Image Processing Software with a Node Editor Interface

Scott Barr
968113

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Bachelor of Science

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

August 13, 2021

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed *Scott Barr* (candidate)

Date August 13, 2021

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed *Scott Barr* (candidate)

Date August 13, 2021

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed *Scott Barr* (candidate)

Date August 13, 2021

I dedicate this document to the memory of my loving mother.

Abstract

With the number of images being uploaded to the internet every day, the need for image processing software accessible for a wider population becomes more and more necessary. Creating such a software with an innately intuitive interface allows for users to complete their goals more efficiently without the need for excessive learning through tutorials. Such an interface can be accomplished using visual programming which visually describes how a program will operate. This document outlines a design and implementation for building a framework to create intuitive applications with node-based interfaces, and gives an example image processing application extended from the node-editor framework. Throughout the whole project, a schedule was used to ensure that the systems were created on time and to a pre-defined plan to ensure that key features for both the node-editor framework and image processing applications were included. This project plan involved a risk mitigation strategy to ensure that no major problems would affect the project. The project resulted in success, achieving a node-editor framework that produces intuitive applications, and an image processing framework capable of matching professional software such as Blender's compositing tool.

Acknowledgements

I'd like to thank my parents, who throughout my 4 years at university always encouraged me, believed in me and made me strive to make them proud. I wouldn't have managed to achieve what I've done so far if it wasn't for you both. To my mother, who I promised that I would achieve a first class honours here at Swansea University, I wish you were here to see it, even though you told me that you already knew I would get it. I love you and miss you so much.

I'd also like to thank my brother who has always been a role model to me, you've always supported and cheered me on when I really needed it.

To James, I'd like to thank you for making me competitive and always try to beat you, I don't think I would have done as well as I have done so far without the competition.

Finally I'd like to thank my supervisor and tutor, Prof. Mark Jones who has given me great advice throughout my final 2 years of university.

Contents

Chapter 1

Introduction

In 2015, it was estimated that 657 billion images were uploaded to the internet every year [1]. With approximately 41.5% of individuals using the internet in 2015, growing to roughly 53.6% towards the end of 2019 [2], it would be sensible to assume that more photos are being posted through social media and other means than ever before. As a consequence, the need for image processing software such as Photoshop is becoming more and more prevalent.

Photoshop is globally recognised as one of the best image processing software, however it is not intuitive for new users. On opening the application, a user is introduced to hundreds of tools with thousands of different options for each, which can be simply overwhelming.

Node-editors are a type of visual programming, where a user can create nodes and connect outputs to inputs, resulting in a graph of nodes connected to each other all to perform a specific task desired by the user. Visual programming itself has been noted to be generally more user friendly and can make programming more accessible for different classes of inexperienced users, as they are generally described as “intuitive" [3]. This idea of being able to use a new technology without having to understand new or difficult concepts was a key motivation for this project.

With only 40 publicly available repositories available on GitHub for node-editors [4], it’s clear that the level of availability and variety in these frameworks are much lower than other areas of computer science, for example the number of publicly available repositories for “Game Engines" is 3,228 [5]. Out of these 40 node-editor frameworks, only one is written in Java and it has very little documentation.

This project presents a solution to being able to design and develop intuitive Java applications featuring node-based interfaces by providing a simple yet functional framework. The framework has been designed to allow for developers to quickly produce these intuitive applications in an efficient way and can be extended to create interfaces for almost any problem domain.

To this extent, the framework has been used to develop an image processing software capable of being a substitute to current image editing applications, yet still being intuitive and easy to understand due to the nature of the node-based interface.

1.1 Motivations

Due to the increasing number of users on the internet and the large numbers of images being uploaded every day, the global desire for image processing software also gets larger. With professional software such as Photoshop requiring a substantial amount of experience and knowledge of image processing, it's unreasonable to assume the average user on the internet has the time and effort to put into learning such technology for something that might be a small hobby of sharing photos with their friends.

This project was designed to allow users to quickly and intuitively edit their photographs in a straightforward and logical manner, by giving the user a graphical interface that is designed for linear progression towards the end result that they desire.

During the initial phase of the project, it was noted that there were not many options given to developers who wanted to create Java applications with node-based interfaces, therefore the project aimed to create such a framework to allow developers to create many different node-based applications.

1.1.1 Objective

This project had 3 main aims:

- To produce a functional node-based interface framework for Java, which can be extended by anybody for a wide array of various applications.

- To produce a complete Image processing application with a large range of processing techniques and algorithms.

- To design the Image processing software in such a way that makes it extremely intuitive for new users, while maintaining a complexity that makes the software viable for experts who know what they are doing.

1.2 Overview

The rest of this introduction includes a brief overview of the remaining chapters, the key contributions produced, followed by similar work relating to the project. Chapter 2 covers design aspects of the node-editor framework and image processing application, giving insight into a wide range of features and their functions. Chapter 3 covers the implementation of the design for both the node-editor framework and the image processing application, this chapter gives a technical explanation as to how the features were implemented. In Chapter 4 an overview of the schedule used throughout the project is given and a risk-mitigation strategy outlined. Chapter 5 presents the achievements resulting from the project, comparing the image processing application to other well known software such as Blender. The document is then wrapped up in Chapter 6 accompanied with considerations for future work and development.

1.3 Contributions

The main contributions of this work can be seen as follows:

- **A framework for creating node-based applications in Java**

  This framework created in Java allows for quick and rapid development of new applications, not limited to any problem domain. That is it is suitable to be used for almost any problem.

- **A node-based image processing software**

  This software produced was an extension of the aforementioned node-based application framework, and allows users to process an image linearly until the desired result has been achieved.

1.4 Related Work

Blender is an open-source computer graphics software, primarily designed for users to create 3D models and animations. The software includes many tools, including an image compositor which consists of a node editor with multiple types of nodes for different image processing algorithms [6]. *Blender* is recognised as an excellent piece of software for the tasks it has been designed for. One of the tools available in *Blender* is the compositor, which allows a user to use a node-based editor to enhance their images or movies. The project resulted in a piece of software similar to the ability of the Blender compositor, without all of the extra tools that come with Blender, such as the 3D modelling and animation tools. The node-editor that comes with Blender is specific to Blender, which is different to the aims of the project where a complete framework has been provided.

Rete.js is a modular node-based visual programming framework created almost entirely with TypeScript [7]. The framework itself includes all of the requirements a developer needs to create an application with a node-based visual programming interface. While *Rete.js* is a brilliant framework, it is limited to applications being built solely for the web. JavaScript is an extremely popular language, however it also comes with a range of performance issues [8]. This project, rather than producing a framework limited to web applications, has produced a framework that can be integrated into stand-alone applications for all domains. While Java is also not recognised as the highest performing language compared to `C` or `C++`, Java has a 16.79% share of the language popularity, compared to `C/C++` which comes in at 5.78% [9].

