Algorithm Learning Benefits Through the Use of Visualization Software

A Thesis
Presented to

The Faculty of the Department of Mathematics and Computer Science
Laurentian University

In Partial Fulfillment for
COSC 4235 Thesis in Computer Science

by

Joshua Renelli

Mar 29th, 2020

Table of Contents

Table of Figures

1 Background

Algorithms are defined as “a finite series of well-defined, computer-implementable instructions to solve a specific set of computable problems.” (). To rephrase the definition in a simpler manner, an algorithm can almost be thought of as a recipe where the instruction (arithmetic logic, variables, loops, etc.) are the “ingredients” and the order in which they are executed in are analogous to the order in which the ingredients would be used. Although the definition might not be overly difficult to understand, the application and conceptualization of a common set of algorithms (sorting, graph traversal, pathfinding, etc.) can prove to be a daunting experience for some. With the grades of first year computer sciences students falling to both extremes of failure and success, correlations have been determined leading to a students lack of success in this subject which include a lack of programming planning, a lack of programming conceptual understanding, and a lack of algorithmic skills (Sarpong *et al.*, 28). With these issues being so closely related to the subject matter and concepts being taught in introductory computer science classes, it is clear that there is a flaw in the methods that the course material is being taught.

1.1 Grade Distributions

Research into the failure rate of students enrolled in introductory computer science classes has been determined to have an average failure rate of 33%, while also producing the most common grade of an A (Bennedsen & Caspersen, 2007), which at first glance might indicate the existence of a bimodal grade distribution. A bimodal grade distribution occurs when the distribution of grades forms two peaks, as opposed to a normal distribution which has only one peak (Statistics How To, 2013). The notion that there could exist a bimodal grade distribution in introductory computer science classes has led to debates over a theory that there exists a “Geek Gene”, which attempts to argue that there exist two different populations of students in introductory computer science classes, those who just get it and those who do not (Basnet, 2019). However, research into the defence of the Geek Gene Hypothesis has shown that although bimodal grade distributions can be found, they should not be regarded as typical (at most 5.8% cases giving a type 1 error)(Patitsas, 2019). Amidst the failure in research towards the validity of the Geek Gene hypothesis, another hypothesis was proposed called the Learning Edge Momementum (LEM), which aims at attributing the problem to the course content and not necessarily the students (cte-blog, 2015).

The LEM hypothesis claims “Given some target domain of new concepts to be learned, any successful learning makes it somewhat easier to acquire further related concepts from the domain, and unsuccessful learning makes it somewhat harder. Thus when learning a new domain the successful acquisition of concepts becomes self-reinforcing, creating momentum towards a successful outcome for the domain as a whole, and similarly the failure to acquire concepts becomes self-reinforcing, creating momentum towards an unsuccessful outcome. " (Robins, 30-31). Subsequently, due to the fact that programming concepts are so tightly integrated (Robins 26), if a student is attempting to learn how to apply algorithms without a good base of knowledge surrounding the other facets of the programming language, they have essentially lost "momentum" and will likely struggle to do so. Given that an introductory to computer science class is most likely taken in a student's first year, this is a time in the student's academic career where they are first tasked with undertaking five university level courses over the course of five months, providing even more room for a student to fall behind due to external factors outside of the classroom (scheduling time, finances, etc.).

1.2 Learning Styles

The learning styles of students can play a big role in the outcome of students taking courses at any level (elementary school, high school, university, etc.), with students spanning across many different types of learning styles it can be difficult to tailor classes to adhere to everyone's best ability to learn. The technique of learning every student's individual learning styles might be a technique that is feasible at the elementary level, but at a post-secondary level where student's interact with professors for a limited amount of time, such a task would be highly unfeasible and instead, the professor should focus on "teaching around the cycle" ensuring that course content is geared towards a plethora of different learning styles (Felder, Felder, & Dietz, 12). In the event that a professor is not able to adhere to multiple types of learning styles in the course content, it has been proven to leave students feeling uncomfortable as they are not able to learn the material properly (Felder, Felder, & Dietz, 12). Referring back to the LEM hypothesis, Robins (34) makes it clear that the area leading to a student's failure in grasping concepts related to the learning of programming languages is closely related to the content that is being taught.

In a study conducted exploring for correlations between course preparedness, learning style, and course material towards the success of a student in an introductory computer science class, the following observation was made, "Several learning style variables were correlated with outcome. Reflective and verbal learning style students achieved top grades more frequently and lower grades less frequently than their scale opposites (active and visual learners respectively)." (Allert, 2004). This observation clearly identifies that the current way that first-year computer science classes are taught leaves students who identify primarily as visual learners in a disadvantageous position strictly due to their learning style. One argument that could be made against the choice of diversifying the course material to make it more digestible for individuals of varying learning styles could be that by doing this you will then be inadvertently hindering the ability of students who were previously more dominant (reflect and verbal learners). Although a valid argument, in practice this is not what would occur as by diversifying the content leaves students functioning in their preferred learning style only part of the time, which actually assists with developing skills in their less preferred learning styles (Felder, Felder, & Dietz, 12).

1.3 Visual Aids in Education

In the education field, it has been estimated that 65 percent of students are visual learners, 30 percent are auditory learners and a mere 5 percent are kinesthetic learners (Gangwer, 2015). With such a large portion of students identified as being visual learners, many forms of visual aids have been created to assist those students in the education process. A visual aid can be defined as “tools that help to make an issue or lesson clearer or easier to understand and know (pictures, models, charts, maps, videos, slides, real objects, etc.)” (Shabiralyani, 226). Shabiralyani (226) goes on to discuss the benefits of utilizing visual aids in education such as providing complete examples for conceptual thinking, creating an environment of interest for the students, providing direct experience to the students, etc. Stepping into the domain of computer science education, many attempts have been made to develop tools to assist visual learners. Examples of such tools include Wally Feurzeig’s LOGO which is a programming language with visual elements incorporated into it, scratch which is another programming language which relies on visuals as programs are created by dragging visual code blocks and finally, a popular genre of tools known as algorithm visualization tools (Yim et al. 465).

1.4 Algorithm Visualizers

Algorithms and data structures are essential concepts that every student studying computer science requires, as such, a popular method of improving the learning of these concepts has been the introduction of algorithm visualizers into the education of this content (Shaffer et al. 2010). Algorithm visualizers aim to assist with shifting the thought process of algorithms away from memory locations and function calls and instead, illustrating the data structures in a more natural and abstract way (Shaffer et al. 2010). The effectiveness of the use of algorithm visualizers is clear as research studies have shown that the student interacting with an algorithm visualizer performed significantly better than a student who simply listened to a lecture on the content (Mudner & Shakshuki,1). Although the effectiveness of algorithm visualizers have been proven, not every aspect of algorithm visualizers have been shown to have pedagogical value. Some features that an algorithm visualizer can have to improve the effectiveness are (1) allowing the user to step through the algorithm manually as opposed to controlling the rate at which the algorithm visualizer runs and (2), having a pseudocode display as well as an active guide as to what step in the pseudocode the algorithm visualizer is currently at causes users to spend substantially more time utilizing the algorithm visualiser (Saraiya et al. 5).

1.5 Benefits of Utilizing Game Engines

For years, game engines have been growing in popularity along with the rise of the gaming market itself. With the global gaming market hovering around $135 billion, as too have many popular game engines such as Unity which currently valued around $6 billion (Techcrunch, 2019). With the large amount of resources being amassed into the creation of feature rich gaming engines, not only have these gaming engines been recognized for the specific use of developing video games but they have also been recognized for their use in developing software for other applications like science visualizations (Jacobson & Michael, 29). The benefits of utilizing a game engine for the creation of software applications span across a plethora of reasons, some of which being low development cost, large communities of support, ability to build projects to multiple device types, etc (Te Brake, Guido, et al., 3).

