DC Circuit Simulator

Contents

[Analysis 2](#_Toc131111385)

[1. Background and problem 2](#_Toc131111386)

[2. Users and their needs 3](#_Toc131111387)

[2.1. End users 3](#_Toc131111388)

[2.2. Interview 3](#_Toc131111389)

[2.3. User requirements 4](#_Toc131111390)

[3. SMART Objectives 5](#_Toc131111391)

[3.1. MVP (Minimum Viable Product) 5](#_Toc131111392)

[3.2. Extension 5](#_Toc131111393)

[4. Modelling the problem 5](#_Toc131111394)

[Design 7](#_Toc131111395)

[1. Overall System Design 7](#_Toc131111396)

[1.1. Front End 7](#_Toc131111397)

[1.2. Back End 8](#_Toc131111398)

[2. System Design 9](#_Toc131111399)

[3. Pseudocode and algorithms 10](#_Toc131111400)

[3.1. Linked list 10](#_Toc131111401)

[3.2. Graph 15](#_Toc131111402)

[3.3. Loop detection 16](#_Toc131111403)

[3.4. Circuit manipulation 17](#_Toc131111404)

[3.5. Physics simulation 18](#_Toc131111405)

[4. Class Definitions 18](#_Toc131111406)

[Solution 19](#_Toc131111407)

[1. Attempt one 19](#_Toc131111408)

[2. Connecting components 25](#_Toc131111409)

[3. Attempt two 27](#_Toc131111410)

[3.1. Initialising Project 27](#_Toc131111411)

[3.2. File Management 29](#_Toc131111412)

[3.3. The Code 29](#_Toc131111413)

[Testing 40](#_Toc131111414)

[1. Tests 41](#_Toc131111415)

[2. Jasmine specs 43](#_Toc131111416)

[2.1. WireSpec.js 43](#_Toc131111417)

[2.2. BatterySpec.js 44](#_Toc131111418)

[2.3. LeverSpec.js 44](#_Toc131111419)

[2.4. NodeSpec.js 45](#_Toc131111420)

[2.5. LinkedListSpec.js 46](#_Toc131111421)

[2.6. ComponentSpec.js 49](#_Toc131111422)

[2.7. GraphSpec.js 49](#_Toc131111423)

[2.8. CircuitSpec.js 51](#_Toc131111424)

[Evaluation 52](#_Toc131111425)

[1. Feedback 52](#_Toc131111426)

[2. What went wrong 52](#_Toc131111427)

[3. What I would do next 53](#_Toc131111428)

# Analysis

## Background and problem

St Peters is a local secondary school that covers Key Stage 3 and Key Stage 4 education for students aged 11 – 16 as they prepare for their GCSEs at the end of Year 11. They are a Church of England school located in Exeter and educates over 1000 students.

Elliot is a Year 10 student at this school with various learning difficulties, including dyslexia, that make it difficult for him to read out of a textbook or sit down and concentrate for long periods of time on work and often requires movement breaks to make school more manageable. At the moment he is struggling to grasp the various topics as part of his GCSE Physics curriculum more specifically electricity and circuits. He prefers to be able to get hands on and learn by doing along with visual examples rather than learning out of a book, of which he finds boring and tiresome, so he is currently looking for a tool that will allow him to have more of an intuitive and visual understanding of circuits and how they function. Especially, parallel and series circuits specifically.

In order to make these concepts more digestible I will be making a DC circuit simulator in the form of a website. Where the user is able to get hands on experience tinkering and constructing various circuits from a list of components making the learning more interactable and hopefully more enjoyable. The website will also be able to provide visual feedback of the various properties of each component such as: resistance, amperage, and potential difference in order to provide a visual display of how the components and circuits interacts with the end goal of making the topic more intuitive.

Learning about electricity and circuits can be challenging for many students, particularly those who find it difficult to understand abstract concepts from a textbook. This is where the DC circuit simulator, an interactive learning tool, can be incredibly useful. By creating a website that allows students to experiment with circuits in a hands on way will allow students like Elliot to unlock a natural curiosity and fascination with the topic potentially impacting their choices and encouraging a more academic route towards further education.

One of the most difficult aspects of Physics as a subject is the fact that a lot of topics can be very abstract and hard to visualise. This is where a simulator can be particularly useful. The advantages of a website simulator over a physical circuit is that by providing students with the ability to see each components internal properties they will be able to get a deeper understanding of the relationships between amperage, resistance and potential difference rather than having to interpret equations. In addition the simulator provides a safe learning environment and allows students to see how the changes they make to their circuits affects the circuit as a whole. This will be very beneficial to not just students that struggle with abstracts concepts and reading out of a textbook, but also encourage students who are academically strong to be more curious and engage with the content even more. This is because students learn best when they are passionate about the subjects they’re studying.

## Users and their needs

### End users

The end users will be accessible to anyone who wishes to use it but is specifically targeted at any students and potentially teachers wanting to use the simulator as an additional tool to enhance the learning of GCSE electricity and its concepts. These students may have a variety of unique preferences to how they learn, the style they learn and different levels of their own knowledge when approaching the topic. In order to make the simulator website engaging and effective for a range of different students, it may be necessary to consider different levels of difficulty or complexity for the circuits, as well as providing different resources or explanations to support the simulator so that the students understand what exactly the simulator is teaching them. For example, it’s one thing to see a wire have 5 amps but the student may not know what 5 amps is.

### Interview

Q: What do you like about the current system with regards to learn the electricity topic?

A: It’s simple and you can experiment with things to see if they work.

Q: What do you find most difficult when learning about electricity and circuits?

A: The broad abstract concepts and having to read textbooks with dyslexia.

Q: What do you not like about the current system?

A: Not clear what the components are or how to use menus.

Q: What are the most important features you would like to see in the new system?

A: The ability to inspect each component within the circuit.

Q: What is your preferred way to learn?

A: Visually and practically and being read things allowed.

Q: Do you think that the new system will help with learning about electricity?

A: Yes.

Q: Are there any features you want to see in the new system?

A: Audio and explanations of why things would or wouldn’t work.

Q: Are there any other comments or ideas for the new system?

A: No.

### User requirements

In order for the new system to be implemented successfully there are various design choices that need to be considered. To meet the needs of the user the following requirements should be considered:

The backend of the simulator: the website should be designed in a way that allows for easy modification and addition of new components and features through the integration of various modular aspects of the code. The simulator should be able to handle different types of circuits, that allow users to experiment with different combinations of components. Additionally, the backend should be designed in a way that allows for easy maintenance and potential updates to the website.

User interface: the user interface of the simulator should be design to be intuitive and easy to use. Users should be able to select and drag components from a menu, and connect them together on a grid to form a circuit. Additionally, the user interface should provide real time feedback on the properties of each component, such as resistance, amperage, and potential difference, so that users can see the effect of their changes in real time.

Menu options: The simulator should provide a range of menu options that allow them to select different components and customise their circuits. For example, users should be able to choose between batteries, lightbulbs, wires, and switches. Additionally, the simulator should provide menu options that allow users to change the voltage of the battery, and individual components resistances to see how those effect the current of the circuit.

Visual design: the visual design of the simulator should be clean, modern, and easy to navigate. The website should be designed with the end user in mind during the whole of development, and should provide a visually engaging appealing experience that encourages user experimentation with circuits hopefully making learning more fun.

Accessibility: the website should be designed to be accessible for users with different levels of knowledge and different learning styles.

## SMART Objectives

## MVP (Minimum Viable Product)

* + 1. There should be circuit components.
       1. There should be a battery.
       2. There should be a wire.
       3. There should be a lever (switch).
    2. Circuits should be made up of connected components.
       1. Circuit manipulation should be done through manipulating a graph data structure.
       2. Connecting components should be done through manipulating a linked list data structure.
    3. Circuits should be displayed on a grid on a HTML page.
       1. There should be menus in order to select components and place them on the grid.
    4. Circuit components should have properties: resistance, amperage, potential difference.
    5. Components properties should be manipulated dependent on components they’re connected to and with accordance to calculations and physics equations.
       1. Algorithms to detect if the circuit is a closed loop.
       2. Appropriate algorithms to determine resistance and potential difference and amperage of each component.
       3. Simulate the flow of current through the circuit by using a directed graph.
    6. Components properties should be displayed on a HTML page in an intuitive way.