In image processing, there are numerous types of algorithms, ranging from point operators which perform a function on each specific pixel of an image, to convolution matrix calculations which perform a function on a pixel while taking the surrounding pixels into account. The majority of the various algorithms incorporated into this image processing part of the project were obtained from the book “Fundamentals of Digital Image Processing : A Practical Approach with Examples in Matlab" [10]. Some of the main algorithms which have been implemented include, thresholding, contrast stretching, Gaussian filtering, Median Filtering, Edge Detection and Dithering.

The project has not only included these algorithms but also variations as well, for example the edge detection node includes an implementation of the Sobel algorithm, and also Roberts Cross [11].

There has been a variety of visual programming systems that have been created throughout the years, with one such system being Smith's Alternate Reality Kit [12], where key features include a Hand which is the user's primary means of interacting with the system. The user can use this hand to interact with various other elements, such as buttons and sliders which can be used to change values associated with certain objects. This allows for objects to influence each other if they are connected.

Another noteworthy system was developed for musicians to allow a user to create a digital synthesiser by drawing block diagrams [13]. This system is included with multiple block elements which all have different functions. Some of these blocks are for simple input, for example the *MIDI input* which allows the system to receive and transmit keyboard data. The *Frequency Modulation* object allows for 6 different inputs, and produces a single audio output. This system also includes an *Envelope Editor* which allows a user to interact with the object to create new envelopes, functions or waves. These features are key for our system where we require input nodes to allow a user to input images, fairly simple nodes which take inputs and produce a different image output, and also more complex nodes which allow users to interact with a component in the node itself. Similar visual programming systems to this include [14], [15], where users are given visual feedback showing each element or object and how it interacts with other components.

Block-based languages are another form of visual programming where users can place large function blocks, which can be nested inside another. Some examples of this are Scratch [16], Blockly [17] and Milo [18]. These block-based languages have been primarily used to teach concepts of programming without the requirement of understanding complex syntax which can be frustrating to new learners, or people without a programming background.

Chapter 2

Design

2.1 Node-Editor Framework

This section describes an overview of the Node-Editor Framework design and the functionality of key components and features for which the project has resulted in.

2.1.1 Editor

The core component for the node-editor framework is the editor that is provided to the developer. The Editor is the most important part of the framework as it is the window which allows a user to interact with the program and allow for node-based programming.

Right clicking on the editor window will display to the user a context menu which can be configured by the developer. This context menu will give a list of the available nodes. Clicking a listed node item will create a node of that type, and place it to where the mouse was clicked. The context menu also allows for submenus should the developer want to create sub nodes. An example of this can be seen in figure 2.1.

Since the framework is based around JavaFx and with the editor being the root of the node-based application, a cascading style sheet can be assigned to the editor which will cascade into all nodes and elements in the window. This allows a developer or user to change the styles of elements on the screen, such as changing colours of text and the backgrounds of nodes. In figure 2.3 we see two different user interfaces, one being dark backgrounds with light text, and the other being light backgrounds with dark text. This allows developers to provide style sheets for users, and also allows a user to be able to create their own style sheet should they wish to.

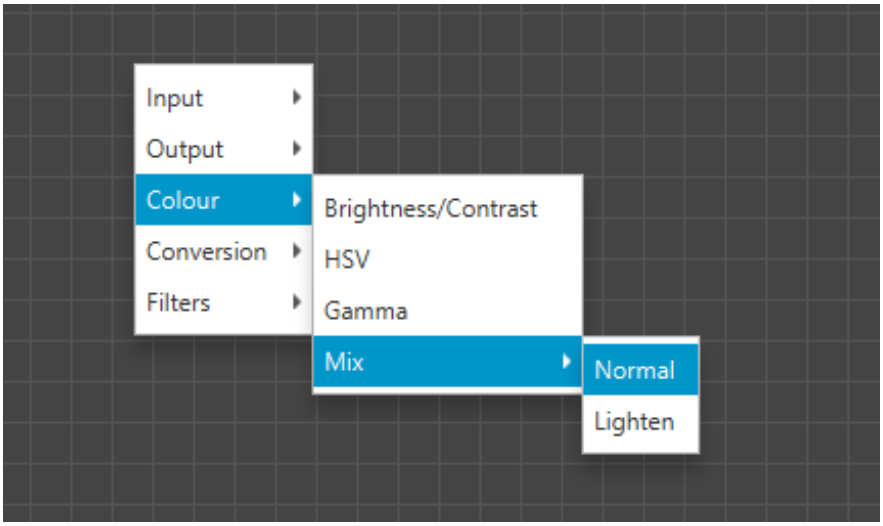
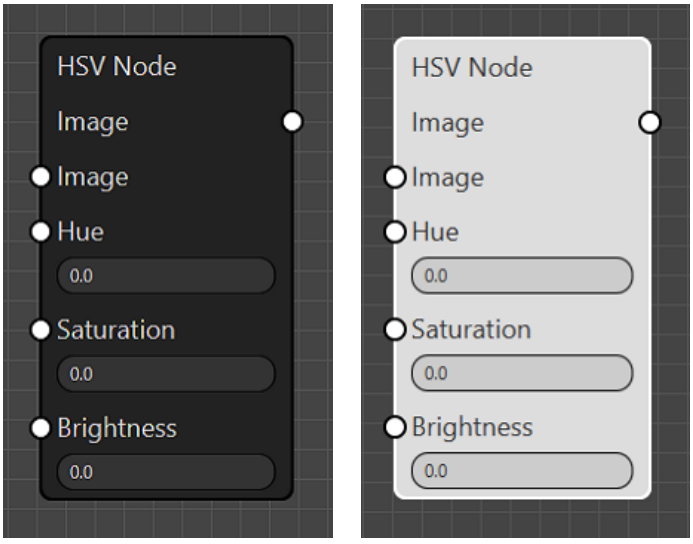


Figure 2.1: A context menu providing a list of image processing nodes including sub-menus.



A. Night-Mode style User Interface. B. Light-Mode style User Interface.

Figure 2.2: Two different cascading style sheets used to produce two visually different user interfaces. In 2.2A. we see a night-mode styled interface, while in 2.2B. we see a light-mode interface.

