OCR GCE A

COMPUTER SCIENCE

PROJECT

H446-03

Name:	Angus Bowling
Candidate Number:	6023
The King's (The Cathedral) School:	22333
Title of Project:	Logic Circuit Simulation

PROJECT CONTENTS

TABLE OF CONTENTS

ANALYSIS

PROJECT DEFINITION

In computing all logic is built up from the seemingly atomic logic gates, this allows for all calculation, logical comparisons, and decision making to take place. Logic gates are electronic building blocks of computing which each carry out one specific logical comparison of one or many inputs and provide the comparisons output.

However, by the nature of logic gates being small and requiring either soldering to a PCB or working on a breadboard, many students have a hard time comprehending logic gates and how their simple operation can lead to complex computing. This lack of hands-on work with logic gates ensures a lack of knowledge about low level computational systems.

To allow for students to gain the knowledge and understanding of logical systems, I intend to create an interactive software simulation of logic gates and their incorporation into logical systems. Including gates such as AND, OR, XOR, and their negations. As well as controllable inputs and outputs; such as buttons, switches, clocks, and lights.

A visual and interactive display of logic gates allows for students to gain a much greater understanding of logical systems and the hands-on nature of the software leads to a deeper than surface level grasp on how logic gates interact with each other and how they can build up more complex systems.

The simulation would also allow for users to see the circuit they create represented in Boolean algebra. This would allow for the breaking down of the disconnect many students have between the abstract Boolean expression they learn in class and the real-life creation and use of these comparisons.

With circuits being represented in a Boolean expression, the software would also allow users to save and load circuits they have created previously. As well as being able to simplify circuits by removing redundant gates, showing students how simplified expressions can save space and money for manufacturers.

The upcoming work will be in researching possible digital representations of logic gates and how to calculate and show logical operations. As well as discussing with computer science educators on how best to layout and show the progression of logic through the circuit in a way that is both appealing and easy to understand for the user. Along with the scope and scale of the simulation as to allow for its effective development.

STAKE HOLDERS

The software is designed to be used by students and educators, but it also caters to the hobbyist demographic. Due to the large and varied scale of the target audience a small but representative sample will make up the stakeholders; this will include students ranging from a deep understanding of logical circuits to ones with no knowledge on the area whatsoever. There will also be representatives from educators so the software can align to their requirements.

The stakeholders that represent the student users of the software will be Richard, Izzy, and Derek; with none, limited, and in-depth understanding of logical circuits as to gain an inclusive overview of necessities for the broad

range of users. The stakeholder that represents the educator's demographic is Mrs. Stimson; she is a computing teacher and has a deep understanding of how students will use and react to certain aspects and features of the software, as well as the knowledge of what areas most students find difficult and need aid on regarding logical circuits. This will allow for me to develop the software with the end user's need and requirements in mind.

SUITABILITY OF A COMPUTATIONAL SOLUTION

The problem is well suited to a computational solution as the simulation of even the most complex and convoluted logical comparisons and circuits is trivial for a computer. The disparity between the human's and computer's capability to rationalise large logical expressions is what makes the simulation useful and usable to students as appose to conventional teaching methods. The visuals and customisability are also only possible with computational methods so there would not be another way to approach the problem. The well-defined nature of the problem lends itself well to a computational approach as the software's goals, correctly calculating logic, are clearly predefined.

COMPUTATIONAL METHODS TO APPROACH A SOLUTION

RECOGNITION OF THE PROBLEM

The base problem is finding and implementing a way to represent and implement logical gates and circuits, as well as allow for them to interact with each other, in a simulated setting. The key part of this will be finding a way to represent the circuit that will be fast to compute outcomes off, while not being overcomplicated and computationally demanding. The implementation of an elegant representation of the circuit should make it much simpler to develop the rest of the simulation such as its intractability and the Boolean representation.

DECOMPOSITION OF THE PROBLEM

The development of the simulation will be made far more efficient by breaking down the overarching problem into several small and solvable sub-problems. These could include:

- Allowing the user to build and edit the circuit.
- Allowing for the user to control the inputs to the circuit.
- Calculating and displaying the flow and outcome of the logic through the circuit.
- Calculating and showing the user a representation of the circuit in Boolean algebra.
- Allowing for the user to personalise settings (such as clock speeds, ascetics, or algebra type).