## Extension

* + 1. There should be other components such as: lightbulbs, capacitors, variable resistors and diodes each with their own corresponding properties.
    2. The website should have an intuitive styled design using CSS.
    3. Components properties should be stored and retrieved in a database (MongoDB atlas) and manipulating components properties directly links to manipulating the values within the database.
    4. The user should be able to log in and save circuits to their account within the database.

## Modelling the problem

The current system that the new system will be improving upon is from the website:

<https://www.physicsclassroom.com/Physics-Interactives/Electric-Circuits/Circuit-Builder/Circuit-Builder-Interactive>

A picture containing chart

Description automatically generatedThe current system has a basic interface where you can select the different components to be placed on the lines of the grid. The current system provides the following components: wire, resistor, filament lightbulb, and ammeter. As you can see in the image on the left, none of the components have any labels. This makes it difficult to know what each component is making it harder to search it up and learn more about it and learn how it functions, unless you already know what it is. For example, I did not know what the component on the far right was until I randomly stumbled across a tool that identifies what each component is. However, this tool is in my opinion unintuitive to use and was hidden under an unrelated menu.

Diagram

Description automatically generated with medium confidenceChart

Description automatically generated with medium confidenceThe next menu initially, I thought was simply a menu to change the voltage of the battery. However, after some messing around I’ve realised that it is a menu to adjust the variables at each component. As you can see in the images on the left. I believe that this is unitivitive and a poor design but could be easily fixed with good form design and labelling which I hope to address in my new system.

A picture containing diagram

Description automatically generatedThe last menu provides a voltmeter and ammeter that can be placed at the junctions of the grid to provide a reading. However, the ammeter in this menu is easy to accidentally place in parallel to the circuit which can provide the wrong values for circuit. This may end up confusing students using the current system.

One thing that the current system does well is that at each junction is states the potential difference at that junction. There is also animations of charge flowing around the circuit which can provide an intuitive understanding of the conventional way to imagine current flowing around a circuit. As well as this, the filament light bulbs provide a light when current is flowing through them. The light also increases and decreases depending on the amount of the current flowing through the lamp.

In the new system it may not be possible to include all these niceties and animations due to time constraints and the other functionality that I hope to include. If there is time these additional animations may be attempted to be added as they are a good way to make the electricity topic more visually intuitive for the students.

# Design

The design below is the design of the final system. Within the technical solution section there were other prototypes that were developed but did not make it into the final design so are not included here.

## Overall System Design

The overall system will be built on a node.js app composed of a frontend and a backend. I chose node.js as it is a very popular server environment that many websites are built off of so would have a lot of information online making research to complete the project a lot easier. Node.js creates a backend a JavaScript runtime environment that allows for JavaScript code to be executed outside a web browser within a server environment.

### Front End

The new system will be made up of a frontend and a backend. The frontend will be the part of the system that the end user actually interacts with. The frontend will be a website that can be accessed by anyone. The new system will have a list of components on the left-hand side of the screen. There will be a grid in the centre that takes up most of the screen this will be the sandbox area for the circuits to be built. The frontend will be made up of HTML, CSS, and JavaScript. JavaScript in the front end will be used to merge the frontend to the backend by taking in user interactions and passing them to the backend to be processed. As well as taking instructions from the backend and displaying them on the frontend. For example, a user may place a component onto a grid. The frontend JavaScript will take the mouse input and pass it to the backend in order to add the component onto the circuit in the correct place then tell the frontend to display an image of the component in the correct place on the circuit builder.

Below is a diagram of what the front end could potentially end up looking like.

Table

Description automatically generated

### Back End

The backend of the circuit simulator has been developed using a node.js server utilising the express framework. The decision to use node.js and express was made because they provide a scalable and modular approach to web development that makes it easy to create fast and efficient web applications. As well as being a very popular server environment allowing researching development for the project to be faster.

If there is time with the project, I hope to include a database that is able to store users and circuits to the database to be accessed later. The database I have chosen to use is MongoDB which is document oriented database that uses a JSON like query language rather than SQL. I chose specifically to use MongoDB Atlas which is a cloud-based database service. This is was done because MongoDB Atlas is easier to maintain and use than a locally stored database because a lot of the work is done for you on the cloud. MongoDB was chosen specifically as my database of choice because it allows for easy storage of JavaScript class objects unlike SQL based databases where I would need to deconstruct each object into each property, do a query per property and then need to reconstruct each property once retrieved from the database in order to be used within the program again.

In addition to node.js and express, I also chose to utilise a number of packages in order to make backend development easier. These packages include CORS, dotenv, and Mongoose. CORS (Cross Origin Resource Sharing) is a security feature that allows web servers to specify which origins are allowed to access their resources. This is an important feature for the simulator because it allows the frontend of the website to communicate with the backend without any security issues.

The dotenv package is used to manage environment variables more specifically the MongoDB Atlas connection link. It is important to store sensitive information, such as the connection link to the database, securely. The dotenv package provides a convenient way to do so.

Mongoose is an Object Data Modelling (ODM) library for MongoDB and node.js. It provides a straightforward, schema based solution to model class objects data in a structed format. This is particularly useful for the simulator because it allows for easy storage and retrieval of circuit data.

Finally, the backend utilises the Jasmine testing framework. Jasmine is a popular testing framework for node.js applications that allows developers to write tests for their code and ensure that it functions as expected. The use of Jasmine ensures that the backend of the simulator is robust and reliable, and allows for the ability to catch any potential bugs early in development and test each part of the simulator as its developed. More can be seen about this in the Testing part of the document.

## System Design

The backend of the system will be broken down into classes for different components for example a wire class and a battery class. Every component inherits from the wire class as it is the most basic type of component. The wire class is an object that has the three physics properties: amperage, resistance, and potential difference.

A problem that ended up taking a long time to solve was exactly how to actually represent the circuit in code in such a way that it could be manipulated. As in adding components and connecting them together to form a loop so that one component was dependent on the other in order to simulate the flow of charge and how to remove components from the circuit in an efficient modular way that can be done during runtime. Multiple solutions were attempted to solve this as seen in attempt 2 and 3 of the technical solution. However, in the design as mentioned I will only discuss the final solution for the project.

In the end the solution I came up with was to represent circuits as a graph data structure something that we had learned about in class. By doing so it allowed me to store components as vertices in a class and connections between nodes as edges as well as opening up a wide array of online resources that I previously didn’t have access to in order to solve the various problems with developing a circuit simulator. Such as how to detect if the circuit was a loop and could be turned on. In order to do this, you are able to perform a depth first search on the graph where there is a stack of already visited nodes and if a neighbour of a node being explored is already on the visited stack a cycle has been found.

Below is an example of a circuit represented as a graph.

Diagram

Description automatically generated

Another advantage of using a graph to represent the circuit is that in order to simulate the flow of charge is as simple as making the graph directed as seen below.

Diagram

Description automatically generated

When the circuit is turned on the physical properties from the battery will be calculated such as current and potential difference (resistance is dependent on each individual component as cannot be changed by other components) and then is passed onto the next component in the circuit performing any calculations necessary on the way. For example, if the wire has a resistance it will be lower the current being passed onto the lightbulb.

Due to the volatile nature of the graph as the circuit will constantly being changed during runtime of the simulation. I chose to develop the graph using linked lists for the adjacency lists as opposed to arrays to avoid any holes in the array when components are inevitably removed from the circuit.

## Pseudocode and algorithms

### Linked list

A linked list is a data structure that links nodes together using a node and a pointer to the next node in the list. This gives the ability to remove any node from the list and not create a hole within the list like you would with an array and have empty indexes. The images below show the difference between an array and a linked list when removing a letter from the word Happy.