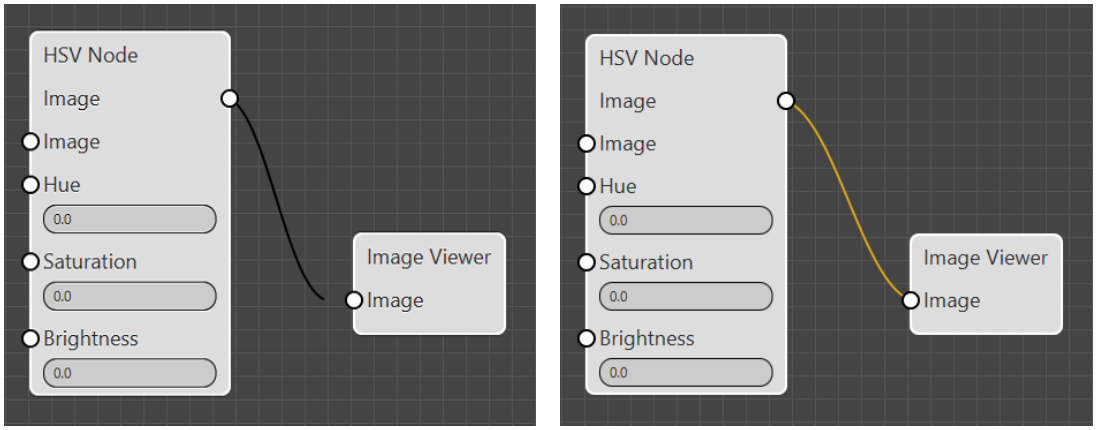
2.1.2 Nodes

A node is an element which is placed on the editor by the user and is they key component of any node-based application. A node is a visual element or block which can be customised in

any way the developer intends.

The framework allows a programmer to be able to develop a new node following the styling of the user interface extremely quickly. The developer only needs to provide the inputs, outputs and the functionality of the node, with the functionality being how the node uses the inputs to produce an output. This could be a simple addition node such as two integer inputs, an integer output, and the functionality adding the two inputs together and outputting it.

Node inputs and outputs all have a socket associated with them, an input would have a socket to allow data to be transferred to the node, whereas an output socket allows data to be transferred out of the node to an input of another node. The sockets can be clicked on and dragged which creates a line to demonstrate the connection. If the line is then dragged over a compatible socket of a different type, the line will glow yellow to show this. Sockets are only compatible with each other if they are opposing, for example input to an output or vice versa, on top of this, the socket must be compatible with the type of data from the socket too, an `Image` socket is not compatible with a `Double` socket.



A. Guide Line from an Output Socket.

B. Compatible sockets from an Image Output Socket to an Image Input Socket.

Figure 2.3: An example of dragging on a Socket to create a Guide Line to visually demonstrate the connection being made between two sockets. 2.3B. shows two compatible sockets through the guide line turning yellow.

An output socket of a node can be connected to multiple different input nodes, however an input node can only have one output that it is connected to. To this effect, clicking and dragging on

an output socket will create a new guide line and potentially a new connection if it is connected to a compatible socket therefore allowing multiple outgoing connections as seen in figure 2.4, however clicking and dragging on an input socket will delete any previous connection and then initiate a new guide line to be connected to an input.

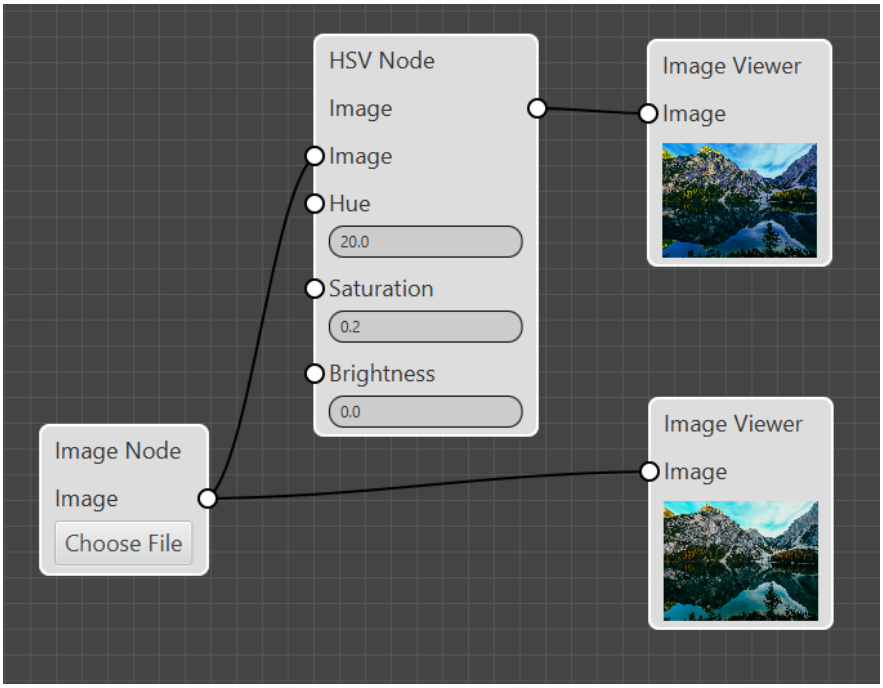


Figure 2.4: An example node layout demonstrating multiple output connections from an `Image` output socket.

2.1.3 Data Control

In certain cases, a user will need to provide an input to a node which will be used for the functionality of a node later on. In the example of an image processing application, a user would need to give an image to a node which can then be passed on to other nodes. This functionality is achieved using data controls which are components to a node that allow a user to interact and manipulate data.

A developer using this framework is able to create and extend new Data Control classes which can be linked to an input or output of a node. If an input has a data control component linked with it, any changes to the data control will cause that input value to change to the value of the data control.

2.1.4 Data Transfer

Data transfer within a network of nodes occurs under certain conditions. When a connection is created, the connection is either initialised as an incoming or outgoing connection. In the event that an incoming connection is made, the node will request the value of the output socket that it is connected with. When an outgoing connection is made, the output value of the socket is passed a long to the input socket of the node that it is connected with.

The other condition in which data transfer occurs is when the input of a node is updated. Once an input socket receives a new value, it will immediately trigger the node to calculate the new output based on the new values and pass these values along. This allows real time feedback of changes made to the nodes. In any case where an output value is requested but has not been initialised yet, no value will be passed on and therefore prevent null values being passed through the node network.

2.2 Image Processing Application

This section will give an overview of the design for the Image-Processing application and certain features of which it contains.

2.2.1 Image Input Node

The image input node is an extremely important part of the image processing application as it allows a user to interact with the node and select which image to be used. The node uses a file control component which is an extension of data control and allows the user to select a file from their system. This can be seen in figure 2.5.

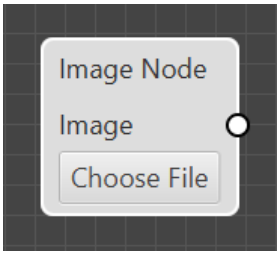
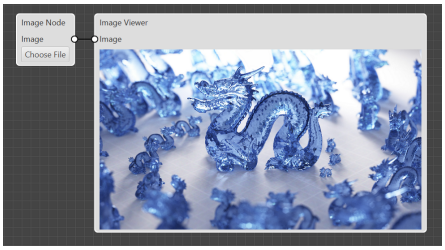


Figure 2.5: The Image input Node which allows a user to give an image file which can then be passed through to other nodes.

