    show DarkSlateBlue = "36"
    show DarkSlateGray = "37"
    show DarkSlateGrey = "38"
    show DarkTurquoise = "39"
    show DarkViolet = "40"
    show DeepPink = "41"
    show DeepSkyBlue = "42"
    show DimGray = "43"
    show DimGrey = "44"
    show DodgerBlue = "45"
    show FireBrick = "46"
    show FloralWhite = "47"
    show ForestGreen = "48"
    show Fuchsia = "49"
    show Gainsboro = "50"
    show GhostWhite = "51"
    show Gold = "52"
    show GoldenRod = "53"
    show Gray = "54"
    show Green = "55"
    show GreenYellow = "56"
    show Grey = "57"
    show HoneyDew = "58"
    show HotPink = "59"
    show IndianRed = "60"
    show Indigo = "61"
    show Ivory = "62"
    show Khaki = "63"
    show Lavender = "64"
    show LavenderBlush = "65"
    show LawnGreen = "66"
    show LemonChiffon = "67"
    show LightBlue = "68"
    show LightCoral = "69"
    show LightCyan = "70"
    show LightGoldenRodYellow = "71"
    show LightGray = "72"
    show LightGreen = "73"
    show LightGrey = "74"
    show LightPink = "75"
    show LightSalmon = "76"
    show LightSeaGreen = "77"
    show LightSkyBlue = "78"
    show LightSlateGray = "79"
    show LightSlateGrey = "80"
    show LightSteelBlue = "81"
    show LightYellow = "82"
    show Lime = "83"
    show LimeGreen = "84"
    show Linen = "85"
    show Magenta = "86"
    show Maroon = "87"
    show MediumAquaMarine = "88"
    show MediumBlue = "89"
```

```
    show MediumOrchid = "90"
    show MediumPurple = "91"
    show MediumSeaGreen = "92"
    show MediumSlateBlue = "93"
    show MediumSpringGreen = "94"
    show MediumTurquoise = "95"
    show MediumVioletRed = "96"
    show MidnightBlue = "97"
    show MintCream = "98"
    show MistyRose = "99"
    show Moccasin = "100"
    show NavajoWhite = "101"
    show Navy = "102"
    show Oldlace = "103"
    show Olive = "104"
    show OliveDrab = "105"
    show Orange = "106"
    show OrangeRed = "107"
    show Orchid = "108"
    show PaleGoldenRod = "109"
    show PaleGreen = "110"
    show PaleTurquoise = "111"
    show PaleVioletRed = "112"
    show PapayaWhip = "113"
    show PeachPuff = "114"
    show Peru = "115"
    show Pink = "116"
    show Plum = "117"
    show PowderBlue = "118"
    show Purple = "119"
    show Red = "120"
    show RosyBrown = "121"
    show RoyalBlue = "122"
    show SaddleBrown = "123"
    show Salmon = "124"
    show SandyBrown = "125"
    show SeaGreen = "126"
    show SeaShell = "127"
    show Sienna = "128"
    show Silver = "129"
    show SkyBlue = "130"
    show SlateBlue = "131"
    show SlateGray = "132"
    show SlateGrey = "133"
    show Snow = "134"
    show SpringGreen = "135"
    show SteelBlue = "136"
    show Tan = "137"
    show Teal = "138"
    show Thistle = "139"
    show Tomato = "140"
    show Turquoise = "141"
    show Violet = "142"
    show Wheat = "143"
    show White = "144"
    show WhiteSmoke = "145"
    show Yellow = "146"
    show YellowGreen = "147"
```

Canvas.hs

```
module Canvas where

import Color (Color (Transparent))
import Maths (Length, Vector (Vector), arg, mag)
import Shape (
    Shape (Curve, Empty, Group, Line, Rect, _options),
    ShapeOptions (_angle, _position),
```

```
    group,
)

-- Data structure to represent a canvas
data Canvas = Canvas
    { _width :: Length
    , _height :: Length
    , _fps :: Int
    , _backgroundColor :: Color
    , _frames :: [Shape]
    }