Graphical user interface

Description automatically generated with medium confidence

Graphical user interface

Description automatically generated with low confidence

In order to have a useful linked list data structure we need the node and pointer and a few methods to manipulate nodes such as:

* insert first (inserts a new node at the start of the list)
* insert last (inserts a new node at the end of the list)
* insert at (inserts a new node at a specific index)
* get at (returns the node at a specific index)
* remove at (deletes a node at a specific index)
* remove (deletes the specific node targeted)
* clear list (deletes the list and all its contents)
* search (returns node that is being searched)
* print (because its not a built in JavaScript data structure you are unable to easily console log the list)

Here is the pseudocode of the most complicated methods:

FUNCTION insertLast(data):

CREATE a new Node with the given data and assign it to a variable called "node"

IF the list is empty:

SET the "head" property of the list to "node"

ELSE:

SET a variable called "index" to the current head of the list

WHILE the "next" property of "index" is not null:

SET "index" to the "next" property of "index"

END WHILE SET the "next" property of "index" to "node"

END IF

INCREMENT the length of the list by 1

END FUNCTION

FUNCTION insertAt(data, index):

IF the given index is greater than 0 and greater than or equal to the length of the list:

RETURN

END IF

IF the given index is 0:

SET the "head" property of the list to a new Node with the given data and the current head as its "next" property

ELSE:

CREATE a new Node with the given data and assign it to a variable called "node"

DECLARE two variables called "current" and "previous"

SET "current" to the current head of the list

SET a variable called "count" to 0

WHILE the value of "count" is less than the given index:

SET "previous" to "current"

INCREMENT "count" by 1

SET "current" to the "next" property of "current"

END WHILE

SET the "next" property of "node" to "current"

SET the "next" property of "previous" to "node"

END IF

INCREMENT the length of the list by 1

END FUNCTION

FUNCTION removeAt(index):

IF the given index is greater than 0 and greater than or equal to the length of the list:

RETURN

END IF

SET a variable called "current" to the current head of the list

DECLARE two variables called "previous" and "count" and set both to 0

IF the given index is 0:

SET the "head" property of the list to the "next" property of "current"

ELSE:

WHILE the value of "count" is less than the given index:

INCREMENT "count" by 1

SET "previous" to "current"

SET "current" to the "next" property of "current"

END WHILE

SET the "next" property of "previous" to the "next" property of "current"

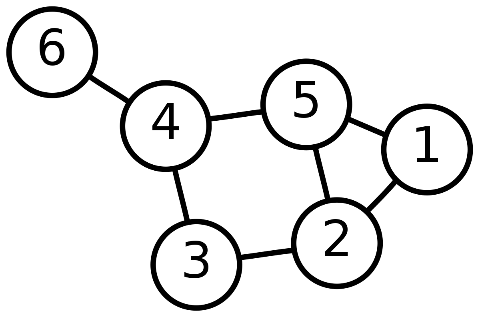
END IF

DECREMENT the length of the list by 1

END FUNCTION

### Graph

Graphs are a data structure that is used to represent various real-world problems such as road networks, social networks, even chemical molecules but I’ll be using it to represent electrical circuits. They consist of a set of nodes or vertices that are connected by edges. The nodes can represent any object in my case electrical components, while the edges represent the connections between them. An example of a graph is shown below.



When coding graphs you need an array of vertices and like the nodes for the linked lists the vertices consist of a data variable and normally an array of vertices it is connected too called an adjacency list. However, for my graph data structure I am using linked lists instead of arrays due to the volatile nature of my graphs in simulator. Each vertex also has a uniquely generated index in order to make each vertex unique as you may have multiple wires within the graph so they each individually need to be able to be identified. In order to have a useful graph data structure we will need the following methods:

* add vertex (give vertex a unique index and then add to vertices list)
* add edge (creates a directed edge between two vertices by adding index of one vertex to the other)
* remove vertex (removes all edges containing target vertex and then removing target vertices list)
* remove edge (removes the edge by removing one vertex to the other)
* print (because its not a built in JavaScript data structure you are unable to easily console log the list)

Here is the pseudocode of the most complicated method:

FUNCTION addVertex(vertex):

(the start of the algorithm finds a free index to assign the new vertex)

SET a variable called “freeIndex” to true

SET a variable called “index” to the length of the graph

WHILE the value of “freeIndex” is true:

FOR variable “i” smaller than length of the graph

IF index is equal to the index of the vertex in vertices list at i

SET “freeIndex” to false

END IF

END FOR

INCREMENT the index by 1

END WHILE

CREATE a new Component with the given index and vertex and add to the end of the vertices list

INCREMENT the length of graph by 1

END FUNCTION

### Loop detection

Loop detection for the circuit can be achieved using a depth first search algorithm on a directed graph. We do this by marking each visited vertex to a set called “onPath” to keep track of nodes that are currently on the path being explored. If we encounter a node that is already in the “onPath” set, then we have found a cycle in the graph.

To achieve this, we create two sets: “visited” and “onPath”. The “visited” set is used to keep track of all nodes that have been visited during the traversal, while the “onPath” set is used to keep track of nodes that are currently on the path being explored.

Then, we start a depth first search (DFS) traversal from each unvisited node in the graph. For each node we visit during the traversal, we add it to the "visited" set and the "onPath" set. We then explore all the neighbours of the node recursively, and if we encounter a neighbour that is already in the "onPath" set, we have found a cycle in the graph and we return true.

If we have explored all the neighbours of the current node and have not found a cycle, we remove the current node from the "onPath" set and return false. Finally, if we have completed the DFS traversal and have not found a cycle, we return false.

The advantage of using DFS for loop detection is that it has a time complexity of O(V+E), where V is the number of vertices and E is the number of edges in the graph. This makes it efficient for detecting loops in large graphs.

The “onPath” set is used in addition to the “visited” set in the DFS algorithm to detect cycles in a directed graph. While the “visited” set keeps track of all the nodes visited during the DFS traversal, the “onPath” set keeps track of the nodes that are currently on the path being traversed by the algorithm.

In a directed graph, it's possible to visit a node multiple times in different paths during a traversal without there being a cycle. However, if a node is visited and added to the “onPath” set and is later visited again while still on the path then a cycle has been detected.

Therefore, the “onPath” set is used to keep track of nodes that are currently being traversed, and to check if any of these nodes are visited again in the same path, indicating the presence of a cycle. Using only the “visited” set would not be sufficient for detecting cycles in a directed graph.

### Circuit manipulation

In order for the backend of simulator to manipulate the circuit, which is represented as a graph data structure, several methods are needed including the loop detection method. These methods will manipulate the graph by adding, modifying and vertices nodes and edges and perform various calculations with the electrical properties of vertices.

One of these methods is the add component method that will create a new vertex with the component that is currently selected then take in a coordinate of where the component is being placed on the grid. It will then look up what coordinates are adjacent to it see if there are any components in those coordinates and create the edges between the appropriate vertices in the graph. Then display an image of the component on the grid.

Select component method will take in a mouse input on a menu of components which changes a variable selectedComponent for when adding the component to the grid.

Remove component will remove the selected component from the grid from the screen as well as removing the vertex associated to the selected component from the graph.

A method for the physics simulation should constantly be updated checking the if loop detection method is true if so does the appropriate calculations and transfer of electrical properties to the vertices around the graph.

A method for when the loop is broken will be needed in order to reset the values of the components back to their default values.

### Physics simulation

The simulator will need to perform various physics calculations and equations in order to accurately simulate the behaviour of electrical circuits. Some of the key equations that need to be implemented in the simulator include:

* Ohm's Law: V = IR

Ohm's Law states that the voltage (V) across a component is equal to the current (I) flowing through it multiplied by the component's resistance (R). This equation is used to calculate the voltage across individual components in a circuit.

* Kirchhoff's Voltage Law: ΣV = 0

Kirchhoff’s Voltage Law states that the sum of the voltages around any closed loop in a circuit must be equal to zero. This equation is used to calculate the voltages across loops of components in a circuit.

* Kirchhoff's Current Law: ΣI = 0

Kirchhoff’s Current Law states that the sum of the currents entering and leaving a junction in a circuit must be equal to zero. This equation is used to calculate the currents flowing through junctions of components in a circuit.

* To calculate the resistance for a circuit in series, the total resistance (R) is simply the sum of the individual resistances of the components in the circuit: R = R1 + R2 + R3 + ...
* To calculate the resistance for a circuit in parallel, the total resistance (R) is calculated using the equation: 1/R = 1/R1 + 1/R2 + 1/R3 + ...