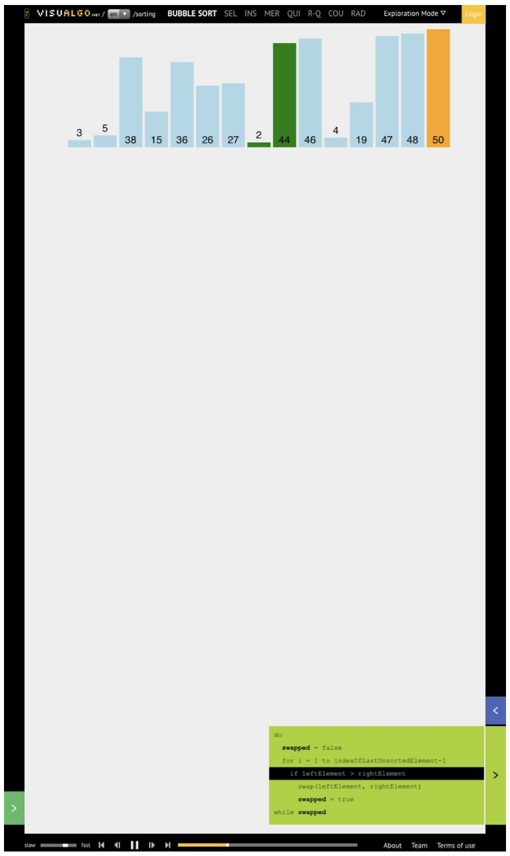
1.6 Popular Algorithm Visualizers

As might be expected based on the research, there do currently exist algorithm visualization tools on the market on both the web and mobile platforms. Now, due to the inherent nature of both platforms, these tools both come with clear advantages and disadvantages. Although the most robust as they can function on both desktops and mobile devices, the web-based visualizers struggle with over-cluttering the screen with displays and options as they have clearly been designed with a desktop in mind. As for the mobile visualizers, although they do benefit from the advantage of being designed specifically for smaller screen sizes and can be run without the need of an internet connection, with such a small market of options to choose from, as a consumer and student you are left seeming complacent with a limited selection. As of the time of the creation for this thesis, based on my research I have not found any multiplatform algorithm visualization tool that can be built natively to both desktop and mobile devices. As mentioned above, the closest applications that mimic this requirement would be that of web-based solutions, leaving a large gap in the market and an opportunity for such an application to be developed.

1.6.1 Web-Based Solutions

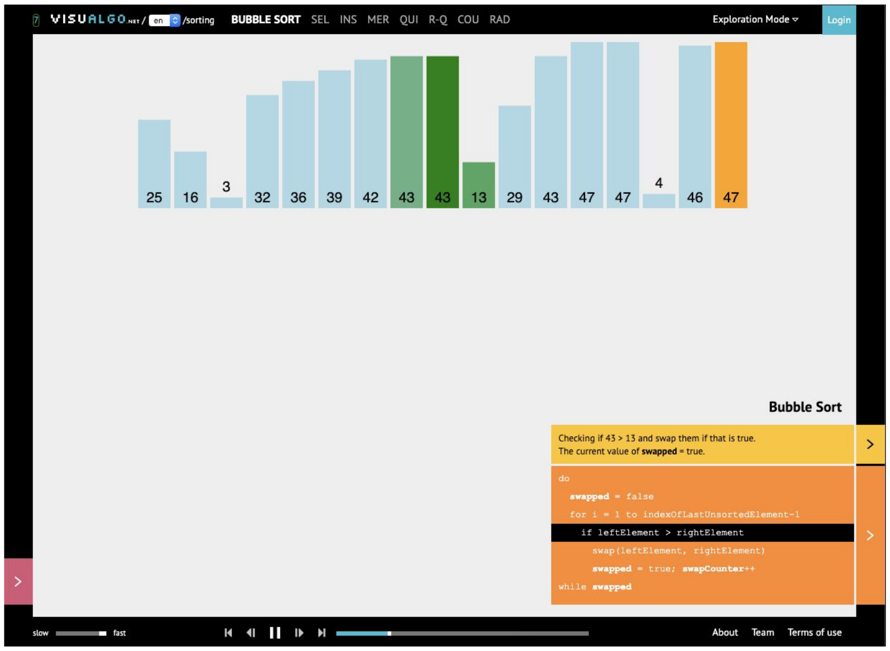
Web-based solutions by design will offer the most robust option for the distribution of the software as it can be utilized on the majority of devices. With that in mind, however, a problem that plagues countless web applications is when it has been designed with a desktop display in mind and has not taken into account, mobile-only users. Although the majority of these websites do adhere to common responsive design principles (as they are required for multiple sized desktop screens) when translating that design onto a drastically smaller medium such as a phone screen, the loss of clarity is immediately abundant. The two web-based visualizations tools that will be critiqued are *Visualgo* (visualgo.net) and *Algorithm-Visualizer* (algorithm-visualizer.org).

Visualgo has the most simplistic design out of the two options, offering visualizations for a plethora of different algorithms and data structures, but for the purpose of this, we are only interested in the sorting algorithm section. The application consists of three main sections, the display of the bars, the display of the code, and a section for altering the settings and pause/play buttons. On a computer, this application would be a perfect suit for students wanting to learn sorting algorithms as it displays the necessary information, without cluttering up the screen with additional (potentially unnecessary) information (Figure 1.2). Unfortunately, on mobile, this experience is not quite the same. The application is either too small to be seen and when zooming in, parts of the application tend to overlap causing some issues with the visuals. Overall, not a terrible experience on mobile but it could use some improvement (Figure 1.1).



```
do
  swapped = false
  for i = 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap(leftElement, rightElement)
      swapped = true
while swapped
```

Figure 1.1: A preview of the *Visualgo* web-app when displayed on an iPhone X in vertical orientation



Checking if 43 > 13 and swap them if that is true.
The current value of **swapped** = true.

```
do
  swapped = false
  for i = 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap(leftElement, rightElement)
      swapped = true; swapCounter++
while swapped
```

Figure 1.2: A preview of the *Visualgo* web-app when displayed in a desktop web-browser

Algorithm-Visualizer takes a slightly different approach than *Visualgo* as it still offers a wide variety of algorithms to view visualizations for, but instead offers this in a more feature-rich environment. With the ability to not only view the bars getting sorted, but the user is also able to see the exact code that is getting run, a 2D trace of the array and even a log of every event that is occurring (Figure 1.4). Another huge difference between the two visualization tools is that in order to edit the values being sorted on *Algorithm-Visualizer*, the user must actually edit the code itself as opposed to being provided with a UI to accomplish the same task. For these two reasons, it can be observed that *Algorithm-Visualizer* might not be the ideal choice of a visualization tool for beginner programmers/students as there exists a good amount of overhead required to view and understand the visualization, which could result in a negative impact on the comprehension of the algorithm due to the user's inability to control the tool. Furthermore, as a result of cluttering the screen with extra information, this tool is essentially not usable on the mobile platform as it has not been designed to shrink properly on smaller screen sizes (Figure 1.3).

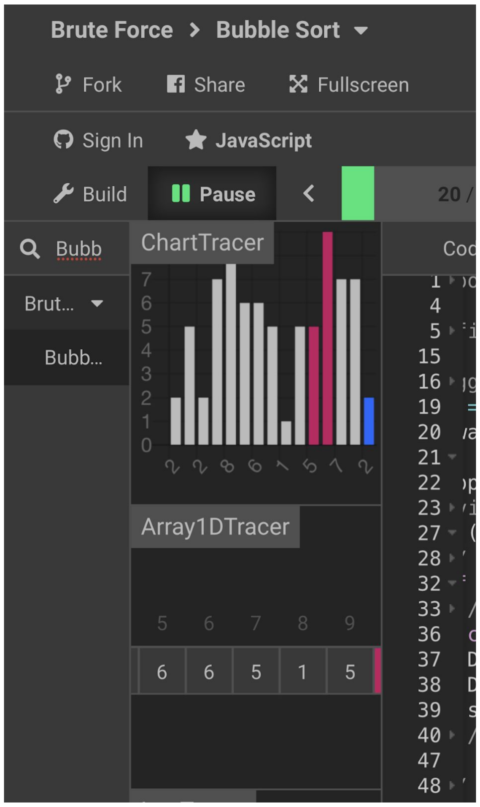
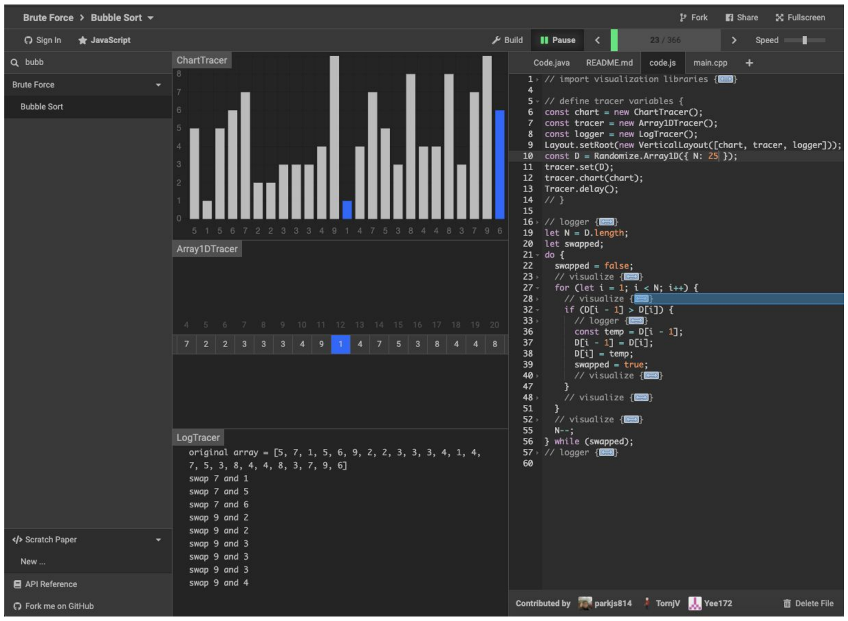