With the independence of these problems, they should be approachable in isolation meaning they can be developed and amended specifically when required. This will lead to a faster and more efficient development.

ABSTRACTION IN THE SOLUTION

The program is intended to be used by students with even no prior knowledge of logic circuits. For this reason, it is vital that the simulation is approachable and easy to understand. This means that many possible complexities will

be abstracted away from the user, leaving them with only clear inputs and outputs, and atomic logic gates that cannot be broken down further. This should allow for the users to focus purely on learning how the gates interact with each other and not how they are made or how they look in reality. For instance, there will be no rise time or maximum fan-out as these concepts do not deepen the user's knowledge of logical operations but only real-world electronic limitations.

Abstraction will also be used to make computation easier and faster, so no high-end hardware is required to run the simulation. This could include only factoring in and calculating the output of gates that will affect the final output even though, in reality, their output would be present.

DIVIDE AND CONQUER METHODOLOGY

Each sub-problem that has been outlined, while difficult, is very much solvable, and much easier to approach than the initial overarching problem. The completion of one sub-problem will also ease the development of the others, as the load will be reduced and new approaches to solutions will become even more evident. The object-based simulation is well suited to be a modular program and modular development using a divide and conquer methodology will speed up development while also making bugs easier to find and fix.

VISUALISATION

The software is designed to allow for the user to be able to understand logic circuits and Boolean algebra. By showing the user these topics via an interactive simulation, it enables them to understand those areas in a different way, utilising visualisation to better their learning as well as the effectiveness of my software. Visualisation will also be used in the software through the computational representation of the logic circuit. The circuit can be edited and have its outputs calculated for quicker and easier by the computer if the software internally represents the circuit as a graph. This will also reduce the hardware requirements of the user as the calculations will be more efficient and the software will use less memory. This is ideal as it makes the software usable by anyone even people just getting in to computing.

CONCURRENCY

The software must be as easy to use as possible so that beginners can use it effectively; due to this, the simulation should not have to be stopped and restated to allow for editing of the circuit. By using concurrent methods in the development of the software, it should allow for the simulation to be more intuitive to use and allow for students to gain a deeper understanding of logical circuits as they can spend all their attention on their work, and not on how to work the software.

INTERVIEWS

I intend to keep my stakeholders involved at all stages of development as to ensure that my software is suitable to their requirements and meets all their needs. I will ask each of the stakeholders a set of questions, different for the student representatives and the educator representative, to gauge what features they would like to see in my software as well as if and how they use current software. The questions will just be a starting point for the interviews, and I will go further in-depth where possible with the stakeholders on areas more specific to them as well as allowing them to ask any questions they have at the end.

QUESTIONS

The questions that I will ask the students (Richard, Izzy, and Derek) are:

1. What would you say your current understanding of logic gates/circuits is?
2. Have you ever used software that simulates logic circuits?
    a. If yes, then what reason did you use it for?
    b. Where you happy with your experience with it?
3. Do you think an interactive software would aid with your understanding of logic circuits?
4. Would you be more likely to use an interactive software than traditional methods of learning?
5. Would you say you have a strong understanding of how Boolean algebra relates to logic circuits?
6. What feature would you say is most important for you that the software has?

The questions that I will ask the educator (Mrs. Stimson) are:

1. Would you say that many students struggle with understanding logic circuits?
2. Would you be open to using an interactive simulation with your students?
3. Do you think there is a disconnect between students' knowledge of Boolean algebra and logic circuits?
4. Do you think an interactive simulation would aid students as appose to using only traditional methods?
5. What features do you think students would require most / make the most use of?
6. Are there any features that would make the software preferable to teachers / tutors?

RESPONSES

After interviewing each of my stakeholders, their responses to each question were as follows:

RICHARD (STUDENT)

What would you say your current understanding of logic gates/circuits is?

» Definitely below the level I wish it was. I don't really get them in the way some people seem to and I fear that it's going to negatively impact my grade if a question comes up in an exam.

Have you ever used software that simulates logic circuits?

» No, I haven't. I never really knew that was a thing. I always just re-read my textbook to try and understand how they work.