Extension equations and calculations to be included if there’s time:

* Power: P = VI

The power (P) dissipated by a component in a circuit is equal to the product of the voltage (V) across the component and the current (I) flowing through it. This equation is used to calculate the power can be used to calculate the amount of energy an individual component has used in a certain time thanks to the equations P = E/t.

* Capacitance: C = Q/V

The capacitance (C) of a capacitor is equal to the charge (Q) stored on the capacitor divided by the voltage (V) across it. This equation is used to calculate the capacitance of individual capacitors in a circuit if the capacitor component is added.

## Class Definitions

Below are class diagrams of most of the classes as mentioned in the design:

Diagram

Description automatically generated

# Solution

There were various prototypes throughout development that did not make it into the final project, so I have separated up the document to include each iteration of the technical solution.

## Attempt one

For the first attempt at the solution, I decided to use HTML5 canvas which is an element built into HTML5 that allows you to draw graphics and images dynamically on a web page using JavaScript creating interactive web pages. The canvas element is a rectangular area on a web page, with a defined width and height, where you can draw shapes, lines, text, and images using various graphics APIs provided by HTML5 and JavaScript. Canvas is similar to a built-in mini game engine in the web browser, however, is not very powerful.

This will allow me to draw a grid onto the canvas as well as have a menu where you are able to select a component and place it onto a cell within the grid. It will also allow me to perform physics calculations in real time and have an interactive graphical animation happen depending what is going on in the simulation. For example, perhaps a flow of current animation when the circuit is turned on.

To get started I created a HTML page and added some basic CSS to have a rough design page with a header with a temporary Menu and Log in button as well as a menu for the list of selectable components and a menu for the properties of the selected component in the circuit. I then added a canvas element to the page and add the simulator script that will interact with the canvas as well as:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<meta http-equiv="X-UA-Compatible" content="IE=edge" />

<meta name="viewport" content="width=device-width, initial-scale=1.0" />

<title>Simulator</title>

<script src="https://cdn.tailwindcss.com"></script>

<style>

.components-box {

box-sizing: border-box;

position: absolute;

width: 254px;

height: 940px;

left: 0px;

top: 84px;

background: #f5ee9e;

}

.properties-box {

box-sizing: border-box;

position: absolute;

width: 1186px;

height: 207px;

left: 254px;

top: 817px;

background: #f49e4c;

}

.header-box {

box-sizing: border-box;

position: absolute;

width: 1440px;

height: 84px;

left: 0px;

top: 0px;

background: #2d728f;

}

.canvas-box {

box-sizing: border-box;

position: absolute;

width: 1186px;

height: 733px;

left: 254px;

top: 84px;

background: #3b8ea5;

}

canvas {

border: 1px solid black;

}

</style>

</head>

<body>

<header class="flex justify-between header-box">

<button class="px-8">Menu</button>

<button class="px-8">Log In</button>

</header>

<div class="flex">

<div class="components-box"></div>

<div class="properties-box"></div>

<div class="canvas-box">

<canvas></canvas>

<script src="./simulator.js"></script>

</div>

</div>

</body>

</html>

Below is a screenshot of what the HTML page now looks like.

A picture containing chart

Description automatically generated

In order to setup the canvas the following boiler plate code is needed:

const canvas = document.querySelector("canvas");

const c = canvas.getContext("2d");

function innit() {

fitToContainer(canvas);

}

function fitToContainer(canvas) {

canvas.style.width = "100%";

canvas.style.height = "100%";

canvas.width = canvas.offsetWidth;

canvas.height = canvas.offsetHeight;

}

innit();

The constant variable canvas is assigned to the canvas element in the HTML page. The constant variable c is short for context and can be thought of as a pen you draw with on the screen, in my case the canvas will be dealing in 2D so I retrieve that context as canvas can also work in 3D. As you can see in the HTML page currently the canvas does not fit to the space dedicated to the grid so the fit to container function fixes this. Now the HTML page looks like this:

Shape

Description automatically generated

In order to draw a grid we need cells that can take in x, y coordinates and w for width of the cell then draw this to the screen to do this I created this class:

class cells {

constructor(x, y, w) {

this.x = x;

this.y = y;

this.w = w;

}

draw() {

c.moveTo(this.x, this.y);

c.rect(this.x, this.y, this.w, this.w);

c.stroke(); // draws on screen

}

}

Then by using this class and some predefined variables I am able to use a nested for loop to draw within the innit() function in order to draw a grid by making use of 2D arrays:

let cellsArray = [];

let rows = 2;

let columns = 2;

let padding = 10;

let cellWidth = 100;

function innit() {

fitToContainer(canvas);

for (let j = 0; j < rows; j++) {

for (let i = 0; i < columns; i++) {

cellsArray[i + j] = new cells(cellWidth \* i + padding, cellWidth \* j + padding, cellWidth);

cellsArray[i + j].draw();

}

}

}

With the predefined variables in this configuration the HTML page now looks like this:

Chart, treemap chart

Description automatically generated

As you can see the bottom and right side of the bottom right cell is more faint than the rest of the cells this is because how the algorithm is designed each rectangle is technically drawn twice going left to right top to bottom like how we read so the bottom right one is only drawn once so fainter. This is a bug that is never fixed due to abandoning this solution.

If we change the rows to 5 and the columns to 3 the HTML page now looks like this:

Chart

Description automatically generated

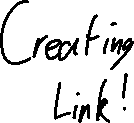
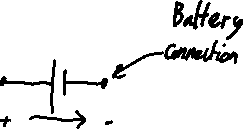
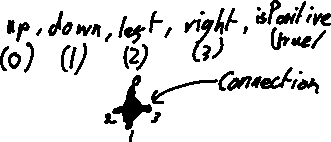
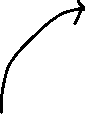
The next stage was to develop a grid class that would initialise the cells and draw the grid with components however during developing this class was when I decided to abandon this solution as the HTML5 would have made the development of the simulation very messy and would have turned into spaghetti code for lack of a better term. As well as not including a backend or the ability to hook up any kind of database which would have limited the simulators possible features dramatically.

## Connecting components

A problem that I struggled with immensely when developing my solution was how exactly to connect electrical components together and although this isn’t the solution, I ended up going with it took a lot of time to develop this system so this section is documenting that process.

The rough plan was to create node like entities on the grid called connectors that would have four connections up, down, left, and right. The connections would have a Boolean property called positive that would be used to simulate polarity across a component to aid in simulating the flow of charge around a circuit. A connection would also have a coordinate that would correspond to where it was on the grid. A component would then have two connections and connections with the same coordinates would link up connecting components.

Below is a drawn explanation:



By doing this it gave the ability to transfer electrical properties from the positive connection to the negative connection at the same coordinate or across components allowing for physics calculations to be made at each connection.

The code for these classes is shown below:

class Connection {

constructor(

isPositive,

current = DEFAULT\_CURRENT,

resistance = DEFAULT\_RESISTANCE,

potentialDifference = DEFAULT\_POTENTIALDIFFERENCE

) {

this.isConnected = false;

this.isPositive = isPositive; // represents connections polarity

this.current = current;

this.resistance = resistance;

this.potentialDifference = potentialDifference;

}

}

The default values for current, resistance, and potential difference are stored in a separate file called properties see Attempt two for more.

class Connector {

constructor(x, y) {

this.coordinate = {};

this.coordinate["x"] = x;

this.coordinate["y"] = y;

this.up = new Connection();

this.down = new Connection();

this.left = new Connection();

this.right = new Connection();

}

}

class Wire {

constructor(

current = DEFAULT\_CURRENT,

resistance = DEFAULT\_RESISTANCE,

potentialDifference = DEFAULT\_POTENTIALDIFFERENCE

) {

this.current = current;

this.resistance = resistance;

this.potentialDifference = potentialDifference;

this.positiveConnection = new Connection(true);

this.negativeConnection = new Connection(false);

}

}

Now that we this setup we need a circuit class to pull all the aspects together into a working circuit simulator. In order to do this a circuit would need to know certain things and have certain methods. A circuit would need to have a list of components currently in the circuit as well as a list of connectors with their respective coordinates in order to make up the grid. To do this we would need an initialising grid function that would instantiate all the connectors with their coordinates.