2.2.2 Image Viewer

An Image Viewer node is another important node for an image processing application, while it doesn't seem like it does too much while having only one input, it allows for a user to visualise the results of their image processing. A user can use this to see the impact of the image processing nodes on an image in real time, that is any time a user changes the input to or value of nodes it will update the image in the Image Viewer node immediately.

The Image Viewer node comes with a range of features such as being able to be resized when clicking and dragging on the bottom right hand corner of the image. Zooming on an image is also possible such that when the mouse wheel is scrolled upwards, the image will be zoomed in such that the pixel the mouse is hovering over remains in the same position relative to the viewport of the image viewer. The zooming feature works in the reverse way too, such that scrolling downwards with the mouse wheel will result in the image zooming back out. This node also allows users to pan an image, by clicking and dragging on the image, the pixel being dragged moves to the location of the mouse relative to the viewport.



A. Image Viewer with no zoom.

B. Image Viewer zoomed onto the head of a dragon.

Figure 2.6: A demonstration of the zooming feature that is part of the Image Viewer node.

Chapter 3

Implementation

3.1 Node-Editor Framework

3.1.1 Editor Context Menu

The Node Editor context menu is a small pop-up menu that appears when a user right clicks on the editor. This menu shows a list of nodes that a user would be able to create when clicking on the menu item. This meant that the context menu came with a non-trivial challenge of instantiating new `Node` objects for the user to interact with on demand. Initially the implementation of this context menu required a developer to add a new menu item to the context menu, and manually code the event handling when that menu item is clicked, resulting in a huge amount of repetitive code for each Node type being added to the menu. However this code was refined to allow a developer to provide only a String for the label of the menu item, and a Class which extends `Node`.

```
1 NodeMenu input = editor.menu.createSubMenu("Input");
2 input.addMenuItem("Image", ImageNode.class);
```

Listing 3.1: A simple example showing the simplicity of adding a sub-menu for input types, and a menu item which instantiates `ImageNode` objects.

This implementation is possible due to the fact that Extensions of the `Node` class require two parameters, being the x and y coordinate of the node to allow the node to be instantiated in the correct position. Knowing that any class inheriting the `Node` class will have only two parameters, Java allows us to obtain a constructor and use it to create a new instance by giving the required parameters. The implementation of this code can be seen in Listing 3.2.

```
1  public <T extends Node> void addMenuItem(String itemLabel, Class<T> type) {
2      MenuItem mItem = new MenuItem(itemLabel);
3
4      mItem.setOnAction(e -> {
5          try {
6              Point2D pos = parent.getPos();
7              T node = type.getDeclaredConstructor(double.class, double.class).
                   newInstance(pos.getX(), pos.getY());
8              parent.canvas.addNode(node);
9          } catch (Exception err) {
10             err.printStackTrace();
11         }
12     });
13
14   getItems().add(mItem);
15 }
```

Listing 3.2: A function which allows a developer to quickly add new menu items to a `NodeMenu` without having to repetitively write the event handler to instantiate the object for each node.

3.1.2 Dynamic Node Construction

When developing a new Node class for a node-based application, the framework allows such tasks to be done rapidly without requiring large amounts of repeated code. Developers can design and implement nodes of varying complexities, such as simply adding two numbers together or performing complex mathematical algorithms on various parameters. A node can be created by defining exactly which inputs exist, along with any outputs and then defining the function a node performs on such inputs to determine the output.

The `NodeInput` class is generically typed and used for defining the inputs of a Node, declaring a private member of `NodeInput<Double>` would represent an input to a node which accepts values of type `Double`. Similarly, a private member of `NodeOutput<Double>` represents an output for a node which produces values of type `Double`. The developer can choose the type for the input and can create new types to be used should they wish. Once the private members are declared, the developer can write the initialization code for those members inside by overriding the `initialize()` method. A snippet of code showing this can be seen in Listing 3.3.

```
1  private NodeInput<Double> num1;
2  private NodeInput<Double> num2;
3  private NodeOutput<Double> output;
4
5  @Override
6  protected void initialize() {
7    num1 = input(Double.class, "Value 1");
8    num2 = input(Double.class, "Value 2");
9    output = output(Double.class, "Sum");
10 }
11
12 @Override
13 public void function() {
14   Double v1 = num1.getValue();
15   Double v2 = num2.getValue();
16
17   output.setValue(v1 + v2);
18 }
```

Listing 3.3: Brief code snippet showing the member variables, the initialize function and the process function to create a new AddNode for adding two numbers together.

Various components which are added to nodes such as `NodeInput` and `NodeOutput` are automatically added to the visual hierarchy of the Node for JavaFX when the members are being initialized. The `Node` base class will dynamically resize itself to fit all components within the node, as well as vertically stack all components in the order they are initialized to allow for varying layouts for nodes.

3.1.3 Data Transfer Between Nodes

A node-based application can sometimes result in a complex network of nodes. A simple solution for passing data through a network of nodes would be to traverse the network from start to finish, every time a change is made, however this approach comes with a lot of cost as there shouldn't be any need for a node to recalculate an output based on inputs which haven't changed. Instead of this approach, a node should recalculate it's value only when an input changes.

JavaFX provides a great framework for triggering and handling events which was then used to create new event types, such as when an input receives an update to a value, or when an output gets updated to a new value. These such events are caught and handled. If an input update event is triggered, the node will attempt to perform a calculation with the new input.

This calculation could possibly change an output value, if this occurs, it will trigger an output update event which also triggers the node to pass this new value to any connected nodes from that output.

A node does not always have an initialised output value, in this case all inputs are required to calculate the value of the output and therefore cannot pass a value until these conditions are met. If a node has already computed an output value, when linking this node to a new input it's essential that this output value is immediately passed onto the input node. For this to work, another type of event was created to listen for when nodes are linked together. When a link is created, the node triggers the functionality to pass these values along if possible.

This implementation ensures that all descendants to a node which has an update to its inputs are also updated according to the new values of the parent nodes, up to the originally changed node. This implementation also means that a user does not need to constantly run new changes and allows for real time updates to any interaction a user makes to the node network and values.

3.2 Image Processing Application

3.2.1 Image Viewer

With the image viewer node having various features such as zooming, panning and resizing, it was important to make sure that the features performed in an intuitive way to the user. Originally, zooming was accomplished by scaling the width and height of the viewport, and maintaining the centre by calculating the difference in width and height and adding half to the start x and start y position of the viewport rectangle. While this technique worked well and was simple, it did not feel intuitive when using the feature. The zooming feature was then implemented by scaling the size of the viewport while maintaining the position of the pixel relative to the viewport. In simple terms this means the pixel that the mouse is hovering over stays in the same position that the mouse is at when the zooming takes place.