-- Creates a new canvas, setting the size of canvas element
createCanvas :: Length -> Length -> Canvas
createCanvas width height = Canvas width height 24 Transparent []

-- Set the background color of the canvas
background :: Color -> Canvas -> Canvas
background color canvas = canvas{_backgroundColor = color}

-- Set the fps of the canvas
fps :: Int -> Canvas -> Canvas
fps fps canvas = canvas{_fps = fps}

-- Add a shape to the canvas
(<<<) :: Canvas -> Shape -> Canvas
(<<<) canvas shape = canvas{_frames = _frames canvas ++ [shape]}

-- Add a list of shapes to the canvas
(<<<:) :: Canvas -> [Shape] -> Canvas
(<<<:) canvas shapes = canvas{_frames = _frames canvas ++ shapes}

-- Set operator precedence
infixl 7 <<<
infixl 7 <<<:

-- This transformation requires the canvas as an input, so it is not included in the
    Shape module
center :: Canvas -> Shape -> Shape
center _ Empty = Empty
center canvas (Group shapes) = group shapes (center canvas)
center (Canvas w h _ _ _) line@(Line l opts) =
    line
        { _options =
            opts
                { _position =
                    Vector
                        ((w - l * cos (_angle opts)) / 2)
                        ((h - l * sin (_angle opts)) / 2)
                }
        }
center (Canvas w h _ _ _) rect@(Rect x y opts) =
    rect
        { _options =
            opts
                { _position =
                    Vector
                        (w / 2 - (x / 2 * cos (_angle opts) - y / 2 * sin (_angle
    opts)))
                        (h / 2 - (x / 2 * sin (_angle opts) + y / 2 * cos (_angle
    opts)))
                }
        }
center (Canvas w h _ _ _) curve@(Curve pts opts) =
    curve
        { _options =
            opts
                { _position =
                    Vector
```

```
                        ((w - mag (last pts) * cos (_angle opts + arg (last pts))) /
    2)
                        ((h - mag (last pts) * sin (_angle opts + arg (last pts))) /
    2)
                }
        }
center (Canvas w h _ _ _) shape =
    shape
        { _options =
            (_options shape)
                { _position = Vector (w / 2) (h / 2)
                }
        }
```

Shape.hs

```
module Shape where

import Color (Color (Black, Transparent))
import Data.List (intercalate)
import Maths (Length, Radians, Vector (..), (^*^), (^+^))

----------------
---- Shapes ----
----------------

-- Data structures to represent shapes
data Connection = Open | Chord | Pie
data ShapeOptions = ShapeOptions
    { _position :: Vector
    , _angle :: Radians
    , _fill :: Color
    , _stroke :: Color
    , _strokeWeight :: Float
    }
data Shape
    = Empty
    | Group [Shape]
    | Line
        { _length :: Length
        , _options :: ShapeOptions
        }
    | Ellipse
        { _horizontalAxis :: Length
        , _verticalAxis :: Length
        , _options :: ShapeOptions
        }
    | Rect
        { _width :: Length
        , _height :: Length
        , _options :: ShapeOptions
        }
    | Polygon
        { _points :: [Vector]
        , _options :: ShapeOptions
        }
    | Curve
        { _points :: [Vector]
        , _options :: ShapeOptions
        }
    | Arc
        { _horizontalAxis :: Length
        , _verticalAxis :: Length
        , _startAngle :: Radians
        , _endAngle :: Radians
        , _connect :: Connection
        , _options :: ShapeOptions
        }
```