A method to add component to the circuit would need to instantiate the currently selected component as well as take in the coordinates of the components connections then build up the relationship between the component’s connections and the connectors connections. This means you would also need an algorithm to determine the orientation of the component from the x and y coordinates the component being instantiated whether the connections on the connectors are up down left or right.

As you can see this is a lot of work and getting incredibly complicated compared to using a graph to represent the circuit as mentioned in the design so once I had this breakthrough I decided to go with this more elegant solution.

## Attempt two

### Initialising Project

To initialise the project the following steps, need to be followed:

1. Open the command prompt terminal
2. Navigate to the correct directory
3. Then run command “npm init” and follow the steps to create a new node server this creates boiler plate code as well as a package.json file.
4. Then run a command that installs the following packages CORS, dotenv, express, Mongoose with the following command: “npm install cors dotenv express mongoose” (to see what these packages are refer to the design section)
5. Next create a index.html file with the standard HTML boiler plate code and a server.js file.
6. In the server.js file put the following code:

// imports the packages into the JavaScript project

const express = require("express");

const cors = require("cors");

const mongoose = require("mongoose");

require("dotenv").config();

// instantiating an express app object and opening up a node server on port 5000

const app = express();

const port = process.env.PORT || 5000;

// imports packages into server

app.use(cors());

app.use(express.json());

// imports connection URI for MongoDB Atlas database from dotenv and connects to database

const uri = process.env.ATLAS\_URI;

mongoose.connect(uri, {

useNewUrlParser: true,

useUnifiedTopology: true,

});

const connection = mongoose.connection;

connection.once("open", () => {

console.log("MongoDB database connection established successfully");

});

// setup a get request and respond data stream with index.html

app.get("/", function (req, res) {

res.sendFile("index.html", { root: \_\_dirname });

});

// opening up the port of the server

app.listen(port, () => {

console.log(`Now listening on port ${port}`);

});

1. Create a dotenv file to securely store the connection URI for the database:

ATLAS\_URI=mongodb+srv://cpalmer:OvBVJHQGMRVMR7yh@cluster0.x1qjsou.mongodb.net/?retryWrites=true&w=majority

1. Now install the Jasmine testing framework with “npm install jasmine” and then use “npx jasmine init” to create a jasmine project. This creates a spec file which is where all the tests are written and a configuration file.
2. Now you can create a file called src which is where all the JavaScript code for the simulator will be written and the project is fully setup.

### File Management

By now the file structure should look like this:

DC Circuit Simulator

node\_modules

spec

src

.env

index.html

package-lock.json

package.json

server.js

All tests will be written in the spec file and all the simulator code will be written in src in order to keep the project file clean and usable. All class JavaScript files will follow the typical naming conventions of starting with a capital for example “Class.js”. All spec test files will start with the name of the class it is testing and then the key word spec for example “ClassSpec.js”

### The Code

Throughout the project encapsulation is used as the files only have access to each other if they are imported or exported specifically this keeps the code cleaner and easier to work with as well as making the project as a whole more robust.

#### properties.js

The first file to be written was “properties.js” that contained all the default values for the components:

// default values for circuit components

const DEFAULT\_CURRENT = 0;

const DEFAULT\_RESISTANCE = 0;

const DEFAULT\_POTENTIALDIFFERENCE = 0;

const DEFAULT\_BATTERY\_POTENTIALDIFFERENCE = 5;

// exporting default values

module.exports = {

DEFAULT\_CURRENT,

DEFAULT\_RESISTANCE,

DEFAULT\_POTENTIALDIFFERENCE,

DEFAULT\_BATTERY\_POTENTIALDIFFERENCE,

};

#### Wire.js

Wire is the most basic circuit component and in terms of coding is only an object to store electrical properties current, resistance, and potential difference. In order to use the default values for these properties we will need to import them using “require()” and then to use the wire class in other classes we will need to export it by using “module.exports”:

// import necessary files

const {

DEFAULT\_CURRENT,

DEFAULT\_RESISTANCE,

DEFAULT\_POTENTIALDIFFERENCE,

} = require("./properties");

class Wire {

constructor(

current = DEFAULT\_CURRENT,

resistance = DEFAULT\_RESISTANCE,

potentialDifference = DEFAULT\_POTENTIALDIFFERENCE

) {

this.current = current;

this.resistance = resistance;

this.potentialDifference = potentialDifference;

}

}

// exporting wire class

module.exports = {

Wire: Wire,

};

#### Battery.js

A battery in terms of coding the component is exactly the same as a wire except that it has a non-zero value of a potential difference in the properties file we arbitrarily set our batteries default potential difference to 5V. So instead, we import the default potential difference specifically for a battery as well as the wire class:

// import necessary files

const { Wire } = require("../src/Wire");

const {

DEFAULT\_CURRENT,

DEFAULT\_RESISTANCE,

DEFAULT\_BATTERY\_POTENTIALDIFFERENCE,

} = require("./properties");

// battery circuit component inherits from wire component class

class Battery extends Wire {

constructor(

current = DEFAULT\_CURRENT,

resistance = DEFAULT\_RESISTANCE,

// batteries have a different default value for potential difference (emf)

potentialDifference = DEFAULT\_BATTERY\_POTENTIALDIFFERENCE

) {

super(current, resistance, potentialDifference);

}

}

// exporting battery class

module.exports = {

Battery: Battery,

};

#### Lever.js (switch)

Because switch is a keyword in JavaScript programming it could not be used a class name so lever was used instead. Like a battery a lever is just like a wire however it has two states closed or open. To represent this in code we use a Boolean called isOn and a method called onClick that will be activated by a click on the lever by the user will change the value of isOn closing the switch on screen:

// import necessary files

const { Wire } = require("../src/Wire");

const {

DEFAULT\_CURRENT,

DEFAULT\_RESISTANCE,

DEFAULT\_BATTERY\_POTENTIALDIFFERENCE,

} = require("./properties");

// battery circuit component inherits from wire component class

class Battery extends Wire {

constructor(

current = DEFAULT\_CURRENT,

resistance = DEFAULT\_RESISTANCE,

// batteries have a different default value for potential difference (emf)

potentialDifference = DEFAULT\_BATTERY\_POTENTIALDIFFERENCE

) {

super(current, resistance, potentialDifference);

}

}

// exporting battery class

module.exports = {

Battery: Battery,

};

#### Node.js

A node class is needed for the linked list class. It is a very simple class only requiring a variable of the data being put into the node and a variable to pointing to the next node in the list:

class Node {

constructor(data, next = null) {

this.data = data;

this.next = next;

}

}

module.exports = {

Node: Node,

};

#### LinkedList.js

The linked list class requires a head node where the list starts and a length variable to keep track of the length of the list. In order for the class to actually be useful several methods are needed to be developed refer to the design section for more information. Methods insertAt() and removeAt() have defensive programming built into the methods to be able to handle extreme situations and method search() is an example of a linear search:

const { Node } = require("./Node");

class LinkedList {

constructor() {

this.head = null;

this.length = 0;

}

// insert first node

insertFirst(data) {

this.head = new Node(data, this.head);

this.length++;

}

// insert last node

insertLast(data) {

const node = new Node(data);

if (!this.head) {

this.head = node;

} else {

let index = this.head;

while (index.next) {

index = index.next;

}

index.next = node;

}

this.length++;

}

// insert at index

insertAt(data, index) {

// if index is out of range

if (index > 0 && index >= this.length) {

return;

}

// if first index

if (index === 0) {

this.head = new Node(data, this.head);

} else {

const node = new Node(data);

let current, previous;

// set current to first

current = this.head;

let count = 0;

while (count < index) {

previous = current; // node before index

count++;

current = current.next; // node after index

}

node.next = current;

previous.next = node;

}

this.length++;

}

// get at index

getAt(index) {

let current = this.head;

let count = 0;

while (current) {

if (count == index) {

return current.data;

}

count++;

current = current.next;

}

return;

}

// remove at index

removeAt(index) {

// if index is out of range

if (index > 0 && index >= this.length) {

return;

}

let current = this.head;

let previous;

let count = 0;