When a mouse scroll event is detected by JavaFX, the `ImageViewer` detects it and a `ScrollEvent` is passed into a function which determines the new location and size of the viewport. An implementation of this function can be seen at Listing A.1 in the appendix. This zoom function initially takes the change in Y direction for the mouse scroll wheel, with positive being an inwards zoom, and negative being an outwards zoom. With this delta value,

a scaling factor is calculated as seen in Listing A.1 line 3. The scaling factor allows us to determine the new size of the viewport by dividing the current size by the scaling factor. This new size is then constrained within a constant minimum size for the viewport, and the maximum size of the image to prevent a user from zooming outside of the image.

In line 11, the function `viewToImage` is called given the x and y coordinates of the mouse relative to the viewport. This function simply takes the mouse coordinates relative to the viewport and translates the x,y coordinates into a position of the pixel relative to the image. This can be seen in Listing A.2 and is used to find out the position relative to the image that the mouse is located at. The ratio of the mouse position relative to the viewport is then calculated and used to compute the minimum x and y positions for the viewport as seen in lines 16 and 17 within Listing A.1. These minimum values are also clamped between 0 and the new size of the viewport subtracted from the size of the image.

3.2.2 Image Processing Nodes

Due to the framework providing a simple solution to creating new types of nodes, image processing nodes are easy to implement. The main behaviour of image processing nodes involve taking an image as an input and performing an algorithm on it to alter the image in a certain way. In JavaFx, an image can be edited by firstly obtaining a `PixelReader` object from the image. The `PixelReader` class provides methods for reading pixel data from the image. With the `PixelReader` we can also instantiate a new `WritableImage` object with a given height and width which represents a new image that can be edited pixel by pixel using a `PixelWriter`.

All of the image processing functions require these 3 objects to be instantiated before being able to produce a new image and therefore to avoid repetitive code, an `ImageProcessor` class was implemented in order to initialise the pixel reader, writable image and pixel writer for any image processing functions to use. An `ImageProcessor` object is instantiated by providing a class that implements the public interface `ImageFunction` which requires all inherited classes to provide an implementation to an application function for the algorithm to be performed. An example implementation of a Brightness Image Function can be seen in Listing 3.4.

```
1  public class Brightness implements ImageFunction {
2
3      private double value = 0;
4
5      public Brightness(double value) {
6          this.value = value;
7      }
8
9      @Override
10     public void apply(ImageProcessor p) {
11
12         // Create a loop for all pixels in the x and y directions
13         for (int y = 0; y < p.height; y++) {
14             for (int x = 0; x < p.width; x++) {
15
16                 // Obtain the Color of the pixel at each x,y coordinate in the
                       image
17                 Color pixel = p.reader.getColor(x, y);
18
19                 // Calculate the red, green and blue components of the image by
                       adding a flat value to each component.
20                 // These values are then clamped to be constrained between 0 and
                       1.
21                 double r = ImageFunction.clamp(pixel.getRed() + value, 0, 1);
22                 double g = ImageFunction.clamp(pixel.getGreen() + value, 0, 1);
23                 double b = ImageFunction.clamp(pixel.getBlue() + value, 0, 1);
24
25                 // Finally set pixel at the same coordinates in the output image.
26                 p.writer.setColor(x, y, new Color(r, g, b, 1));
27             }
28         }
29     }
30 }
```

Listing 3.4: An implementation of the Brightness function to change the brightness of an image.

Chapter 4

Project Management

4.1 Time Management

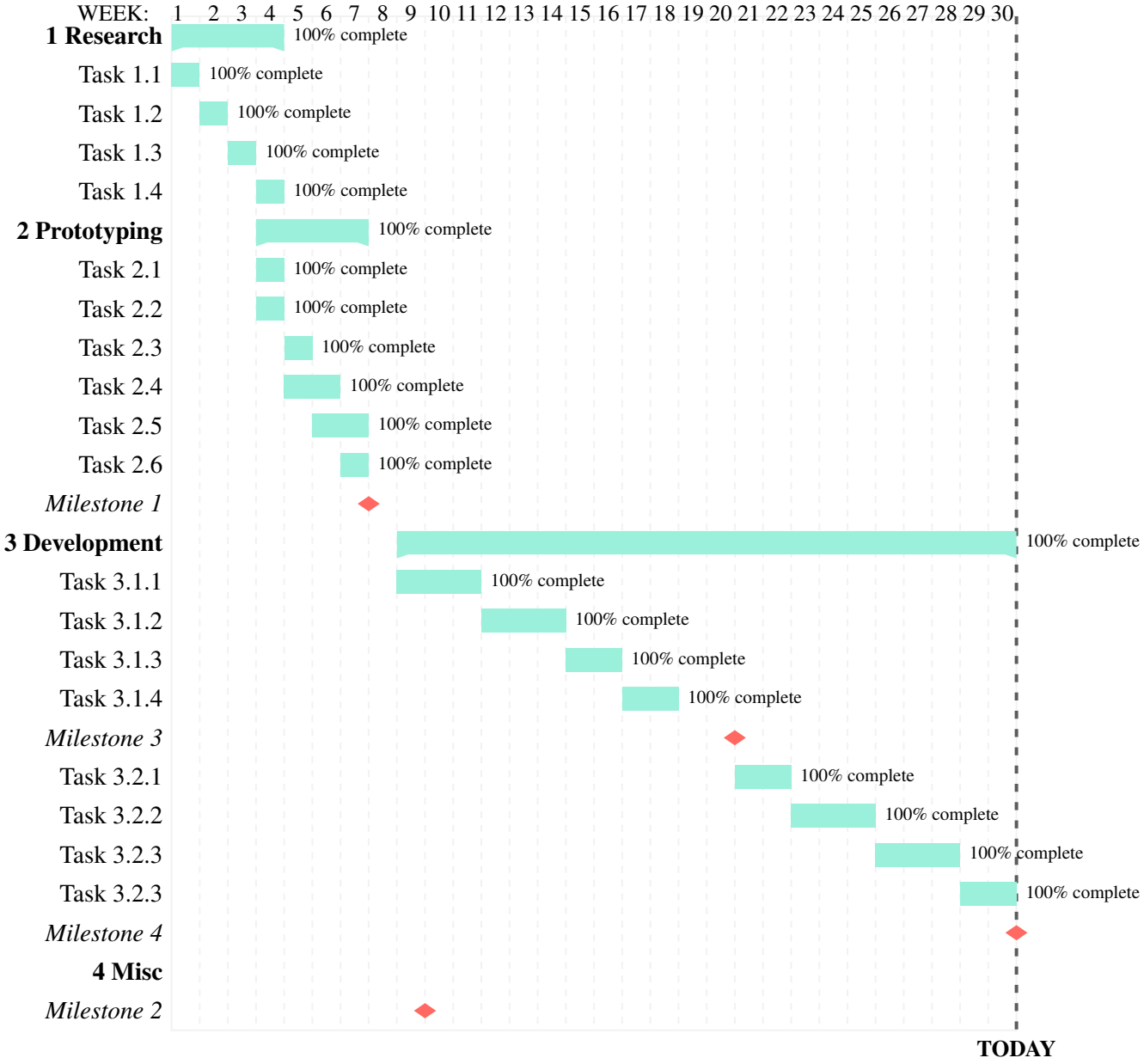
During the initial phase of the project, a reasonable amount of time was spent researching various technology and features that would be required for the project to be successful. This research phase included understanding what makes an efficient and usable framework, looking into the programming technologies needed which could allow for the project to continue. This stage resulted in the decision of using JavaFX for the entirety of the system, followed by this, further research into how to use JavaFX and, planning of the requirements of the node-editor framework was completed.