```
53
54  -- Default arguments for shapes
55  defaultFill :: Color
56  defaultFill = Transparent
57
58  defaultStroke :: Color
59  defaultStroke = Black
60
61  defaultStrokeWeight :: Float
62  defaultStrokeWeight = 1
63
64  defaultPosition :: Vector
65  defaultPosition = Vector 0 0
66
67  defaultAngle :: Radians
68  defaultAngle = 0
69
70  defaultOptions :: ShapeOptions
71  defaultOptions = ShapeOptions defaultPosition defaultAngle defaultFill defaultStroke
        defaultStrokeWeight
72
73  -- Combine two shapes into a group
74  (&) :: Shape -> Shape -> Shape
75  (&) Empty right = right
76  (&) left Empty = left
77  (&) (Group left) (Group right) = Group (left ++ right)
78  (&) (Group left) right = Group (left ++ [right])
79  (&) left (Group right) = Group (left : right)
80  (&) left right = Group [left, right]
81
82  -- Identity shape
83  empty :: Shape
84  empty = Empty
85
86  -- Functions to create shapes
87  line :: Length -> Shape
88  line length = Line length defaultOptions
89
90  ellipse :: Length -> Length -> Shape
91  ellipse horizontalAxis verticalAxis = Ellipse horizontalAxis verticalAxis
        defaultOptions
92
93  circle :: Length -> Shape
94  circle radius = ellipse radius radius
95
96  rect :: Length -> Length -> Shape
97  rect width height = Rect width height defaultOptions
98
99  square :: Length -> Shape
100 square size = rect size size
101
102 polygon :: [Vector] -> Shape
103 polygon points = Polygon points defaultOptions
104
105 regular :: Int -> Length -> Shape
106 regular s r = polygon [Vector (x i) (y i) ^*^ r | i <- [1 .. s]]
107   where
108     angle = 2 * pi / fromIntegral s
109     x i = cos (angle * fromIntegral i)
110     y i = sin (angle * fromIntegral i)
111
112 bezier2 :: Vector -> Vector -> Shape
113 bezier2 controlPoint endPoint = Curve [controlPoint, endPoint] defaultOptions
114
115 bezier3 :: Vector -> Vector -> Vector -> Shape
116 bezier3 controlPoint1 controlPoint2 endPoint = Curve [controlPoint1, controlPoint2,
        endPoint] defaultOptions
117
118 arc :: Length -> Length -> Radians -> Radians -> Shape
```

```
119 arc horizontalAxis verticalAxis startAngle endAngle = Arc horizontalAxis verticalAxis
        startAngle endAngle Open defaultOptions
120
121 segment :: Length -> Length -> Radians -> Radians -> Shape
122 segment horizontalAxis verticalAxis startAngle endAngle = Arc horizontalAxis
        verticalAxis startAngle endAngle Chord defaultOptions
123
124 pie :: Length -> Length -> Radians -> Radians -> Shape
125 pie horizontalAxis verticalAxis startAngle endAngle = Arc horizontalAxis verticalAxis
        startAngle endAngle Pie defaultOptions
126
127 -------------------------
128 ---- Transformations ----
129 -------------------------
130
131 -- Chain transformations
132 (>>>) :: Shape -> (Shape -> Shape) -> Shape
133 (>>>) shape transform = transform shape
134
135 -- Apply a transformation to a group of shapes
136 group :: [Shape] -> (Shape -> Shape) -> Shape
137 group shapes transform = Group [transform shape | shape <- shapes]
138
139 -- Functions to manipulate shapes
140 fill :: Color -> Shape -> Shape
141 fill _ Empty = Empty
142 fill color (Group shapes) = group shapes (fill color)
143 fill color shape = shape{_options = (_options shape){_fill = color}}
144
145 stroke :: Color -> Shape -> Shape
146 stroke _ Empty = Empty
147 stroke color (Group shapes) = group shapes (stroke color)
148 stroke color shape = shape{_options = (_options shape){_stroke = color}}
149
150 strokeWeight :: Float -> Shape -> Shape
151 strokeWeight _ Empty = Empty
152 strokeWeight weight (Group shapes) = group shapes (strokeWeight weight)
153 strokeWeight weight shape = shape{_options = (_options shape){_strokeWeight = weight
        }}
154
155 translate :: Vector -> Shape -> Shape
156 translate _ Empty = Empty
157 translate vector (Group shapes) = group shapes (translate vector)
158 translate vector shape = shape{_options = (_options shape){_position = _position (
        _options shape) ^+^ vector}}
159
160 rotate :: Radians -> Shape -> Shape
161 rotate _ Empty = Empty
162 rotate angle (Group shapes) = group shapes (rotate angle)
163 rotate angle shape = shape{_options = (_options shape){_angle = _angle (_options
        shape) + angle}}
164
165 scale :: Float -> Shape -> Shape
166 scale _ Empty = Empty
167 scale scaleFactor (Group shapes) = group shapes (scale scaleFactor)
168 scale scaleFactor (Line length options) = Line (length * scaleFactor) options
169 scale scaleFactor (Ellipse horizontalAxis verticalAxis options) = Ellipse (
        horizontalAxis * scaleFactor) (verticalAxis * scaleFactor) options
170 scale scaleFactor (Rect width height options) = Rect (width * scaleFactor) (height *
        scaleFactor) options
171 scale scaleFactor (Polygon points options) = Polygon [Vector (x * scaleFactor) (y *
        scaleFactor) | Vector x y <- points] options
172 scale scaleFactor (Curve points options) = Curve [Vector (x * scaleFactor) (y *
        scaleFactor) | Vector x y <- points] options
173 scale scaleFactor (Arc horizontalAxis verticalAxis startAngle endAngle open options)
        = Arc (horizontalAxis * scaleFactor) (verticalAxis * scaleFactor) startAngle
        endAngle open options
174
175 -- Shorthand transformations
176 noStroke :: Shape -> Shape
```