// remove first

if (index === 0) {

this.head = current.next;

} else {

while (count < index) {

count++;

previous = current;

current = current.next;

}

previous.next = current.next;

}

this.length--;

}

// remove

remove(target) {

let current = this.head;

let count = 0;

while (current) {

if (current.data === target) {

this.removeAt(count);

}

count++;

current = current.next;

}

}

// clear list

clearList() {

this.head = null;

this.length = 0;

}

// linear search

search(target) {

let current = this.head;

while (current) {

if (current.data === target) {

return current.data;

}

current = current.next;

}

return;

}

// print list data

print() {

let index = this.head;

while (index) {

console.log(index.data);

index = index.next;

}

}

}

module.exports = {

LinkedList: LinkedList,

};

#### Component.js

The component class is another simple class that functions as the vertex of the graph class. A component object stores the index of the component which is necessary in order to keep track of components within the circuit as there will most likely be multiple of the same components in one circuit as well as a component variable that stores an instance of a component class and a listOfConnections variable that is an instance of a linked list. The listOfConnections variable is used as each vertex’s adjacency list for the graph. All methods have defensive programming in order to be able to handle all possible scenarios with the graph. Thanks to the linked list class the graph class was relatively simple thanks to the prior system already built robustly I didn’t need to worry about the linked list failing:

const { LinkedList } = require("./LinkedList");

class Component {

constructor(*index*, *component*) {

this.index = *index*;

this.component = *component*;

this.listOfConnections = new LinkedList();

}

}

module.exports = {

Component: Component,

};

#### Graph.js

The graph class requires a vertices list which is an instance of a linked list (refer to the design section for why I used a linked list and how more detail on how graphs work conceptual and the what the graph methods actually do) and a length to keep track of how many vertices are in the graph. A graph also requires various methods to be useful:

const { LinkedList } = require("./LinkedList");

const { Component } = require("./Component");

class Graph {

constructor() {

this.vertices = new LinkedList();

this.length = 0;

}

// add vertex

addVertex(vertex) {

// checks index is free to use if not increments index until free

let freeIndex = true;

let index = this.length;

while (!freeIndex) {

for (let i; i < this.length; i++) {

if (index == this.vertices.getAt(i).index) {

freeIndex = false;

}

}

index++;

}

this.vertices.insertLast(new Component(index, vertex));

this.length++;

}

// add edge

addEdge(vertex1, vertex2) {

// check that vertcies exist

if (!this.vertices.getAt(vertex1)) {

return;

}

if (!this.vertices.getAt(vertex2)) {

return;

}

this.vertices.getAt(vertex1).listOfConnections.insertLast(vertex2);

}

// remove vertex

removeVertex(vertex) {

// loops through each vertex

for (let i = 0; i < this.length; i++) {

// removes edges where target vertex is present

if (

this.vertices.getAt(i).listOfConnections.search(vertex) != undefined

) {

this.vertices.getAt(i).listOfConnections.remove(vertex);

}

}

// now all edges of target vertex gone now remove vertex from vertices

this.vertices.remove(this.vertices.getAt(vertex));

}

// remove edge

removeEdge(vertex1, vertex2) {

// check if edge exists

if (!this.vertices.getAt(vertex1).listOfConnections.search(vertex2)) {

return;

}

// remove vertex 2 from vertex1 adjacency list

this.vertices.getAt(vertex1).listOfConnections.remove(vertex2);

}

// print graph

print() {

for (let i = 0; i < this.length; i++) {

console.log(this.vertices.getAt(i));

}

}

}

module.exports = {

Graph: Graph,

};

#### Circuit.js

The circuit class is where everything comes together. I have talked in more depth in the design section but it would basically link the frontend to the backend by taking inputs from the user process it in the backend and then display the output on the screen. The main function of the class is to manipulate the graph representation of the circuit and run the physics simulation. The x and y arrays in the constructor are for mapping the circuits components to a x and y coordinates a grid that would be displayed to the user. One function that is required is to detect a loop in the graph (circuit) in order to turn on the circuit and start the physics simulation this is done using a depth first search:

const { Graph } = require("./Graph");

class Circuit {

constructor() {

this.graph = new Graph();

this.x = [];

this.y = [];

}

// select component

// add component

// coordinate connection

// remove component

// turn on circuit (physics simulation)

// turn off circuit

// loop detection

hasLoop() {

const visited = new Set();

const onPath = new Set();

// parsing vertex == vertexs index not component object

function depthFirstSearch(vertex, graph) {

visited.add(vertex);

onPath.add(vertex);

graph = graph;

for (let i = 0; i < graph.length; i++) {

let neighbour = graph.vertices.getAt(i).index;

if (onPath.has(neighbour)) {

// a cycle has been found

return true;

}

if (!(visited.has(neighbour)) && depthFirstSearch(neighbour)) {

// a cycle has been found

return true;

}

}

onPath.delete(vertex);

return false;

}

for (let i = 0; i < this.graph.length; i++) {

let vertex = this.graph.vertices.getAt(i).index;

if (!(visited.has(vertex)) && depthFirstSearch(vertex, this.graph)) {

// a cycle has been found

return true;

}

}

// no cycle has been found

return false;

}

}

// exporting circuit class

module.exports = {

Circuit: Circuit,

};

As you can see there are comments planning out methods that are yet to be coded. Unfortunately, this is where I ran out of time and was unable to do any more of the project. Refer to the evaluation for why I ran out of time and what I would have done if I more time to continue the project.

# Testing

Throughout development of my technical solution, I used an industry practice for software development called TDD (Testing Driven Development) by using a testing framework for JavaScript called Jasmine. Jasmine allows you to write specs which are pieces of code that tests other pieces of code by running this piece of code and comparing the actual output to an expected output and then provides useful error messages if the test fails. This allowed me to test each part of code as I developed it in order to find out that all parts of the solution worked as they were developed. Otherwise, I wouldn’t know if half of my code worked until integrating all the systems together integration itself can cause plenty of bugs itself let alone bugs in the algorithms themselves. The alternative would be the laborious task of manually tracing algorithms during development which is also prone to more human error. I believe that although development was slower and more meticulous this overall has saved a lot of time.

## Tests

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test | Test Spec | Test Type | Test Description | Test Outcome | Expected Outcome |
| 1 | Wire | Normal | Wire object has correct default values when instantiated. | Success | Success |
| 2 | Battery | Normal | Battery object has correct default values (different to wires defaults) | Success | Success |
| 3 | Lever | Normal | Lever object has correct default values when instantiated. | Success | Success |
| 4 | Lever | Normal | Lever has a method onClick which alternates the levers isOn Boolean value. | Success | Success |
| 5 | LinkedList | Normal | LinkedList has a head node and default length of 0 when instantiated. | Success | Success |
| 6 | LinkedList | Normal | LinkedList method insertFirst pushes a new node onto the head of the data structure and increments the length of the linked list. | Success | Success |
| 7 | LinkedList | Normal | LinkedList method insertLast adds a new node to the end of the data structure and increments the length of the linked list. | Success | Success |
| 8 | LinkedList | Erroneous | LinkedList method insertAt attempts to insert a node at an index outside of the length of the linked list but fails. | Success | Success |
| 9 | LinkedList | Boundary | LinkedList method insertAt attempts to insert a node at the start of the linked list and increments the length. | Success | Success |
| 10 | LinkedList | Normal | LinkedList method insertAt attempts to insert a node at the provided index in the linked list and increments the length. | Success | Success |
| 11 | LinkedList | Normal | LinkedList method getAt returns the data associated to the node in the data structure at the index provided. | Success | Success |
| 12 | LinkedList | Erroneous | LinkedList method getAt attempts to retrieve data at an index outside the length of the linked list and fails. | Success | Success |
| 13 | LinkedList | Erroneous | LinkedList method removeAt attempts to remove a node outside the length of the linked list and fails. | Success | Success |
| 14 | LinkedList | Boundary | LinkedList method removeAt removes the node at the head from the linked list and moves the head to the next node and decrements the length of the linked list. | Success | Success |
| 15 | LinkedList | Normal | LinkedList method removeAt removes the node at the index provided and decrements the length of the linked list. | Success | Success |
| 16 | LinkedList | Normal | LinkedList method clearList sets the head of the linked list to null and sets length to 0. | Success | Success |
| 17 | LinkedList | Normal | LinkedList method search returns the data associated to the node at the index provided. | Success | Success |
| 18 | LinkedList | Normal | LinkedList method remove removes the node with the data associated to the input of the method. | Success | Success |
| 19 | Graph | Normal | Graph has an empty linked list of vertices and a length of 0 when instantiated. | Success | Success |
| 20 | Graph | Normal | Graph method addVertex adds a vertex to the graph as well as instantiating an adjacency list as a linked list and increments the length of the graph. | Success | Success |
| 21 | Graph | Normal | Graph method addEdge checks if the vertices inputted exist then adds the index of one vertex to the adjacency list of the second vertex. | Success | Success |
| 22 | Graph | Normal | Graph method removeEdge checks if the target edge exists then removes the edge from the adjacency list. | Success | Success |
| 23 | Graph | Normal | Graph method removeVertex loops through all vertex’s adjacency list removing the edges of the vertex that is to be removed then removes the vertex from the graph and decrements the length. | Success | Success |
| 24 | Circuit | Normal | Class method hasLoop performs a depth first search algorithm on a predefined circuit graph with a loop. | Success | Success |
| 25 | Circuit | Normal | Class method hasLoop performs a depth first search algorithm on a predefined circuit graph without a loop. | Fail (tested positive for loop when there wasn’t one) | Success |