Figure 1.3: A preview of the *Algorithm-Visualizer* web-app when displayed on an iPhone X in vertical orientation



```
// import visualization libraries {}

// define tracer variables {
const chart = new ChartTracer();
const tracer = new Array1DTracer();
const logger = new LogTracer();
Layout.setRoot(new VerticalLayout([chart, tracer, logger]));
const D = Randomize.Array1D({ N: 25 });
tracer.set(D);
tracer.chart(chart);
Tracer.delay();
// }

// logger {}
let N = D.length;
let swapped;
do {
  swapped = false;
  // visualize {}
  for (let i = 1; i < N; i++) {
    // visualize {}
    if (D[i - 1] > D[i]) {
      // logger {}
      const temp = D[i - 1];
      D[i - 1] = D[i];
      D[i] = temp;
      swapped = true;
      // visualize {}
    }
    // visualize {}
  }
  // visualize {}
  N--;
} while (swapped);
// logger {}
```

```
original array = [5, 7, 1, 5, 6, 9, 2, 2, 3, 3, 3, 4, 1, 4,
7, 5, 3, 8, 4, 4, 8, 3, 7, 9, 6]
swap 7 and 1
swap 7 and 5
swap 7 and 6
swap 9 and 2
swap 9 and 2
swap 9 and 3
swap 9 and 3
swap 9 and 3
swap 9 and 4
```

Figure 1.4: A preview of the *Algorithm-Visualzier* web-app when displayed on a desktop web-browser

1.6.2 Mobile-based Solutions

During a time in society and education where the majority of students own a smartphone, it is obvious that in order to have an application be consumed by the most amount of people, that it is developed with this in mind. As an alternative to a web application, native phone applications will provide the student/user with the best experience on their mobile device. Being the only phones/tablets that I have access to are apple products (iPhone, iPad), my research into the current most popular algorithm visualizers on mobile devices has been limited to the offerings on the app store and not the google play store (android applications). From the time of creating this thesis, it appears that there only exist two algorithm visualizers on the app store, *Algorithm: Sorting Visualizer* (https://apps.apple.com/ca/app/algorithm-sorting-

visualizer/id1503961121), and *Algorithms: Explained&Animated* (*https://apps.apple.com /ca/app/algorithms-explained-animated/id1047532631*).

The first application *Algorithm: Sorting Visualizer,* gives the user the most control over the sorting, allowing them to select the number of bars to sort as well as the sorting. The application also gives the user the ability to read a quick description of the selected sorting algorithm along with providing the time and space complexity of the algorithm, which is very pertinent knowledge for any student enrolled in a data structures or algorithm analysis class. Unfortunately, unlike their web-based alternatives, this mobile application does not display the code or pseudocode for the algorithm while it is sorting, which makes it more difficult for the user to better understand what is actually occurring during each step as they are simply looking at the bars move with no other information (Figure 1.5).

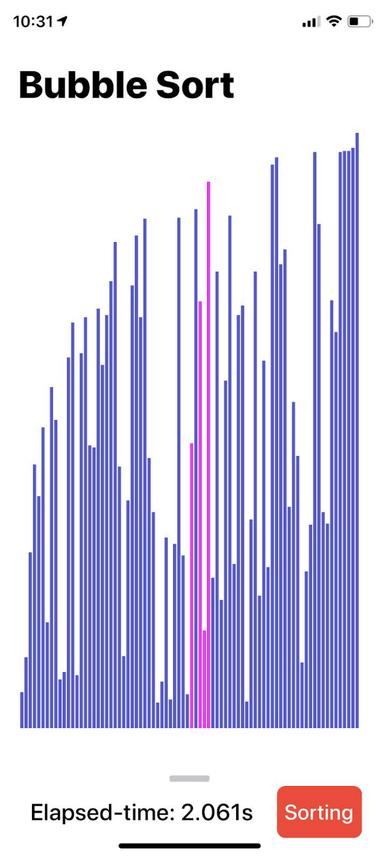


Figure 1.5: A preview of the *Algorithm: Sorting Visualizer* mobile-app when displayed on an iPhone X in vertical orientation

The second application *Algorithms: Explained&Animated,* although providing the user with a more broad range of algorithms to choose from (sorting, list searching, graph searching, etc.), delivers a more shallow learning experience. This application has by far the least amount of user control out of any the options, only allowing the user to shuffle the values, move ahead a step in the algorithm, and play through the entire algorithm. The lack of controls, coupled with the fact that (similar to the previously mentioned application) there is no display of the code for the algorithm, does not make this application an excellent option for students wanting to fully engage in the learning of sorting algorithms.

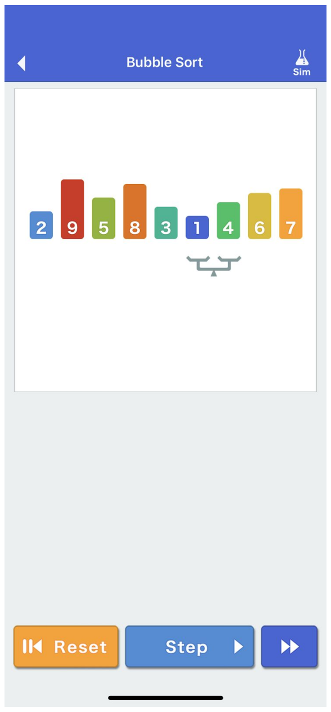


Figure 1.6: A preview of the *Algorithms: Explained&Animated* mobile-app when displayed on an iPhone X in vertical orientation

2 Methodology

The method approached in this thesis in order to create an algorithm visualizer involves the use of the *Unity* game engine to create a multi-platform solution to assist with the learning of sorting algorithms. The use of a game engine as the platform for the software solution provides a plethora of advantages for the purposes of this thesis as well as any other multi-platform application that incorporates a user interface.

2.1 Unity

Unity is a game engine that as you might expect from the name (game engine), is traditionally used for the creation of both 3D and 2D games. When determining what software platform to choose during the planning stage of any software project, it might not be initially apparent to look into the use of a game engine, especially if you are not making a game specifically. Although there are many aspects that game engines (*Unity* specifically) contain that make it a great candidate to look into as an option. The benefits of using Unity are as follows; first off, since the primary goal of Unity is to create fully-functional games, Unity by default provides essentially all graphics capabilities that you would require for most applications. Secondly, although not an aspect that every game engine contains, *Unity* provides the ability to build one project to any device that you would most likely need for your application (i.e. Windows, MacOS, iOS, Android, HTML OpenGL, etc.). Finally, due to the aforementioned fact that Unity has been around since 2005 and is also one of the most popular game creation engines on the market, it’s time on the market paired with popularity has allowed for a very large community to grow to aid with the creation of new and existing projects.

2.1.1 Unity Graphics Capabilities

When creating a software solution that requires the use of a user interface, depending on what platform the solution is being built to or what it is being built with, it can be a challenge to narrow down the options and what will suit the project the best. By opting to go with a game engine such as Unity, it already contains the groundwork required to use graphics APIs such as OpenGL, Direct3D, and many others. When working on a Unity project there is the option to work in either a 3D or 2D space, giving the user flexibility depending on what they are hoping to achieve out of their application. When utilizing Unity in order to create a user interface, it is clear that this will be taking advantage of the 2D space. Furthermore, since Unity is the wrapper containing both the assets, rendering and scripts, there exists simple and dedicated support for building the user interface to devices that encompass different screen sizes.

2.1.2 Unity Multi-platform Builds

It is a well-known fact that custom software solutions take a large amount of time to create due to the many steps involved. The three main steps encompassing any software solution being (1) requirements & design, (2) planning, architecture & development, and (3) software testing. Now although there is obviously room for variation depending on the scope of a project and the team working on it, on average by following these three steps it would take four to nine months to end up at a finished product (Soltech 2019). Keeping in mind that this is only taking into account the development for one platform, and although the first step might not have to be repeated for development on other platforms, the bulk of the workload which is contained inside the second two steps would have to be.

In an attempt to mitigate this issue for development on certain platforms, there exist frameworks that allow for the creation of so-called “hybrid” applications. Hybrid applications allow for the creation of one (generally web-based) codebase that are then put into a native application shell that is able to interface with the specific native API calls associated with that device. Due to the drastic reduction in delivery time for software applications these frameworks have been growing in popularity as of recent. Some popular examples of hybrid frameworks include the likes of Electron which is a javascript framework for the creation of hybrid desktop applications, and React Native which is another javascript framework that is used to create hybrid mobile applications.