The second phase of the project was prototyping which allowed for the understanding of how to use JavaFX to be put into practical application. Many core features of the node-editor framework were implemented in an extremely basic way with no focus on quality of code. Features such as creating a node dynamically, moving objects, connecting objects and understanding how to style things were implemented during this phase. This phase took approximately 5 weeks and was included in the schedule to allow for much faster implementation during the development stage, with the idea that already having code that provides a function for what is required can be referred to when creating higher quality code.

The development stage of the project had been given a much larger time frame to ensure higher code quality and to allow for extra time to implement additional features if required. The development of the node-editor framework took approximately 10 weeks in total where key features that were implemented during the prototyping stage were implemented. The final portion of

development involved using the node-editor framework for creating an image processing application. Specialised nodes were created for various image processing techniques such as input nodes, mixing nodes and filter nodes.

4.1.1 Gantt Chart



Research - Understanding the requirements for the project

Task 1.1 Understanding Frameworks

Task 1.2 Programming Technologies to use

Task 1.3 Researching JavaFX and how to use it

Task 1.4 Planning out framework requirements

Prototyping - Developing very basic components necessary for full implementation

Task 2.1 Node Creation

Task 2.2 Node Manipulation - Moving, Scaling, Zooming etc...

Task 2.3 Node Connections - Connecting multiple nodes together

Task 2.4 Functionality for a node to be dynamically created from a list of inputs and outputs

Task 2.5 Node Graph for inputs and outputs

Task 2.6 User Interface for Node-Editor

Development

Task 3.1.1 Develop core functionality
Node movement, node linking, dynamic node creation, event handling

Task 3.1.2 Develop graph structure
Logic for passing data through node links

Task 3.1.3 User Interface styling

Task 3.2.1 Create Input/Output nodes

Task 3.2.2 Create Basic Function Nodes
E.g. point operation nodes such as a threshold node

Task 3.2.3 Implement filter nodes
E.g. Gaussian blur, Sobel Operator

Task 3.2.4 Mixing Nodes
E.g. Layering images together, mixing colours of two images etc...

Milestones

Milestone 1 Prototyping Completed

Milestone 2 Initial Document Completed

Milestone 3 Node-Editor Framework Finished

Milestone 4 Image Processing Application Finished

4.2 Risk Management

The project had certain risks which had to be considered to prevent failure of completion, some of these risks were mitigated to ensure the safety of the project. This section aims to give an idea of the risks with the project and for each risk, if it occurred and how the risk was mitigated.

4.2.1 Project Specific Risks

Risk	Probability	Impact	Occured?	Mitigation Strategy
Inability to handle specific image file types such as gifs due to a problem with reading the META data which is different to what is expected.	High	Medium	Yes	When selecting image files, the file selector was setup in a way to only accept certain file types e.g. .png, .jpg and other static image file types.
Multiple separate graph structures creating an issue due to more than one output being required	Medium	Medium	No	Using JavaFx events allowed for instant triggering of data transfer when changes occurred, meaning if a separate network changed, data would be passed through that network too.
A low level issue that occurs within JavaFx causes an error which prevents specific things from displaying to the user.	Low	Medium	No	Prototyping during initial stages of project allowed quick testing of important features to ensure they worked correctly. If an important feature wasn't compatible with the project an alternative would have had to have been found.

4.2.2 General Risks

Risk	Probability	Impact	Occurred?	Mitigation Strategy
Loss of code due to a hardware failure or	Low	High	No	Git was used to regularly upload code changes to Github which allowed the source code to be safe, with a version history to be able to backup and retrieve earlier versions of the code if needed.
Development falls behind schedule due to poor planning	Medium	Medium	No	The project had a clear path for the tasks which needed to be completed.
Can't acquire the skill set for certain features	Medium	Medium	No	Complex tasks were well thought out prior to development which allowed for solving a task more easily.
Failing to test frequently and introducing a bug which isn't caught, causing the project to be delayed.	Low	Medium	No	Frequent tests were done after making a change to ensure the system is still functioning as expected.
List of requirements and features increase during development	Low	Low	Yes	Any extra features needed were only added when extra time was found, for instance if a particular task was completed ahead of schedule, the remaining time could be used for extra features.

Chapter 5

Results

This chapter explains the resulting contributions from the project whilst comparing them to other implementations of similar works. To compare the node-editor framework, Rete.js [7] will be used as a comparison for key node-based programming functions, whereas for the image processing application, blender will be used as a reference for completing certain tasks.

5.1 Node-Editor Framework

In order to obtain any data on how effective the resulting framework is, another framework with a similar purpose is required for a reference. The most popular framework in the Github topic for Node-editors is by far, Rete.js which although is not a Java framework, it can still be used effectively to compare the effectiveness of both frameworks.

For both Rete.js and the project's framework, the key functionality of node-editor frameworks are implemented as you would expect, for example a user can click and drag on a node in order to move it to where the user desires, and being able to drag on a node's socket and create a new connection to another socket.

The resulting project's framework has a huge advantage over Rete.js when it comes to simplicity for creating new node classes. An implementation of an colour mixing node given in one of the example programs using Rete.js can be seen in Listing B.1. The implementation given in this example appears to be a lot more complex than what would be necessary in the project's framework. To compare the Rete.js implementation of a colour mixing node, listing 5.1 shows a much more condensed and simple implementation to achieve the same goal.

```
private NodeInput<Color> colour1;
private NodeInput<Color> colour2;
private NodeOutput<Color> output;

public ColourMixNode(double x, double y) {
  super(x, y, "Colour Mix Node");
}

@Override
protected void initialize() {
  colour1 = input(Color.class, "Colour 1", new ColorControl());
  colour2 = input(Color.class, "Colour 2", new ColorControl());
  output = output(Color.class, "Output");
}

@Override
public void function() {
  Color col1 = colour1.getValue();
  Color col2 = colour2.getValue();
  Color out = ImageMixFunction.blend(col1, col2, 0);

  output.setValue(out);
}
```