## Jasmine specs

Below are the spec tests I’ve written for each class as well as the Jasmine testing output when running the specs.

Graphical user interface, text

Description automatically generated

### WireSpec.js

// import necessary files

const { Wire } = require("../src/Wire");

const {

DEFAULT\_CURRENT,

DEFAULT\_RESISTANCE,

DEFAULT\_POTENTIALDIFFERENCE,

} = require("../src/properties");

describe("Wire", () => {

// initialises connection objects before each test

let wire;

beforeEach(() => {

wire = new Wire();

});

// tests default values have been set correctly

it("has a default current of zero", () => {

expect(wire.current).toEqual(DEFAULT\_CURRENT);

});

it("has a default resistance of zero", () => {

expect(wire.resistance).toEqual(DEFAULT\_RESISTANCE);

});

it("has a default potential difference of zero", () => {

expect(wire.potentialDifference).toEqual(DEFAULT\_POTENTIALDIFFERENCE);

});

});

### BatterySpec.js

// import necessary files

const { Battery } = require("../src/Battery");

const {

DEFAULT\_CURRENT,

DEFAULT\_RESISTANCE,

DEFAULT\_BATTERY\_POTENTIALDIFFERENCE,

} = require("../src/properties");

describe("Battery", () => {

// initialises battery objects before each test

let battery;

beforeEach(() => {

battery = new Battery();

});

// tests default values have been set correctly

it("has a default current of zero", () => {

expect(battery.current).toEqual(DEFAULT\_CURRENT);

});

it("has a default resistance of zero", () => {

expect(battery.resistance).toEqual(DEFAULT\_RESISTANCE);

});

it("has a default potential difference of zero", () => {

expect(battery.potentialDifference).toEqual(

DEFAULT\_BATTERY\_POTENTIALDIFFERENCE

);

});

});

### LeverSpec.js

// import necessary files

const { Lever } = require("../src/Lever");

const {

DEFAULT\_CURRENT,

DEFAULT\_RESISTANCE,

DEFAULT\_POTENTIALDIFFERENCE,

} = require("../src/properties");

describe("Lever", () => {

// initialises lever objects before each test

let lever;

beforeEach(() => {

lever = new Lever();

});

// tests default values have been set correctly

it("has a default current of zero", () => {

expect(lever.current).toEqual(DEFAULT\_CURRENT);

});

it("has a default resistance of zero", () => {

expect(lever.resistance).toEqual(DEFAULT\_RESISTANCE);

});

it("has a default potential difference of zero", () => {

expect(lever.potentialDifference).toEqual(DEFAULT\_POTENTIALDIFFERENCE);

});

// tests that lever functionality works

it("has a boolean isOn with default of false", () => {

expect(lever.isOn).toEqual(false);

});

it("has a method onClick which swtiches the lever isOn bool", () => {

let initialValue = lever.isOn;

lever.onClick();

expect(lever.isOn).toEqual(!initialValue);

});

});

### NodeSpec.js

// import necessary files

const { Node } = require("../src/Node");

describe("Node", () => {

// initialises connection objects before each test

let node;

beforeEach(() => {

node = new Node();

});

it("has data", () => {

node.data = 0;

expect(node.data).toEqual(0);

});

it("has next node pointer", () => {

expect(node.next).toEqual(null);

});

});

### LinkedListSpec.js

// import necessary files

const { LinkedList } = require("../src/LinkedList");

describe("LinkedList", () => {

// initialises connection objects before each test

let linkedList;

beforeEach(() => {

linkedList = new LinkedList();

});

it("has head", () => {

expect(linkedList.head).toEqual(null);

});

it("has length 0", () => {

expect(linkedList.length).toEqual(0);

});

it("has method insertFirst that pushes new nodes onto head", () => {

linkedList.insertFirst(1);

expect(linkedList.head.data).toEqual(1);

});

it("has method insertLast that adds new nodes to the end of the list", () => {

linkedList.insertLast(1);

linkedList.insertLast(2);

expect(linkedList.head.data).toEqual(1);

expect(linkedList.getAt(linkedList.length - 1)).toEqual(2);

});

it("has method inserAt that inserts data at an index within list", () => {

// tests that nothing happens if index is out of range

linkedList.insertAt(1, 10);

expect(linkedList.head).toEqual(null);

// initialising list

linkedList.insertLast(1);

linkedList.insertLast(2);

linkedList.insertLast(4);

// tests for inserting at first index

linkedList.insertAt(0, 0);

expect(linkedList.head.data).toEqual(0);

// tests for inserting at specific index

linkedList.insertAt(3, 3);

expect(linkedList.getAt(3)).toEqual(3);

});

it("has method getAt that returns the data of the node at index", () => {

// initialising list

linkedList.insertLast(0);

linkedList.insertLast(1);

linkedList.insertLast(2);

linkedList.insertLast(3);

// testing for when index out of range

expect(linkedList.getAt(10)).toEqual(undefined);

expect(linkedList.getAt(2)).toEqual(2);

});

it("has method removeAt that removes nodes at index", () => {

// initialising list

for (let i = 0; i < 10; i++) {

linkedList.insertLast(i);

}

// tests when index out of range

linkedList.removeAt(20);

expect(linkedList.length).toEqual(10);

// tests when removing head

expect(linkedList.head.data).toEqual(0);

linkedList.removeAt(0);

expect(linkedList.head.data).toEqual(1);

// tests when removing at index

expect(linkedList.getAt(5)).toEqual(6);

linkedList.removeAt(5);

expect(linkedList.getAt(5)).toEqual(7);

});

it("has method clearList that clears the list", () => {

// initialising list

for (let i = 0; i < 10; i++) {

linkedList.insertLast(i);

}

for (let i = 0; i < 10; i++) {

expect(linkedList.getAt(i)).toEqual(i);

}

linkedList.clearList();

expect(linkedList.head).toEqual(null);

expect(linkedList.length).toEqual(0);

});

it("has length property that works properly", () => {

expect(linkedList.length).toEqual(0);

linkedList.insertFirst(0);

expect(linkedList.length).toEqual(1);

linkedList.insertLast(1);

expect(linkedList.length).toEqual(2);

linkedList.insertAt(3, 0);

expect(linkedList.length).toEqual(3);

linkedList.removeAt(0);

expect(linkedList.length).toEqual(2);

});

it("has method search that returns the target if found", () => {

linkedList.insertLast(2);

linkedList.insertLast(3);

linkedList.insertLast(4);

linkedList.insertLast(5);

expect(linkedList.search(4)).toEqual(4);

expect(linkedList.search(6)).toEqual(undefined);