```
noStroke = stroke Transparent

noFill :: Shape -> Shape
noFill = fill Transparent

translateX :: Float -> Shape -> Shape
translateX x = translate (Vector x 0)

translateY :: Float -> Shape -> Shape
translateY y = translate (Vector 0 y)

-- Set operator precedence
infixr 8 &
infixl 9 >>>
```

Maths.hs

```
module Maths where

import Data.Time.Clock.POSIX (getPOSIXTime)

type Length = Float

----------------
---- Angles ----
----------------

type Radians = Float
type Degrees = Float

radians :: Degrees -> Radians
radians degrees = degrees * pi / 180

degrees :: Radians -> Degrees
degrees radians = radians * 180 / pi

-----------------
---- Vectors ----
-----------------

-- Data structure to represent vectors
data Vector = Vector Float Float deriving (Eq)

-- Vector maths
(^+^) :: Vector -> Vector -> Vector
(^+^) (Vector x1 y1) (Vector x2 y2) = Vector (x1 + x2) (y1 + y2)

(^-^) :: Vector -> Vector -> Vector
(^-^) (Vector x1 y1) (Vector x2 y2) = Vector (x1 - x2) (y1 - y2)

(^*^) :: Vector -> Float -> Vector
(^*^) (Vector x y) s = Vector (x * s) (y * s)

(^/^) :: Vector -> Float -> Vector
(^/^) (Vector x y) s = Vector (x / s) (y / s)

infixl 6 ^+^
infixl 6 ^-^
infixl 7 ^*^
infixl 7 ^/^

mag :: Vector -> Length
mag (Vector x y) = sqrt (x ^ 2 + y ^ 2)

arg :: Vector -> Radians
arg (Vector x y) = atan2 y x

norm :: Vector -> Vector
```

```haskell
norm v = v ^/^ mag v

dot :: Vector -> Vector -> Float
dot (Vector x1 y1) (Vector x2 y2) = x1 * x2 + y1 * y2

----------------
---- Random ----
----------------

randoms :: Int -> [Double]
randoms seed = map fst (iterate (lcg . snd) (lcg seed))
  where
    lcg :: Int -> (Double, Int)
    lcg seed = (fromIntegral newSeed / fromIntegral (2 ^ 32), newSeed)
      where
        newSeed = (1664525 * seed + 1013904223) `mod` 2 ^ 32

seed :: IO Int
seed = do
    time <- getPOSIXTime
    return (floor (time * 1000000))
```

Color.hs