This is where Unity comes in, with the ability to build one project to over 25 different devices, what was once a pain-point for many software applications is now an afterthought. Of course, some provisions must be put in place to ensure that the application runs and displays properly on different devices with different screen sizes. In summary, by using a multi-platform system such as Unity, the delivery time for the software application can be reduced from 1.5 - 2 years (assuming the application intended to be built to around 3 different device types), to the originally stated 4-9 months for an application that can be built to all devices.

2.1.3 Unity Community/Developer Support

It is clear at this point that the Unity game engine contains a lot of features that make it appealing to begin development with (expansive graphics suite, multi-platform building, etc.), but at times frameworks/engines like this can be plagued with a cumbersome learning curve for new developers that deter them from wanting to approach and utilize these tools. To Unity's great benefit, due to its longevity in the market as one of the most popular gaming engines, the engine has both an incredible community of developers releasing content teaching the mechanics of Unity, coupled with developers who offer continued support and simple documentation.

From experience, three of the more useful resources for learning the "ins and outs" of Unity are Youtube, the Unity forums, and Unity's very own learning portal. The constant need for effective Unity tutorials has spawned many successful Youtubers, some of which have amassed followings of over a million subscribers, thereby allowing them to pursue it as their full-time job and in turn consistently releasing new Unity tutorials. In the event that a problem with a Unity project arises (that is not covered directly in a Youtube tutorial), the Unity forums act as a great hub for quick solutions to sometimes simple or challenging problems (the Unity forums provide the same means of assistance that Stack Overflow would provide to non-Unity application development). In the event that a developer is looking for a more applicable and content-rich tutorial, Unity has a dedicated learning “portal”, located both on the Unity website as well as in the Unity Hub application. On the learning portal, developers are able to load in what are essentially half-finished projects and are provided with the tools and explanation on how to complete it. Although these tutorials are more sparse than what can be found on Youtube, they are more direct with the assistance given as its features are built directly into the Unity application.

2.1.4 Unity Immediate Mode GUI (IMGUI)

When developing a user interface in Unity there are two main options to choose from, those options being either the *gameobject*-based UI where each UI element is associated with a *gameobject* or the Immediate Mode GUI (IMGUI) system that involves the creation of UI programmatically. For systems that involve a good portion of UI elements being generated dynamically (as is the case with this application), it makes sense to go with the IMGUI approach as the entire system can be generated without the need to create *gameobjects*. By creating UI elements at startup, the system is able to determine the device’s screen size and set the size of all instantiated UI elements accordingly, allowing for simple and seamless responsive design.

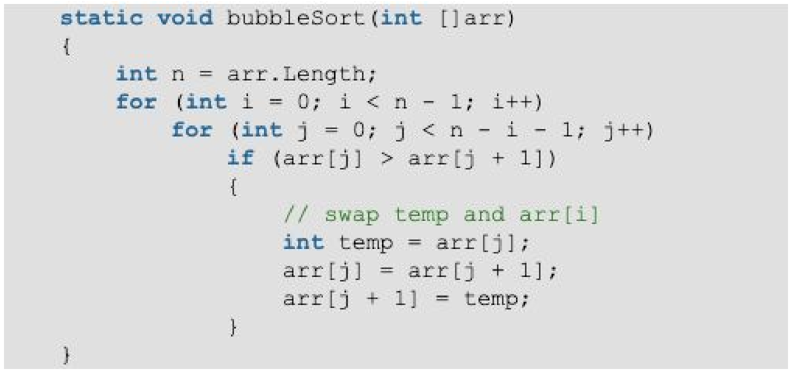
2.2 Sorting Algorithms

Sorting algorithms are among the first (if not the first) algorithms that are taught to students taking a first-year computer science course. They are a beneficial set of algorithms, to begin with as sorting is both an essential concept for any computer scientist and there exist many different types of sorting algorithms allowing for simpler ones to be taught first. The algorithms that have been studied and applied in this thesis include bubble sort, insertion sort, selection sort. These algorithms have been selected as they are the ones a student in a first or second-year university class might expect to encounter.

The actual process of a sorting algorithm involves the rearrangement of values in a data structure that is generally some time of array or list. The order of rearrangement is determined by how the algorithm is defined, and the items are sorted based on their value (i.e. integer values sorted based on numeric value).

2.2.1 Bubble Sort

Bubble sort is normally dubbed as the “simplest” sorting algorithm and also commonly referred to as a teaching algorithm as there are generally no real applications that implement bubble sort to sort data as it is not efficient. Given an array of size n, this algorithm involves repeatedly checking adjacent values (array[n] and array[n+1]) and swapping them if they are out of place (Figure 2.1). Every iteration of the outer loop (from 0 to n) results in a value being placed in its final position. From a complexity point of view, this algorithm has an order of n^2.

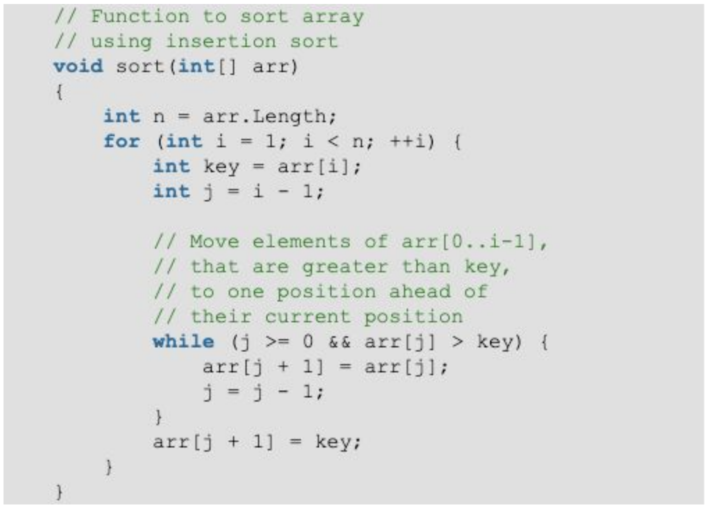


```
static void bubbleSort(int []arr)
{
    int n = arr.Length;
    for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
            {
                // swap temp and arr[i]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
}
```

Figure 2.1: Example code snippet of the Bubble Sort algorithm

2.2.2 Insertion Sort

Insertion sort is yet another simple sorting algorithm that functions by conceptually dividing the unsorted array into 2 different sections. The first sub-array which starts empty and is at the beginning of the array will contain the newly sorted values and the second sub-array are the remaining unsorted values that follow the sorted sub-array. Given an array of size n, the algorithm will loop through the unsorted subarray n times and based on the first value in the unsorted array it will determine its location in the sorted subarray and then shift the elements of the sorted subarray in order to fit the new value (Figure 2.2). This algorithm has an order of n^2 as the algorithm uses a double loop, one for retrieving the new key and one for shifting all of the elements in the sorted sub-array.

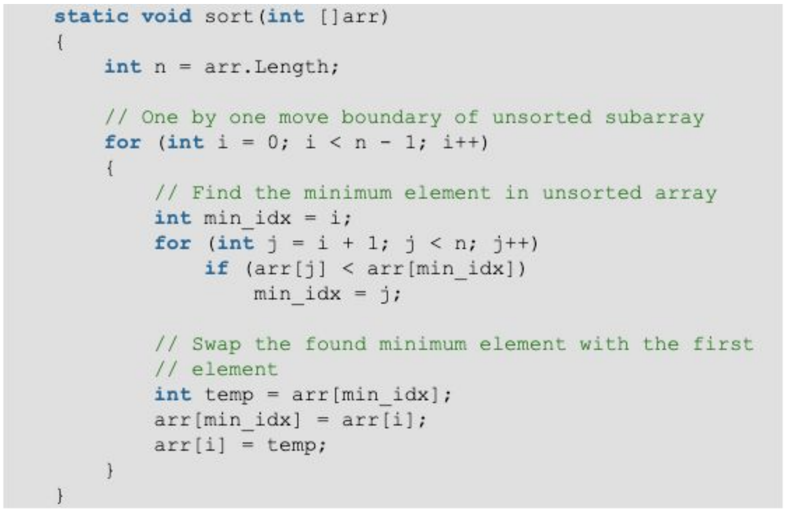


```
// Function to sort array
// using insertion sort
void sort(int[] arr)
{
    int n = arr.Length;
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        // Move elements of arr[0..i-1],
        // that are greater than key,
        // to one position ahead of
        // their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Figure 2.2: Example code snippet of the Insertion Sort algorithm

2.2.2 Selection Sort

Selection sort is yet another simple sorting algorithm that functions similarly to that of the insertion sort algorithm. The selection sort algorithm divides the input array into two sub-arrays, one that is sorted and one that is unsorted. Every iteration of this algorithm loops through the unsorted subarray in order to find the minimum value and then swaps that values position with the position of the first value in the unsorted array (essentially placing the minimum value at the end of the sorted array) (Figure 2.3). As with two previously mentioned sorting algorithms (bubble sort and insertion sort), this algorithm operates with an order of n^2 as it uses a nested loop structure.