Do you think an interactive software would aid with your understanding of logic circuits?

» Yes, I feel that I would be much more engaged by something that I could control and experiment with. A different take on logic circuits may let me understand them in a different way.

Would you be more likely to use an interactive software than traditional methods of learning?

» Yes definitely. I think it would be much more fun and interesting than just reading my textbook. I would be much more likely to continue revising if it were using software that made me interact with it instead of just passively taking it in.

Would you say you have a strong understanding of how Boolean algebra relates to logic circuits?

» No not really. I know that they are meant to show the same thing, but they just seem so incompatible. I don't get how logic gates can be mixed up and put together in like a 2d circuit and then be represented as a simple expression.

What feature would you say is most important for you that the software has?

» Not a feature per se, but I would need any software I use to be really simple to pick up. I think if I had to try and learn how to use a complex and cluttered interface I wouldn't get as much out of the software compared to if I could just start using it straight away.

IZZY (STUDENT)

What would you say your current understanding of logic gates/circuits is?

» I understand them a lot more than I did before I started my A Level course. They are one of my weaker areas because I only really learnt how to answer question about them and not understand them on a deeper level.

Have you ever used software that simulates logic circuits?

» Yes, I think I have. Sir made us use them in a lesson once when we were first learning about adders.

If yes, then what reason did you use it for?

» We were mainly just seeing if any of us could work out how to make a full adder. Then after he told us how to we experimented with mixing in the other gates.

Where you happy with your experience with it?

» It was a bit overwhelming to use when we first opened it. The control of the gates themselves was intuitive though and I liked that there were many ways to gives the circuit inputs and view the outputs.

Do you think an interactive software would aid with your understanding of logic circuits?

» Definitely. I think it would really help me to comprehend how you can join logic gates together and create computation like addition and memory that we only wrote notes about.

Would you be more likely to use an interactive software than traditional methods of learning?

» Yes, I would. It was so much more fun and memorable when we actually built the circuits than when we just read about them or copy them off the board. I really like how you can just tinker with a circuit and see real changes to the output caused by your actions.

Would you say you have a strong understanding of how Boolean algebra relates to logic circuits?

» Sort of. I know that you can express the gates as those symbols and the inputs as letters, but I would have no idea how to actually convert a circuit to algebra or vise versa. I really don't get how people can read the algebra and just know how it would react to certain inputs.

What feature would you say is most important for you that the software has?

» When we used the software in class the part, I found most memorable was messing about with the different inputs and outputs; it just really made me feel like I was in control of the circuit itself. I would definitely like there to be lots of input and output types to choose from.

DEREK (STUDENT)

What would you say your current understanding of logic gates/circuits is?

» I would say that I am confident with them. I did some further reading on them when I finished my GCSE courses because I found them really fascinating, so when it came to my A Level, I felt like I already knew a lot of what we learnt.

Have you ever used software that simulates logic circuits?

» Yes. When I was reading about logic gates, I search online for a simulation so I could test my understanding of them and how they interact with each other.

If yes, then what reason did you use it for?

» Mainly just to make sure that my theoretical understanding of how they worked was correct in actuality. I also tested out building some things like latches and flip-flops to see how complex actions could be made by simple logic gates.

Where you happy with your experience with it?

» Most of the software I found online worked as intended, but I feel like it was very bare bones and lacked lots of customisability that could have made it even more useful. However, I only used free software so maybe those calibers of features are locked behind a pay wall.

Do you think an interactive software would aid with your understanding of logic circuits?

» Yes, I would say so. I understood all the theory behind logic circuits when I read it in the textbook, but I didn't really understand why many things worked like they do. I found that I only really understood the Boolean simplification when I made two circuits side by side and saw how they were different, but their outputs were always the same, seeing how parts of the circuit remained unaffected by certain inputs really cleared things up for me.

Would you be more likely to use an interactive software than traditional methods of learning?

» For sure. I think the textbook I use really doesn't explain logic gates in a useful way. It focused on just memorizing truth tables for certain gates and circuits which was hard until I used an interactive software. It really allowed me to understand how the outputs are generated from the inputs when I could control the circuit myself. Know I don't even need to remember the truth tables because I can work them out on the fly.

Would you say you have a strong understanding of how Boolean algebra relates to logic circuits?