});

it("has method remove that removes a target node", () => {

linkedList.insertLast(2);

linkedList.insertLast(3);

linkedList.insertLast(4);

linkedList.insertLast(5);

expect(linkedList.search(4)).toEqual(4);

linkedList.remove(4);

expect(linkedList.search(4)).toEqual(undefined);

});

});

### ComponentSpec.js

// import necessary files

const { Component } = require("../src/Component");

const { LinkedList } = require("../src/LinkedList");

const { Wire } = require("../src/Wire");

describe("Component", () => {

// initialises connection objects before each test

let component;

beforeEach(() => {

component = new Component(0, new Wire());

});

it("has component", () => {

expect(component.index).toEqual(0);

});

it("has component", () => {

expect(component.component).toBeInstanceOf(Wire);

});

it("has a linked list", () => {

expect(component.listOfConnections).toBeInstanceOf(LinkedList);

});

});

### GraphSpec.js

// import necessary files

const { Graph } = require("../src/Graph");

const { LinkedList } = require("../src/LinkedList");

const { Component } = require("../src/Component");

const { Wire } = require("../src/Wire");

const { Battery } = require("../src/Battery");

const { Lever } = require("../src/Lever");

describe("Graph", () => {

// initialises connection objects before each test

let graph;

beforeEach(() => {

graph = new Graph();

});

it("has list of vertices", () => {

expect(graph.vertices).toBeInstanceOf(LinkedList);

});

it("has length 0", () => {

expect(graph.length).toEqual(0);

});

it("has method addVertex that adds a vertex to vertices", () => {

graph.addVertex(new Wire());

graph.addVertex(new Battery());

graph.addVertex(new Lever());

expect(graph.vertices.getAt(0).index).toEqual(0);

expect(graph.vertices.getAt(0).component).toBeInstanceOf(Wire);

expect(graph.vertices.getAt(0).listOfConnections).toBeInstanceOf(

LinkedList

);

expect(graph.vertices.getAt(1).index).toEqual(1);

expect(graph.vertices.getAt(1).component).toBeInstanceOf(Battery);

expect(graph.vertices.getAt(1).listOfConnections).toBeInstanceOf(

LinkedList

);

expect(graph.vertices.getAt(2).index).toEqual(2);

expect(graph.vertices.getAt(2).component).toBeInstanceOf(Lever);

expect(graph.vertices.getAt(2).listOfConnections).toBeInstanceOf(

LinkedList

);

});

it("has method addEdge that adds edge between two vertices", () => {

graph.addVertex(new Wire());

graph.addVertex(new Battery());

graph.addVertex(new Lever());

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(1, 2);

graph.addEdge(1, 3);

expect(graph.vertices.getAt(0).listOfConnections.getAt(0)).toEqual(1);

expect(graph.vertices.getAt(0).listOfConnections.getAt(1)).toEqual(2);

expect(graph.vertices.getAt(1).listOfConnections.getAt(0)).toEqual(2);

expect(graph.vertices.getAt(1).listOfConnections.getAt(1)).toEqual(undefined);

});

it("has method removeEdge that removes an edge between two vertices", () => {

graph.addVertex(new Wire());

graph.addVertex(new Battery());

graph.addVertex(new Lever());

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(1, 2);

graph.addEdge(1, 3);

expect(graph.vertices.getAt(0).listOfConnections.getAt(1)).toEqual(2);

graph.removeEdge(0, 2);

expect(graph.vertices.getAt(0).listOfConnections.getAt(1)).toEqual(undefined);

});

it("has method removeVertex removes vertex from graph", () => {

graph.addVertex(new Wire());

graph.addVertex(new Battery());

graph.addVertex(new Lever());

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(2, 0);

graph.addEdge(1, 2);

graph.addEdge(1, 3);

expect(graph.vertices.getAt(0).component).toBeInstanceOf(Wire);

expect(graph.vertices.getAt(0).listOfConnections.getAt(0)).toEqual(1);

expect(graph.vertices.getAt(0).listOfConnections.getAt(1)).toEqual(2);

expect(graph.vertices.getAt(2).listOfConnections.getAt(0)).toEqual(0);

graph.removeVertex(0);

expect(graph.vertices.getAt(0).component).toBeInstanceOf(Battery);

expect(graph.vertices.getAt(1).listOfConnections.getAt(0)).toEqual(undefined);

});

});

### CircuitSpec.js

const { Battery } = require("../src/Battery");

const { Circuit } = require("../src/Circuit");

const { Wire } = require("../src/Wire");

const { Lever } = require("../src/Lever");

describe("Circuit", () => {

let circuit;

beforeEach(() => {

circuit = new Circuit();

});

it("has method hasLoop which returns a boolean value on whether there is a cycle or not in the graph", () => {

// instantiates components in graph

circuit.graph.addVertex(new Wire());

circuit.graph.addVertex(new Battery());

circuit.graph.addVertex(new Lever());

// adds edges to graph

circuit.graph.addEdge(0, 1);

circuit.graph.addEdge(1, 2);

// loop is present

expect(circuit.hasLoop()).toEqual(false);

circuit.graph.removeEdge(0, 1);

// loop is broken

expect(circuit.hasLoop()).toEqual(false);

});

});

# Evaluation

## Feedback

Elliot: After being explained in layman’s terms how the simulator works, the project looks extremely complicated. It would have been nice to have a working tool but I understand why it fell short.

\_\_\_\_\_\_\_\_\_\_\_\_\_ 30/03/2023



## What went wrong

I do not think that I have managed to produce an MVP over the course of the project. I believe the over ambitiousness along with the lack of knowledge and experience hindered me from completing the project to the best of my ability. One month was dedicated just to researching the best solution to the problem, however due to the specific nature of the problem and lack of experience and not knowing what the correct buzzwords to use to effectively search for the information I needed in order to complete the project was extremely difficult. Another month was lost to the original design as seen in attempt one of the technical solution I had finally settled on attempting, relying on solely front end which not only would have been inefficient and messy but also lack any possibility of a database at all. It was extremely difficult to know how much time to dedicate to attempting one solution before calling it quits and trying another. If I were to have stuck with one solution, I always ran the risk of hitting a dead end and not finishing the project for examples through limitations of the frameworks I’ve chosen. On the other hand, if I attempted a new solution, I then have to start a new with less time. Finally, I had settled on a more ambitious more efficient solution that uses many practices from industry level software developers such as the Jasmine testing environment I used throughout the development of the solution. Unfortunately, I came across a difficult logical problem (the connecting components problem) to solve that to my disappointment took two months to finally come up with a viable solution at which point I was left with very little time to complete the rest of the project. In hindsight it would have been smart to divert my efforts into developing the frontend or database while I had hit that snag in the backend. However, in the moment it felt as though without that problem overcome, I could not continue on to the rest of the project because it felt like the rest of the project relied on that one bit of infrastructure. I believe that if I had not had these setbacks than the project would have been a lot more complete and impressive due to the extra time I would have had to continue development. I would argue that the project or at least the solution I chose to implement for the project was way above A level standard due to the complexity of the project and the industry standard solutions I chose to implement such as Jasmine testing and the MERN stack I was implementing. The MERN (Mongoose, Express, React, Node) stack is one way to implement a full stack project however I did not get onto the react part of the project which is more of the frontend.

## What I would do next

If I had more time, I would have developed the MongoDB database and the relevant schemas using the Mongoose package to map the component class objects to JSON files which is how you send data to a MongoDB database.

Then I would start development on the frontend using react and bootstrap (a CSS framework) that allows for the ability to drag components from a menu to the grid. I would then develop the relevant methods in the circuit class that I ran out of time for. Including visual display and animations of the relevant information on screen such as the electrical properties of each component and even an animation to show flow of charge.

By then I would have a great MVP but that would take probably another couple months of development at least to get to this point. Time I would have had if I didn’t have certain setbacks. If I were to extend the project I would start to implement features from the extension section in the smart objectives. Specifically, capacitor components and the relevant physics that comes included with that as well as a signup page that allows a user once logged in to save and open premade circuits. I could also have developed other more educational features that better teach the user the physics behind the circuit that they have developed such as perhaps detecting features of a circuit and providing relevant YouTube video links explaining the concept. For example, if it detects a parallel circuit it could prompt a link to a YouTube video explaining the differences between parallel and series circuits.