```
static void sort(int []arr)
{
    int n = arr.Length;

    // One by one move boundary of unsorted subarray
    for (int i = 0; i < n - 1; i++)
    {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first
        // element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Figure 2.3: Example code snippet of the Selection Sort algorithm

3 Implementation

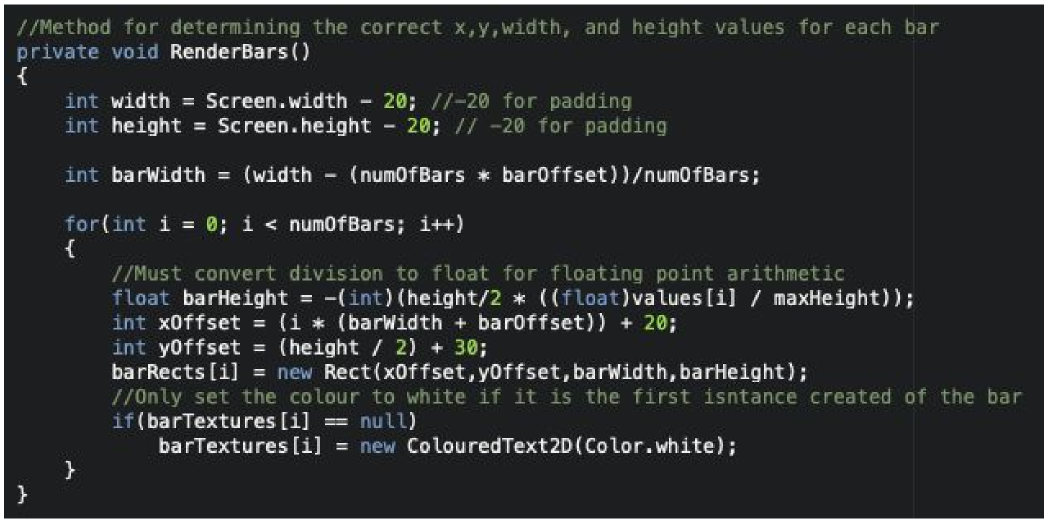
The implementation of this thesis was created completely using the Unity game engine due to the aforementioned benefits and by association, all of the coding was done using the C# programming language as that is the primary programming language that Unity utilizes for its script system. The visualization software allows for the visualization of 4 different sorting algorithms (bubble, insertion, selection), of which the work will be discussed by breaking down the implementation of a few of the main inner-systems. These inner-systems include the visualization of the array data as bars with a height that is representative of their associated value, a settings menu which allows for the customization of data to be displayed and sorted along with the algorithm to be utilized, the step-by-step display of the pseudocode for the given algorithm, and finally the process of bringing all of these inner-systems together to apply the sorting algorithms. To note, all of the systems have been implemented with the thought process of building to multiple devices and systems in mind, ensuring that the correct measures have been put in place to allow for a “responsive” application that looks and operates well on a plethora of different display sizes.

3.1 Bar Display

In order to visually display the values that will be used in the sorting algorithms, a commonly used approach by other visualization tools and the one that is used in this thesis involves displaying the values as bars of which the horizontal height is used to distinguish the value of the bar. With a couple of different methods that could be used to display the bars to choose from, that being the traditional combination of *gameobjects* with a *spriterenderer*, the main user interface system in Unity, or the immediate mode GUI system. In the end, taking into consideration the fact that it is a flat 2D interface that will require the ability to dynamically change size depending on display size, I ended up going with the IMGUI approach.

The main sorting system is located inside of a *gameobject* called BarsHandler, which is an empty *gameobject* holding nothing but the *sorting.cs* script. This is because as mentioned before, with the IMGUI approach there is no need to store any rendering information as that will all be done programmatically in the script. The script maintains three main arrays that are essential for both the sorting algorithms and the rendering of the bars to the screen. The array *barRects* holds the Rect (Unity object which holds position and size) references for each bar, the array *values* which holds the values of the bars and finally, the array *barTextures* which holds ColouredText2D's (which are a custom wrapper object of Unity's Texture2D object) and are essentially used to store the current colour of any given bar.

The Rect objects stored in *barRects* are calculated both during the initial rendering of the bars as well as any time there is any update to the *values* array, as *barRects* is essentially a map of *values* converting them into Rect objects. To calculate the required dimension for the bars, first off, the total displayable area that can be taken up is calculated with a width of the entire width of the screen (minus 20 pixels of padding) and a height of half the screen height (minus 10 pixels for padding). Next, the actual bar width is then calculated based on the total usable width divided by the total number of bars. Finally, the height of the bar is calculated by determining the proportion of the value based on the maximum possible value (max height), and then taking that proportion and multiplying it by the maximum available height for the display (Figure 3.1).



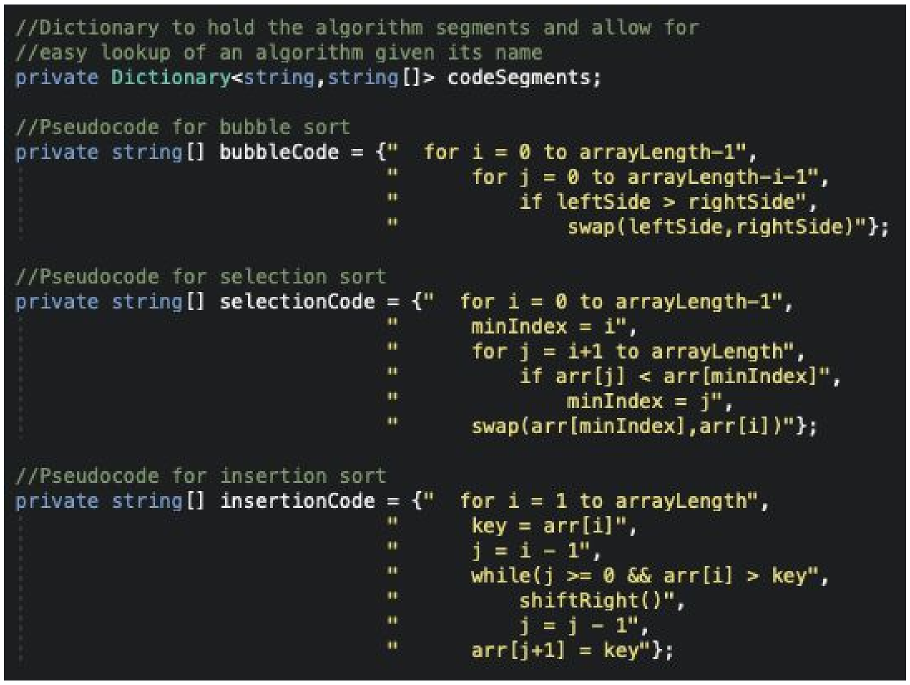
```
//Method for determining the correct x,y,width, and height values for each bar
private void RenderBars()
{
    int width = Screen.width - 20; //-20 for padding
    int height = Screen.height - 20; // -20 for padding

    int barWidth = (width - (numOfBars * barOffset))/numOfBars;

    for(int i = 0; i < numOfBars; i++)
    {
        //Must convert division to float for floating point arithmetic
        float barHeight = -(int)(height/2 * ((float)values[i] / maxHeight));
        int xOffset = (i * (barWidth + barOffset)) + 20;
        int yOffset = (height / 2) + 30;
        barRects[i] = new Rect(xOffset,yOffset,barWidth,barHeight);
        //Only set the colour to white if it is the first isntance created of the bar
        if(barTextures[i] == null)
            barTextures[i] = new ColouredText2D(Color.white);
    }
}
```

Figure 3.1: Code snippet from *Sorting.cs* of method used to calculate rectangles size/position that represent the bars

3.2 Code Display

Following in suit with the method of displaying the bars to the screen, displaying the pseudocode for the algorithms is being done with the IMGUI system due to its flexibility of being able to adjust dimension programmatically in a simple manner. The code display system is primarily located in the *CodeDisplay.cs* script which is instantiated in the scene in a *gameobject* called *CodeDisplay*.