» Yes, I do. I find it really useful to look at logic circuits as a Boolean expression because it allows me to see the circuit like a conditional statement in a programming language. It really helps me see what a specific output would be a lot faster than running logic through a circuit gate by gate.

What feature would you say is most important for you that the software has?

» For me customisability is really important. I think that it's such a shame that the interactive nature of a lot of the simulations out there ends at just creating the circuit from the gates. I would also find it very useful to be able to save and load circuits that I have built before, so I don't have to worry about losing a circuit I spent a long time on.

MRS. STIMSON (EDUCATOR)

Would you say that many students struggle with understanding logic circuits?

» Yes, I would.

Would you be open to using an interactive simulation with your students?

» It would be beneficial.

Do you think there is a disconnect between students' knowledge of Boolean algebra and logic circuits?

» Yes - I think students can recognise the different logic gates but then when they have to apply that knowledge to a given scenario they struggle.

Do you think an interactive simulation would aid students as appose to using only traditional methods?

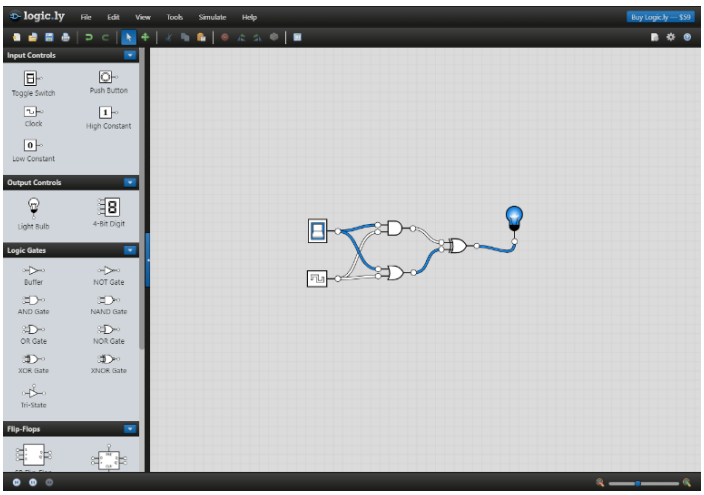
» Instructions on how to use, ability to refresh and start again, once they have the logic circuit set up maybe having it written out in plain English then they can check what they have done in relation to what they wanted to do.

What features do you think students would require most / make the most use of?

» The ability to save and share logic circuits.

RESEARCH INTO EXISTING SOLUTIONS

LOGIC.LY

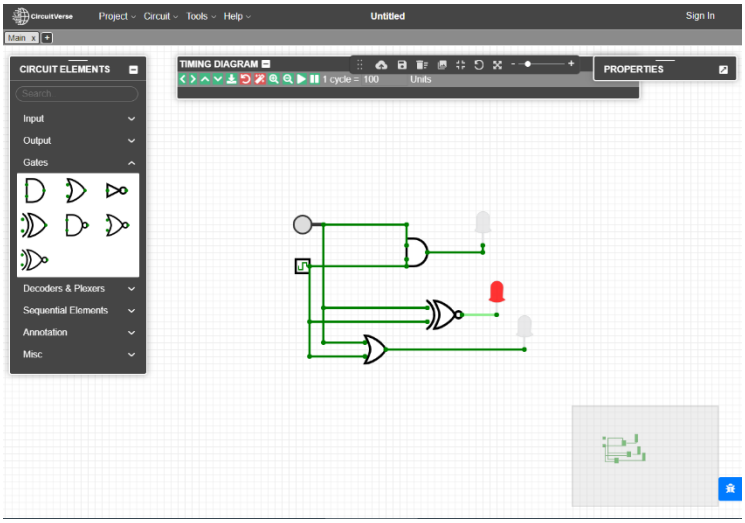


Logic.ly is a web-based logic circuit simulator. It uses drag and drop functionality to move and place logic gate, inputs, and outputs. It uses a visual colour change to represent the output of the logic gates making it easy to trace the flow of logic throughout the circuit. The simulation comes with some prebuilt circuits such as flip-

flops to allow for their easy implementation. The user can also control settings such as colour schemes and propagation timings. Along with this the users can scroll in and out, as well as panning across the grid. The simulation allows for saving and loading of circuits that have been built before and there is an option to generate a truth table for the circuit, which can be exported as a CSV. The simulation did have a lot of customisability but lacked the option to have clocks with different speeds. There was also no option to generate a representation of the circuit in Boolean algebra.