```haskell
module Color where

-- Data structure to represent CSS colors
data Color
    = RGB Int Int Int
    | RGBA Int Int Int Float
    | Hex String
    | HSL Int Int Int
    | HSLA Int Int Int Float
    | Transparent
    | AliceBlue
    | AntiqueWhite
    | Aqua
    | AquaMarine
    | Azure
    | Beige
    | Bisque
    | Black
    | BlanchedAlmond
    | Blue
    | BlueViolet
    | Brown
    | BurlyWood
    | CadetBlue
    | Chartreuse
    | Chocolate
    | Coral
    | CornflowerBlue
    | Cornsilk
    | Crimson
    | Cyan
    | DarkBlue
    | DarkCyan
    | DarkGoldenRod
    | DarkGray
    | DarkGreen
    | DarkGrey
    | DarkKhaki
    | DarkMagenta
    | DarkOliveGreen
    | DarkOrange
    | DarkOrchid
    | DarkRed
    | DarkSalmon
```

```
45      | DarkSeaGreen
46      | DarkSlateBlue
47      | DarkSlateGray
48      | DarkSlateGrey
49      | DarkTurquoise
50      | DarkViolet
51      | DeepPink
52      | DeepSkyBlue
53      | DimGray
54      | DimGrey
55      | DodgerBlue
56      | FireBrick
57      | FloralWhite
58      | ForestGreen
59      | Fuchsia
60      | Gainsboro
61      | GhostWhite
62      | Gold
63      | GoldenRod
64      | Gray
65      | Green
66      | GreenYellow
67      | Grey
68      | HoneyDew
69      | HotPink
70      | IndianRed
71      | Indigo
72      | Ivory
73      | Khaki
74      | Lavender
75      | LavenderBlush
76      | LawnGreen
77      | LemonChiffon
78      | LightBlue
79      | LightCoral
80      | LightCyan
81      | LightGoldenRodYellow
82      | LightGray
83      | LightGreen
84      | LightGrey
85      | LightPink
86      | LightSalmon
87      | LightSeaGreen
88      | LightSkyBlue
89      | LightSlateGray
90      | LightSlateGrey
91      | LightSteelBlue
92      | LightYellow
93      | Lime
94      | LimeGreen
95      | Linen
96      | Magenta
97      | Maroon
98      | MediumAquaMarine
99      | MediumBlue
100     | MediumOrchid
101     | MediumPurple
102     | MediumSeaGreen
103     | MediumSlateBlue
104     | MediumSpringGreen
105     | MediumTurquoise
106     | MediumVioletRed
107     | MidnightBlue
108     | MintCream
109     | MistyRose
110     | Moccasin
111     | NavajoWhite
112     | Navy
113     | Oldlace
114     | Olive
```

```
115     | OliveDrab
116     | Orange
117     | OrangeRed
118     | Orchid
119     | PaleGoldenRod
120     | PaleGreen
121     | PaleTurquoise
122     | PaleVioletRed
123     | PapayaWhip
124     | PeachPuff
125     | Peru
126     | Pink
127     | Plum
128     | PowderBlue
129     | Purple
130     | Red
131     | RosyBrown
132     | RoyalBlue
133     | SaddleBrown
134     | Salmon
135     | SandyBrown
136     | SeaGreen
137     | SeaShell
138     | Sienna
139     | Silver
140     | SkyBlue
141     | SlateBlue
142     | SlateGray
143     | SlateGrey
144     | Snow
145     | SpringGreen
146     | SteelBlue
147     | Tan
148     | Teal
149     | Thistle
150     | Tomato
151     | Turquoise
152     | Violet
153     | Wheat
154     | White
155     | WhiteSmoke
156     | Yellow
157     | YellowGreen
158     deriving (Eq)
```