The code display system functions by storing the pseudocode for the three selectable algorithms as string arrays , where each index of the array stores a different line in the algorithm (Figure 3.2). Then, depending on which algorithm is currently “selected”, the sorting algorithm is able to interface with the code display system in order to indicate the current step (current line of pseudocode) or steps that should be highlighted on the code display.



```
//Dictionary to hold the algorithm segments and allow for
//easy lookup of an algorithm given its name
private Dictionary<string,string[]> codeSegments;

//Pseudocode for bubble sort
private string[] bubbleCode = {"  for i = 0 to arrayLength-1",
                               "      for j = 0 to arrayLength-i-1",
                               "          if leftSide > rightSide",
                               "              swap(leftSide,rightSide)"};

//Pseudocode for selection sort
private string[] selectionCode = {"  for i = 0 to arrayLength-1",
                               "      minIndex = i",
                               "      for j = i+1 to arrayLength",
                               "          if arr[j] < arr[minIndex]",
                               "              minIndex = j",
                               "      swap(arr[minIndex],arr[i])"};

//Pseudocode for insertion sort
private string[] insertionCode = {"  for i = 1 to arrayLength",
                               "      key = arr[i]",
                               "      j = i - 1",
                               "      while(j >= 0 && arr[i] > key",
                               "          shiftRight()",
                               "          j = j - 1",
                               "      arr[j+1] = key"};
```

Figure 3.2: Code snippet from *CodeDisplay.cs* of pseudocode stored for each sorting algorithm

Similarly to the approach for displaying bars, each line of code to be displayed for the currently selected algorithm also has an associated texture (for colour) and Rect (for position and size). In this case, the array holding the textures is called *codeTextures* and holds objects of type CodeSegmentTexture which is another custom Texture2D wrapper class, and the array holding the Rect objects is called *codePositions*. To calculate the correct height and width for each line of code, first, a container GUI box is rendered to the screen with a width that is either ½ or ⅓ the screen size depending on the screen width, and a height that is relative to the remaining space between the bar display and the bottom settings bar. From this container, we are able to calculate the width of each label which is equal to the width of the container and then the height is calculated by dividing the total available height by the number of lines of code.

At this point, it is able to render both the container for the code and the Rect's for each line of code but the last calculation that needs to be made is one of the more interesting ones which is determining a font size on a responsive UI component. The first step involves determining which line of code is the longest as if we can determine a font size that makes the longest line fit in the container, then it is clear that the other lines of code will do the same. In order to find the longest line of code, we can simply loop through all of the lines of code and use the string Length property to determine this. After retrieving the longest line of code, we create a GUIStyle object that matches the one we use for rendering the lines of code to the screen and take advantage of a member method called *CalcSize()* which allows us to determine the amount of space a GUIContent (our line of code) will use given a certain GUIStyle. Using this method, we are able to create a loop that continuously performs this check against the total available size, increasing the font size until it is either too large vertically or horizontally, at which point we are left with the desired font size (Figure 3.3).

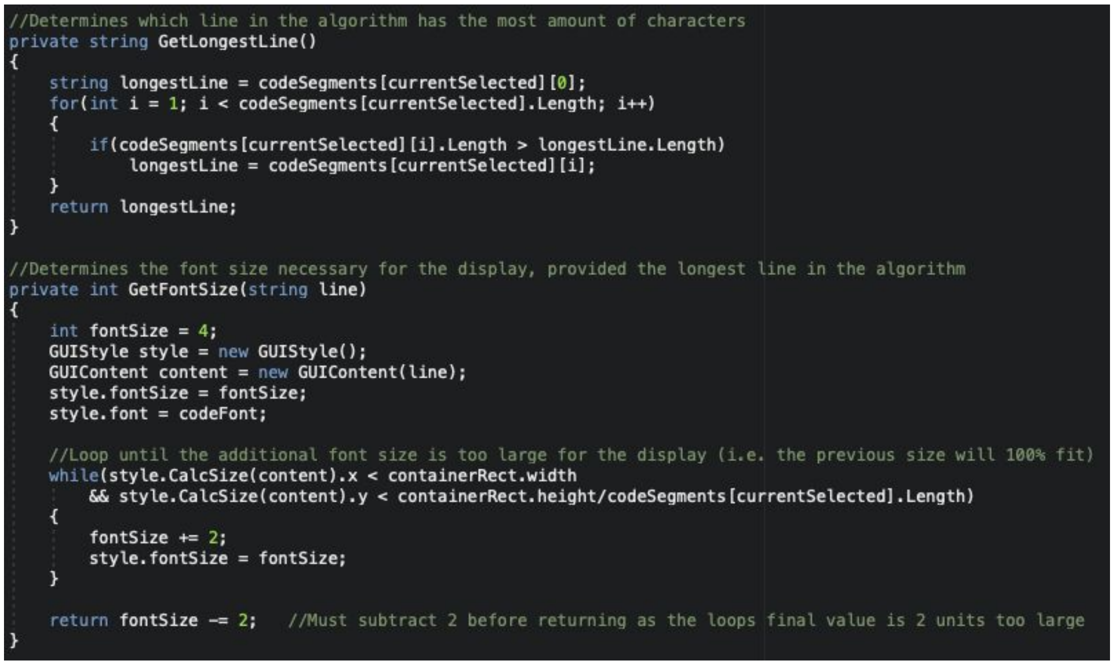
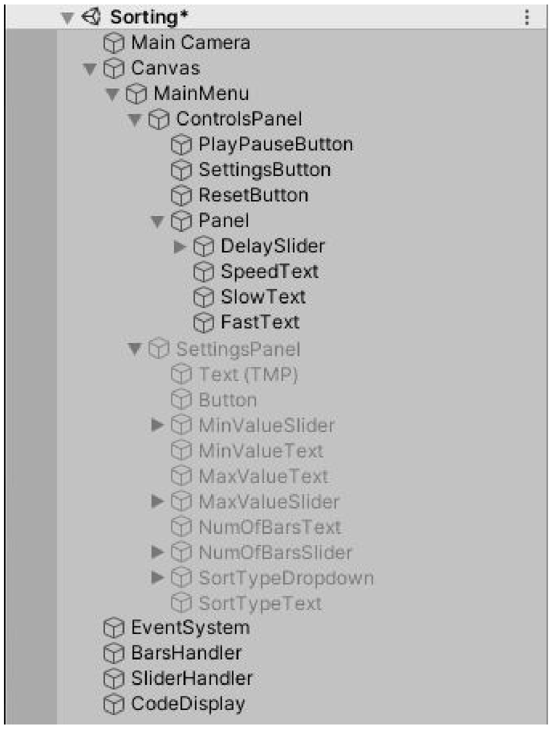

```
//Determines which line in the algorithm has the most amount of characters
private string GetLongestLine()
{
    string longestLine = codeSegments[currentSelected][0];
    for(int i = 1; i < codeSegments[currentSelected].Length; i++)
    {
        if(codeSegments[currentSelected][i].Length > longestLine.Length)
            longestLine = codeSegments[currentSelected][i];
    }
    return longestLine;
}

//Determines the font size necessary for the display, provided the longest line in the algorithm
private int GetFontSize(string line)
{
    int fontSize = 4;
    GUIStyle style = new GUIStyle();
    GUIContent content = new GUIContent(line);
    style.fontSize = fontSize;
    style.font = codeFont;

    //Loop until the additional font size is too large for the display (i.e. the previous size will 100% fit)
    while(style.CalcSize(content).x < containerRect.width
        && style.CalcSize(content).y < containerRect.height/codeSegments[currentSelected].Length)
    {
        fontSize += 2;
        style.fontSize = fontSize;
    }

    return fontSize -= 2;   //Must subtract 2 before returning as the loops final value is 2 units too large
}
```

Figure 3.3: Code snippet from *CodeDisplay.cs* of methods used to calculate the desired font size given the display size