Logic.ly has many well integrated features and, while the screen may be close to being overcluttered, the gates have an appealingly simplistic design that make it easy for beginners to use. Logic.ly is aimed more at allowing users who already understand logic gates to design circuits whereas my software will focus more on teaching and educating users about logic gates and circuits. Logic.ly is well made and runs smoothly with no obvious errors and I intend to make my software fast and efficient as it should run and update in real time, allowing for users to edit circuits while the simulation is running.

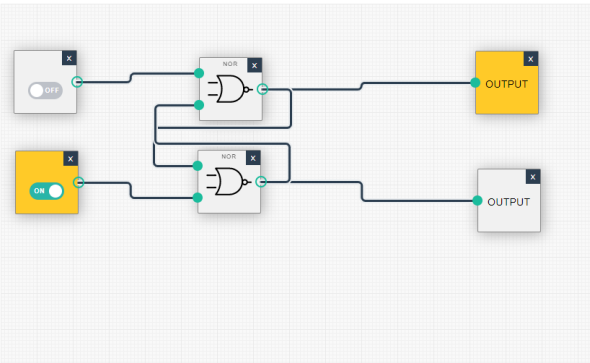
CIRCUIT VERSE

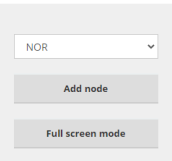


Circuit Verse is an online logic simulator which allows users to build, edit, and runs logic circuits in the browser. The simulation allows for users to build circuits up from base logic gates as well as prebuilt components such as flipflops, adders, or decoders. Circuit Verse also gives the option to run the software in “Lite Mode” which allows it to run smoothly on weaker hardware. You also have the option to edit the clock speed of the simulation, however you cannot edit the speed of individual clocks. You are given the option to load and save projects both online and offline. The UI can be toggled to one of a few themes and can be dragged and dropped to wherever is desired. The software also allows the user to generate truth table; either by hand or from an algebraic expression. This truth table can then be used to generate a circuit; however, you cannot generate an algebraic expression from a circuit that you build.

Circuit Verse may be less intuitive than other software available, but it has lots of unique feature once you understand how to use it. Its UI is slightly clunky and confusing to navigate for beginners which is something I will aim to avoid, allowing my software to be approachable. I like the ability to generate circuits from logical expressions, however if I were to implement this, I would allow for the user to also generate a logical expression from a circuit such that they could more easily understand how Boolean algebra represents real circuits.

ACADEMO





The Academo logic gate simulator is logic circuit simulator that runs in the browser. The software allows users to create logic circuits by inputting logic

gates as well as inputs and outputs. It uses a drop-down menu to select which gate you want to input to the simulation; this allows the user to use the software even if they do not know the symbols for specific logic gates. The simulation uses simple on-off toggle switches for the inputs and uses colour to represent the outputs. All the components are given as blocks that can be dragged around as well as removed. The UI of the software is very clean and easy to understand and use, this makes it ideal for beginners approaching it for the first time. It also comes with a full screen mode to allow for the use of the software without distraction.

Academo has a lot less functionality than many other available simulations, however, its lack of over-cluttering makes it far more approachable for new users and users with limited knowledge. As an educational tool, it is vital that my software is not intimidating to students who lack understanding of logic gates. Due to this I will make sure that my simulation is also easy to start using no matter the user's prior knowledge or abilities. I like the use of a drop-down selection for the gates, as anyone can know what gates they are using. However, if I implement this, I will have different menus for inputs, logic gates, and outputs. I would also aim to provide more components, such as clocks or push buttons as the Academo software is lacking many of these.

STAKEHOLDER INPUT

I reconvened with several of my stakeholders as to present features and attributes from the existing software. This allowed me to be certain about the key features that my simulation required as well as ensuring I felt justified with my inclusions and exclusions; as my solution had to meet my stakeholders' desires. When shown my research and asked what features they say as the most vital, theses were their responses:

IZZY

I really like the look of the Academo software, the box style and simple complementary colours make it obvious what is going on. The large number of different elements available in the Logic.ly software also seems very useful, as you could really experiment with many different unique combinations without getting bored so I think it would be great to use in class.