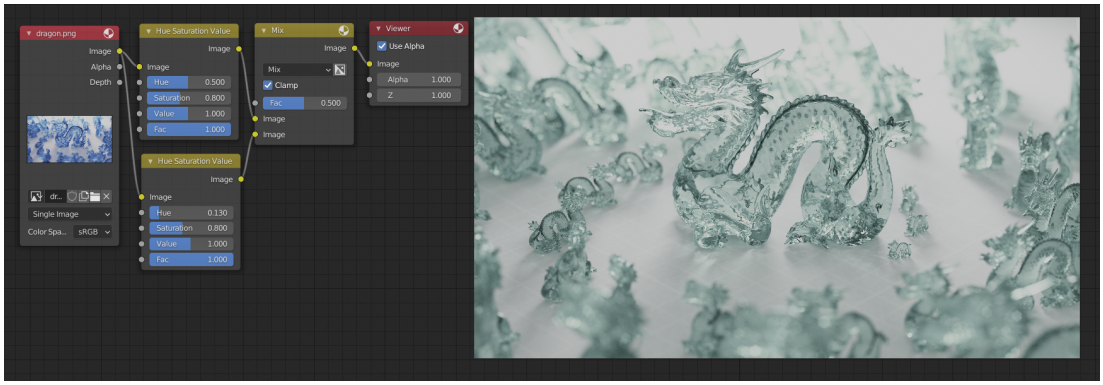
Listing 5.1: A simple implementation of a colour mixing node implemented in the project's node-editor framework.

5.2 Image Processing Application

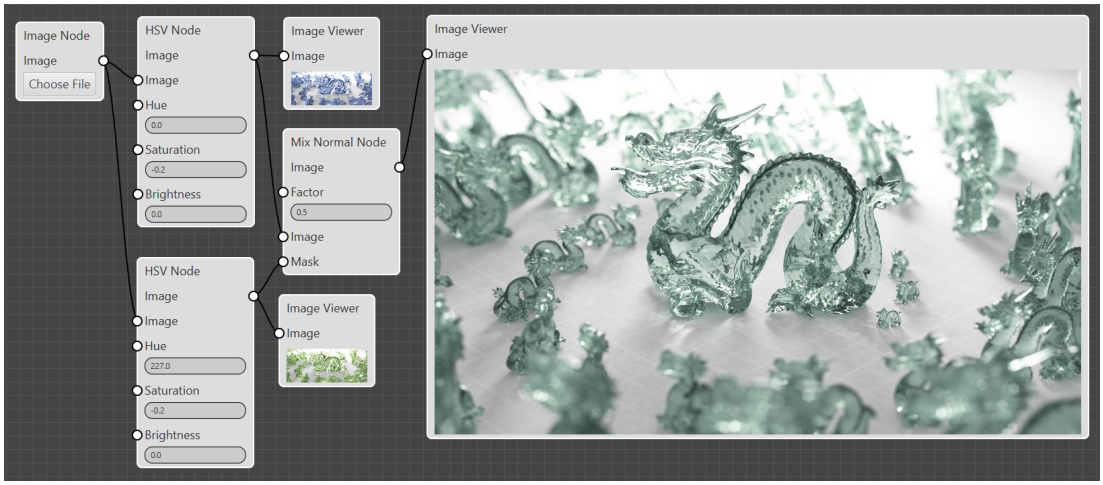
Since Blender is considered a professional software with a range of uses, primarily 3D modelling and animating, it also includes an image compositing tool which allows a user to digitally enhance images using a node-based editor. This tool is perfect to use as a reference to compare the image-processing application that was produced in the project.

5.2.1 Quality

For the image processing applications, an example image of 3D ray traced dragons was used as the input image. With this image, a node network was set up the same on both applications in order to get a fair comparison. The node network consisted of a single image being passed into two different Hue Saturation Value nodes followed by mixing the two HSV node outputs together using a "Normal" blend mode.



A. Comparison node network implemented in Blender with default settings.



B. Comparison node network implemented in the image processing application that was produced during the project.

Figure 5.1: A comparison between the image processing application developed throughout the project and Blender's compositor tool with default settings.

The resulting images of the node networks seen in figure 5.1 were then sent to a small sample size of 14 participants. The participants then decided which image they personally liked most and gave a small reasoning into what they preferred about the image they thought was best. Of the 14 participants, 12 preferred the image that resulted from the node network shown in figure 5.1B. with the reasoning generally being due to the fact that the image was a lot clearer and well defined than the resulting image of the node network in figure 5.1A. The 2 participants who preferred the image resulting from figure 5.1A. mentioned that while it was not quite as clear, they still preferred the washed out, low contrast look of the Blender result.

5.2.2 Ease of Use

A huge goal for this project was to create a node-editor framework that allowed for applications to be developed which require a little amount of knowledge on how it works. This goal resulted in an extremely intuitive image processing application which users are able to easily understand how the program works. A user given this program can quickly learn the basics and complete a task of digitally editing an image. On the contrary to this, Blender is an extremely complicated program which requires many hours of learning to be able to use the program to it's full potential. Certain features of Blender such as the image viewer node is extremely unintuitive to manipulate which is a huge downside to a powerful tool designed for allowing a user to manipulate images, where it would be expected to be able to resize an image and zoom in when necessary, however these features are almost impossible to complete in certain situations in Blender.

5.2.3 Efficiency

When it comes to computational efficiency and image processing, Blender has a notable advantage over the image processing application developed during the project. For large images or sizeable node networks, the project's image processing application takes considerably longer. The node networks shown in figure 5.1 were completed in similar times, however when using an image of larger resolution (4K), Blender completed the computation in approximately 1 second, however the project's processing application took approximately 10 seconds.

The efficiency issue with this project's image processing application can cause the program to feel very slow, especially when trying to make very small adjustments to an image to create the perfect result. This is extremely apparent when manipulating image control sliders as the whole program essentially halts when a change is made, only allowing further changes to the input when the computation has completed, whereas in Blender, changes to control sliders can be done during computation of new values which gives the program a much smoother and more efficient feel.

Chapter 6

Conclusions and Future Work

This project has demonstrated the implementation of a node-editor framework for Java which can be used to create a wide variety of applications with node-based interfaces. This document also includes a demonstration of an application for image processing extended from the framework to show how effective it is for creating intuitive applications with a node-based interface.

The node-editor framework has proved to be a great success when it comes to allowing developers to create new applications in a simple and efficient way due to the framework abstracting a large amount of technicality, while still being complex enough for the developer to create new components and extend the framework in almost any such way they desire. While integrating the framework into applications is exactly as intended, there are certain elements which have room for improvement.

The image processing application has also shown to be a great success due to the ability to match professional applications such as Blender's compositing tool. The resulting application from the project created images which a majority of survey participants preferred to the Blender counterpart due to a complexity in Blender's settings. The project had a key goal of producing an image processing application which requires very little learning by abstracting complex functions to allow a user to intuitively use the program.

Due to the scope of the project requiring a wide range of knowledge from a lot of different aspects of programming, the vast amount of learning that was required to accomplish all objectives of the project has been a very fulfilling experience.