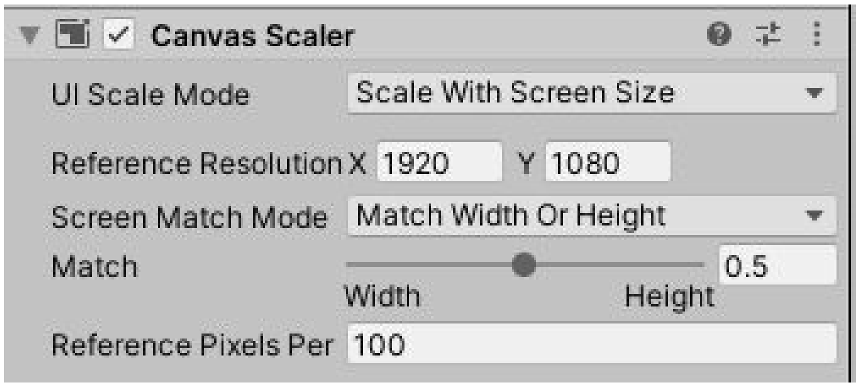
3.3 Controls UI

Unlike the method that the bar and code displays have been implemented - which have been done using IMGUI objects in Unity - the user interface for the controls of the software has been implemented using Unity's *gameobject*-based UI (Figure 3.4). The *gameobject-based* UI approach made more sense for the implementation of the controls as the proportionate location of the controls remains static, and by using the Unity editor we are able to easily organize where all of the different components will be placed. Now, due to the fact that they are not being instantiated in code and organized in a responsive way through that medium, this responsive nature can also easily be created directly on the *gameobjects* themselves. When developing the UI in this manner, the first *gameobject* type that must be created is a UI *Canvas*, essentially serving as a container to hold all other UI elements. The canvas element itself contains a component called *Canvas Scaler* (Figure 3.5) which by settings it's associated *UI Scale Mode* to the value of “Scale With Screen Size”, allowed us to enter a reference screen width and height that the UI will be built from and then if this resolution is changed (used on a different device) then the sizing of all elements contained within the *Canvas* will also be changed by the correct proportions.



```
▼ Sorting*
    Main Camera
  ▼ Canvas
    ▼ MainMenu
      ▼ ControlsPanel
          PlayPauseButton
          SettingsButton
          ResetButton
        ▼ Panel
          ▶ DelaySlider
            SpeedText
            SlowText
            FastText
      ▼ SettingsPanel
          Text (TMP)
          Button
        ▶ MinValueSlider
          MinValueText
          MaxValueText
        ▶ MaxValueSlider
          NumOfBarsText
        ▶ NumOfBarsSlider
        ▶ SortTypeDropdown
          SortTypeText
    EventSystem
    BarsHandler
    SliderHandler
    CodeDisplay
```

Figure 3.4: Gameobject-based UI elements for the various settings menus


Canvas Scaler

UI Scale Mode: Scale With Screen Size

Reference Resolution X 1920 Y 1080

Screen Match Mode: Match Width Or Height

Match: 0.5

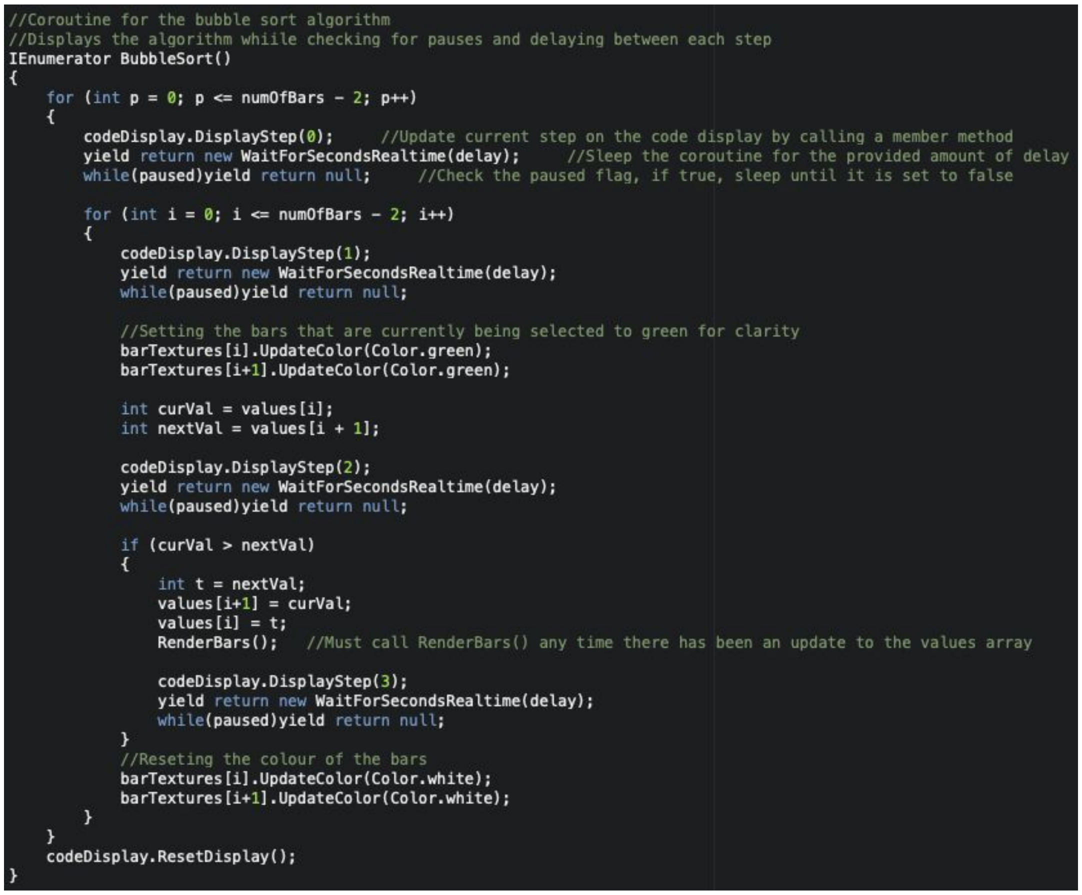
Reference Pixels Per 100

Figure 3.5: Canvas Scaler settings on main UI Canvas

With that step complete, that marks half of the work necessary for the UI to behave responsively at different resolutions. The next step involves ensuring that UI elements remain in their correct locations after which a change in resolution is made. The way that this is handled in Unity is by setting anchors on every element to indicate whether it should be either positioned relative to a specific side of a parent UI element, or if instead the UI element should either grow or shrink horizontally or vertically. For example, in the case of the buttons, we do not want them to change size as they could end up becoming too small to be used on smaller device sizes, so the buttons have been simply anchored to either the side or the bottom of the settings panel. In the case of the sliders, however, they take up a lot of room horizontally so in order to not end up overlapping UI elements, the slider for the delay has been set to anchor itself to the bottom of the panel and also grow/shrink horizontally relative to the size of the display.

3.4 Applying the Sorting Algorithms

Finally, as it might have been expected, one of the most important implementations of the algorithm visualizer tool is the implementation of the algorithms themselves. The logic behind the implementation of the sorting algorithms is located in *Sortings.cs* (alongside the implementation of the bars). Each sorting algorithm is created inside of its own IEnumerator method, treating it as a coroutine in order to yield the thread when necessary (pausing the simulation, allowing for step-by-step execution). Once a coroutine has been started, it has a number of tasks that it completes as clearly it is not as simple as just running the sorting algorithm. At any given point during the execution of the selected sorting algorithm, the coroutine must deal with changing the colour of the bars, re-rendering the bars once a value has been changed, yield the coroutine after each step, and check/handle pauses (Figure 3.6).



```
//Coroutine for the bubble sort algorithm
//Displays the algorithm whiile checking for pauses and delaying between each step
IEnumerator BubbleSort()
{
    for (int p = 0; p <= numOfBars - 2; p++)
    {
        codeDisplay.DisplayStep(0);     //Update current step on the code display by calling a member method
        yield return new WaitForSecondsRealtime(delay);     //Sleep the coroutine for the provided amount of delay
        while(paused)yield return null;     //Check the paused flag, if true, sleep until it is set to false

        for (int i = 0; i <= numOfBars - 2; i++)
        {
            codeDisplay.DisplayStep(1);
            yield return new WaitForSecondsRealtime(delay);
            while(paused)yield return null;

            //Setting the bars that are currently being selected to green for clarity
            barTextures[i].UpdateColor(Color.green);
            barTextures[i+1].UpdateColor(Color.green);

            int curVal = values[i];
            int nextVal = values[i + 1];

            codeDisplay.DisplayStep(2);
            yield return new WaitForSecondsRealtime(delay);
            while(paused)yield return null;

            if (curVal > nextVal)
            {
                int t = nextVal;
                values[i+1] = curVal;
                values[i] = t;
                RenderBars();   //Must call RenderBars() any time there has been an update to the values array

                codeDisplay.DisplayStep(3);
                yield return new WaitForSecondsRealtime(delay);
                while(paused)yield return null;
            }
            //Reseting the colour of the bars
            barTextures[i].UpdateColor(Color.white);
            barTextures[i+1].UpdateColor(Color.white);
        }
    }
    codeDisplay.ResetDisplay();
}
```

Figure 3.6: Code snippet from *Sorting.cs* with example of the coroutine used for the Bubble Sort algorithm

Due to the nature that each sorting algorithm operates in slightly different ways to achieve the same result, the way that they incorporate all of the above steps can (and are) be implemented in different orders. Expanding slightly on how each of the steps has been implemented, first off, at any point if there is a change in the visible state of either the bars or the code display, it must be followed up with the combination of yielding the coroutine for the given delay amount and checking to see if the algorithm has been paused. At any point during the execution of the algorithm, the user is able to hit the spacebar or click the pause button in order to trigger a pause, which on the backend triggers the *paused* flag to be set. If the check of the paused flag is true, the thread will yield indefinitely until the flag has been set back to false. Whenever a new section of the algorithm is entered, a call is made to the referenced CodeDisplay in order to increase the currently highlighted code segment(s) to the appropriate one(s). Finally, a popular convention that is commonly used among similar programs is to highlight the current bar that is being used for comparisons, and such a process has been implemented in this system as well. For example, in the case of the bubble sort algorithm it will change the colour of the two bars used for the comparison of the adjacent bars to green, and in the case of the selection sort algorithm, it will make the current pivot point green as well as each bar that it compares it to.

4. Results

4.1 Does it achieve the goal?

The main objective of this thesis was to create a tool that could be utilized by students in conjunction with their coursework, to assist with the learning/application of sorting algorithms. Unfortunately, with no formal method of conducting user testing available with the creation of this thesis, there can be no quantitative assessment to validate the effectiveness of the application. With that in mind we can however compare the resultant application to the initial goal of the thesis, and by that margin judge the effectiveness that the tool could potentially have. With the goal being to create a tool that could effectively display various sorting algorithms in a visually appealing and digestible way, I believe I am able to confidently say that this goal has been met. Since the tool allows a student to both step through sorting algorithms as well as provide the visual display of pseudocode, we can assume based on the research conducted that the student will gain pedagogical value from the tool as well as be more inclined to want to learn the algorithms.

4.2 Limitations of the System

Due to the nature of how the current method of applying the sorting algorithms are implemented, it is difficult to incorporate sorting algorithms that require recursive function calls. An example of sorting algorithms that would require recursive function calls could be either merge sort or quick sort, as they both utilize divide and conquer techniques which are generally implemented by using recursive function calls. The reason that it cannot be easily implemented in the system is that the current sorting algorithms are all implemented in an *iEnumerable* interface, which is a C# interface that allows for the creation of coroutines that can be yielded. The reason we require the coroutines is so we can pause the thread at given times in order to either pause the visualization or when a change from a previous step is being displayed. Based on my knowledge and research of coroutines in C#, it would not be worth re-factoring the entire sorting system as there seemed to be no better alternative that could allow for the yielding of threads and recursive calls.

5. Conclusion & Future Work

5.1 Future Work

When this thesis was first proposed, along with the proposed (and now implemented) idea to display sorting algorithms, was also the idea to display pathfinding algorithms. Due to time restraints, this feature was not able to be implemented into the final application but is most certainly something that could be added in the future. By utilizing Unity's scene management, a new scene could be added for the creation of a pathfinding algorithm display (or any other type of algorithm display), which would then allow the navigation between different types of algorithm visualizations. Implementing a pathfinding algorithm visualizer could be implemented by creating a 2D grid of nodes, which could be either *gameobject's* or UI elements. The grid display would function by allowing the user to first place a start and end node on the grid by clicking with the mouse. After these points have been placed, if the user clicks any of the nodes in the grid it will create a "wall" node, thus allowing the user to create a custom maze of obstacles for the algorithm(s) to run against. Since Unity already has the built-in functionality to handle mouse click events on *gameobjects* and UI elements, such a task would be more than feasible.

After creating the entire system, it became apparent after reading through the Unity documentation and Unity forum posts that the IMGUI system is meant to be used to create interactable components for developer testing. Although there is nothing inherently wrong with using them for the main user interface, that is not exactly what they are intended to be used for. For this reason, in the future, it would be advantageous to go back and re-implement all of the IMGUI elements with a *gameobject* based UI element. This will allow for the simplification of even more future improvements like the addition of animations, as IMGUI elements are a lot more difficult to animate than their *gameobject* based UI alternative. On top of that, the *gameobject* based UI is also more efficient at rendering to the screen, due to the fact that the IMGUI elements get re-calculated every frame in the associated *OnGUI()* Unity method. Of course, efficiency was not a top priority for this thesis as the design was more of a priority along with the fact that the system itself is not overly computationally complex.

Finally, the last main area of improvement that stands out for the application could be the addition of an option to allow the user to change the programming language that the code display uses. Currently, the code display only displays a pseudocode version of the selected algorithm, but it would not be too big of a challenge to incorporate the option to select the programming language of choice. This would involve adding some type of input field to retrieve the desired programming language (best suited would most likely be a dropdown), and then on top of holding the currently selected algorithm, the code display class could also be adjusted to keep track of the desired programming language and store each of the algorithms in every available programming language. This change would assist students that want a more 1:1 experience with the algorithm that they have been given to learn as it would allow them to preview it in the language that they are being taught (which in most cases is probably either Java or Python).

5.2 Conclusion

In conclusion, the goal of this thesis was to provide students with a tool that could be utilized to help increase their performance in the learning and comprehension of sorting algorithms across multiple platforms, and this algorithm visualizer reaches that goal. The choice to implement the tool in Unity has allowed for the simple multi platform development which has been optimized to ensure that the learning of the algorithms will not be compromised due to issues with the software not functioning or displaying correctly and efficiently across all platforms. With no other popular algorithm visualization tools being provided in as robust of a package as is done with this thesis, we have given the students the ability to utilize the tool on whichever medium they feel suits their learning style the most.

Due to the fact that this algorithm visualizer would be distributed to student's devices locally, there is no need for a constant internet connection to use the application which tackles a common problem facing certain algorithm visualization tools that do not offer offline variants (Yim et al, 466). Although no formal testing on the impact of the tool could be conducted, it is clear based on prior research that an algorithm visualization tool developed in the way that this one has been, has been proven to provide a positive effect on a students retention of the material as well their engagement with the content.

Thus, for computer science students who are struggling with grasping the concepts of algorithms and losing momentum in their courses, the utilization of this tool would most certainly provide a positive impact on both their current and future performance.

References

Basnet, Ram B., et al. "Exploring bimodality in introductory computer science performance distributions." EURASIA J. Math. Sci. Technol 14 (2018): 10

Bennedsen, Jens, and Michael E. Caspersen. “Failure Rates in Introductory Programming.” *ACM SIGCSE Bulletin*, vol. 39, no. 2, 2007, p. 32., doi:10.1145/1272848.1272879.

Doucette, John. “High Failure Rates in Introductory Computer Science Courses: Assessing the Learning Edge Momentum Hypothesis – John Doucette, CUT Student .” *Centre for Teaching Excellence Blog*, 10 Sept. 2015, cte-blog.uwaterloo.ca/?p=5153.

Felder, Richard M., Gary N. Felder, and E. Jacquelin Dietz. "The effects of personality type on engineering student performance and attitudes." Journal of engineering education 91.1 (2002): 3-17.

Gangwer, Timothy. Visual Impact, Visual Teaching: Using Images to Strengthen Learning. Skyhorse Pub., 2015.

Lewis, Michael, and Jeffrey Jacobson. "Game engines." Communications of the ACM 45.1 (2002): 27.

“The Definitive Glossary of Higher Mathematical Jargon.” *Math Vault*, 06 Mar. 2020, mathvault.ca/math-glossary/#algo.

Mudner, T., and E. Shakshuki. “A New Approach to Learning Algorithms.” International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004., 2004, doi:10.1109/itcc.2004.1286440.

Patitsas, Elizabeth, et al. "Evidence that computer science grades are not bimodal." Communications of the ACM 63.1 (2019): 91-98.

Peckham, Eric. “How Unity Built the World's Most Popular Game Engine.” TechCrunch, TechCrunch, 17 Oct. 2019, techcrunch.com/2019/10/17/how-unity-built-the-worlds-most-popular-game-engine

Saraiya, Purvi, et al. "Effective features of algorithm visualizations." Proceedings of the 35th SIGCSE technical symposium on Computer science education. 2004.

Sarpong, Kofi Adu-Manu, John Kingsley Arthur, and Prince Yaw Owusu Amoako. "Causes of failure of students in computer programming courses: The teacher-learner Perspective." International Journal of Computer Applications 77.12 (2013).

Shabiralyani, Ghulam, et al. "Impact of Visual Aids in Enhancing the Learning Process Case Research: District Dera Ghazi Khan." Journal of education and practice 6.19 (2015): 226-233.

Shaffer, Clifford A., et al. "Algorithm visualization: The state of the field." ACM Transactions on Computing Education (TOCE) 10.3 (2010): 1-22.

Stephanie. “Bimodal Distribution: What Is It?” *Statistics How To*, 29 July 2013, www.statisticshowto.datasciencecentral.com/what-is-a-bimodal-distribution/.

Tate, Thayer. “How Long Does It Take To Build Custom Software?” *SOLTECH*, SOLTECH, 15 Nov. 2019, soltech.net/how-long-does-it-take-to-build-custom-software/.

Te Brake, Guido, et al. "Developing adaptive user interfaces using a game-based simulation environment." Proceedings of ISCRAM (2006).

Yim, Ketrina, et al. Computer Science Illustrated: Engaging Visual Aids for Computer Science Education. 2009.