6.1 Future Work

Although the project has resulted in the goals being achieved for both the node-editor framework and the image processing application, there are still a small number of features that could be added and solutions to certain problems that can be explored and implemented. This section will discuss some of the areas for improvement that will be worked on in the future.

As it currently stands, the node-editor framework essentially requires a new node class to be created for any different node, even if the implementation of the node function is slightly different, for example if there are two blurring algorithms, such as a Gaussian blur and Cubic blur, there would require two nodes, one for each algorithm. A solution to this would be to create a single blur node which allows the user to select which type of blurring algorithm is being used, and for each selection, allow the node itself to change to account for different inputs which may be required.

For the image processing application, the only major advantage that Blender had over it was the speed in which the computation was carried out. Since the image processing functions all used the JavaFX library to manipulate pixel data, it's entirely possible that JavaFX is not performing as efficiently as possible. To explore this, the project will intend to test the difference between JavaFX and reading the image data directly and writing to a new file. While this approach will probably not reach the speeds of Blender, any improvements to efficiency will be a great achievement.

Bibliography

[1] R. Eveleth, "How many photographs of you are out there in the world?" Nov 2015. [Online]. Available: https://www.theatlantic.com/technology/archive/2015/11/how-many-photographs-of-you-are-out-there-in-the-world/413389/

[2] ITU, "Ict data, world, 2005-2019," 2019, iTU Report. [Online]. Available: https://www.itu.int/en/ITU-D/Statistics/Pages/stat/default.aspx

[3] K. N. Whitley and A. F. Blackwell, "Visual programming: The outlook from academia and industry," in *Papers Presented at the Seventh Workshop on Empirical Studies of Programmers*, ser. ESP '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 180–208. [Online]. Available: https://doi.org/10.1145/266399.266415

[4] "Topic: Node-editors," 2020. [Online]. Available: https://github.com/topics/node-editor

[5] "Topic: Game engines," 2020. [Online]. Available: https://github.com/topics/game-engine

[6] B. R. Kent, *3D Scientific Visualization with Blender®*, ser. 2053-2571. Morgan & Claypool Publishers, 2015. [Online]. Available: http://dx.doi.org/10.1088/978-1-6270-5612-0

[7] "rete.js." [Online]. Available: https://rete.js.org/#/

[8] M. Selakovic and M. Pradel, "Performance issues and optimizations in javascript: An empirical study," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 61–72. [Online]. Available: https://doi.org/10.1145/2884781.2884829

[9] "Popularity of programming languages," 2020. [Online]. Available: http://pypl.github.io/PYPL.html

[10] C. Solomon and T. Breckon, *Fundamentals of digital image processing: a practical approach with examples in matlab.* New York: WILEY, 2010.

[11] L. G. Roberts, *Machine perception of three-dimensional solids*. Massachusetts Institute of Technology, 1963.

[12] R. B. Smith, "Experiences with the alternate reality kit: An example of the tension between literalism and magic," *SIGCHI Bull.*, vol. 17, no. SI, p. 61–67, May 1986. [Online]. Available: https://doi.org/10.1145/30851.30861

[13] D. Blythe, J. Kitamura, D. Galloway, and M. Snelgrove, "Virtual patch-cords for the katosizer," *ICMC 86 Proceedings*.

[14] P. E. Haeberli, "Conman: A visual programming language for interactive graphics," *SIGGRAPH Comput. Graph.*, vol. 22, no. 4, p. 103–111, Jun. 1988. [Online]. Available: https://doi.org/10.1145/378456.378494

[15] J. Shin, R. Siegwart, and S. Magnenat, "Visual programming language for thymio ii robot," in *Conference on Interaction Design and Children (IDC'14)*. ETH Zürich, 2014.

[16] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Trans. Comput. Educ.*, vol. 10, no. 4, Nov. 2010. [Online]. Available: https://doi.org/10.1145/1868358.1868363

[17] J. Trower and J. Gray, "Blockly language creation and applications: Visual programming for media computation and bluetooth robotics control," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 2015, pp. 5–5.

[18] A. Rao, A. Bihani, and M. Nair, "Milo: A visual programming environment for data science education," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2018, pp. 211–215.

Appendix A

Implementations of Relevant Algorithms

```
1  private void zoom(ScrollEvent e) {
2          double delta = e.getDeltaY();
3          double scale = Math.pow(1.001, delta);
4
5          double newWidth = viewport.getWidth() / scale;
6          double newHeight = viewport.getHeight() / scale;
7
8          newWidth = clamp(newWidth, MIN_ZOOM_WIDTH, width);
9          newHeight = clamp(newHeight, MIN_ZOOM_WIDTH * height/width, height);
10
11         Point2D i = viewToImage(e.getX(), e.getY());
12
13         double vx = e.getX() / view.getBoundsInLocal().getWidth();
14         double vy = e.getY() / view.getBoundsInLocal().getHeight();
15
16         double minX = i.getX() - (vx * newWidth);
17         double minY = i.getY() - (vy * newHeight);
18
19         minX = clamp(minX, 0, width-newWidth);
20         minY = clamp(minY, 0, height-newHeight);
21         viewport = new Rectangle2D(minX, minY, newWidth, newHeight);
22         view.setViewport(viewport);
23 }
```

Listing A.1: An implementation for digital zooming of an image in an ImageViewer node.

```
1 private Point2D viewToImage(double x, double y) {
2         double xRatio = x / view.getBoundsInLocal().getWidth();
3         double yRatio = y / view.getBoundsInLocal().getHeight();
4 
5         double ix = xRatio * viewport.getWidth() + viewport.getMinX();
6         double iy = yRatio * viewport.getHeight() + viewport.getMinY();
7 
8         return new Point2D(ix, iy);
9 }
```

Listing A.2: An implementation of a function which allows us to take a coordinate relative to the viewport and translate it to a coordinate relative to the image.

Appendix B

Implementations from other Sources

```
    constructor() {
      super("Mix");
    }

    builder(node) {
      var inp1 = new Rete.Input("a", "Value", Socket.value);
      var inp2 = new Rete.Input("b", "Value", Socket.value);
      var out = new Rete.Output("result", "Value", Socket.value);

      inp1.addControl(new TextControl(this.editor, "a"));
      inp2.addControl(new TextControl(this.editor, "b"));

      return node
        .addInput(inp1)
        .addInput(inp2)
        .addControl(new TextControl(this.editor, "preview", true))
        .addOutput(out);
    }

    worker(node, inputs, outputs) {
      var n1 = inputs["a"].length ? inputs["a"][0] : node.data.a;
      var n2 = inputs["b"].length ? inputs["b"][0] : node.data.b;

      var sum = Color(n1).mix(Color(n2));

      this.editor.nodes
        .find((n) => n.id == node.id)
        .controls.get("preview")
        .setValue(sum);
      outputs["result"] = sum;
    }
```

Listing B.1: An implementation of a colour mixing node in Rete.js.