DEREK

What stands out to me straight away is how the Circuit Verse software allows you to create circuits from Boolean expressions. That would really help people understand how the expressions are linked to the circuits, and along with the option to build a circuit from a truth table, you could create a circuit that acted in a specific way with no prior experience. I also liked the prefabs that Logic.ly had, such as the flipflops, because that would really let you experiment at a higher level without the screen getting over-cluttered.

FEATURES OF PROPOSED SOLUTION

At a general level, my software will be a web-based simulation of logic circuits. The simulation will be fully interactive and allow users to create logic circuits from scratch; alternatively, they can generate them from truth tables or Boolean expressions. The users will also be able to edit any circuits using drag-and-drop functionality as well as customise many features, such as clock speed or Boolean algebra type. All these features are designed in such a way as to make the software as user friendly as possible, as the simulation is designed to be useable by complete beginners in the field. This is also the reasoning behind the minimalistic and approachable UI, as the user

should not be intimidated by the software because that may impede their use of the simulation. The ability to generate a Boolean expression or truth table from a user-built circuit will also be available. This will aim to break down students disconnect between logic circuits and theoretical logical constructs, such as Boolean algebra. This will also allow educators to save and reuse specific circuits between multiple classes, ensuring that the software is applicable to use in school. To be certain that the simulation caters to the requirements of higher-level users with a greater breadth of knowledge, the ability to insert and work with more complex components, such as flipflops or latches, will be available. This will make sure that they can focus on the interaction between logical elements, and not just replicating them multiple times.

LIMITATIONS OF PROPOSED SOLUTION

My proposed solution is intended to allow students to learn about logic gates and how the interact with each other to form logic circuits. Due to the scope of my project, and the limited amount of time I have to work on it, it is necessary for my simulation to have some limitation. A key limitation that I will have, due to time and computational power, while be the real-world accuracy of my simulation. Although the logic of my gates will all be correct, some of the physical attributes of the logic gates will have to be ignored, as the program will not simulate the flow of electricity through the circuit, only the logic. Examples of electronic effects that will not be included are rise-time of the gates, as the gates will switch between states instantly. Another limitation is the lack of a maximum fan-out for the gates, as each gate will be able to be an input to any number of gates without losing power. I feel like these limitations will not detract from the overall focus of the software, teaching users about logic circuits, as in all but very low-level circuit design, these attributes are abstracted away. An example of when users will not need to worry about these physical effects is for students who want to learn about how logic is built up while programming.

SOFTWARE & HARDWARE REQUIREMENTS

To develop and run the software specific software and hardware requirements will need to be met. This will allow for the creation as well as the testing and deployment of my simulation and is vital to set out early as it will determine the scope of my software.

HARDWARE REQUIRMENTS

- **A modern computer**
  - » The simulation will require a typical computer with the power to complete calculations about the logic circuit on each simulated clock cycle. If the created circuit is smaller or the clock speed is decreased, then the simulation would also run on low-end/older computers.
  - » While developing the software it is important that it runs smoothly so I can focus on fixing errors caused by the program and not by a bottlenecked computer.
- **Typical IO peripherals**
  - » The software is designed to be interactive to give the user the deepest learning experience; This means that to use the software the students must have suitable peripherals. This includes a

monitor, keyboard, and mouse/trackpad. Allowing users the ability to control and interact with the software in their desired manner.

» During development I will need to write, edit, and test my software to ensure that it runs as intended. This will require me to use hardware such as a keyboard and mouse to interact with the IDE I am using and prototypes of my simulation.

SOFTWARE REQUIRMENTS

- **Modern OS (Windows, Mac, or Linux based)**
  - » To run the simulation, and any required software, such as web browsers, an operating system is required. Due to JavaScript's wide scale use it is supported on most modern OS's so my software should work on any of them.
  - » While writing my software I will need to use programs such as web browsers and an IDE. This software is supported by almost any modern OS, especially multitasking ones such as Windows, Mac, or Linux.
- **Web browser**
  - » The software will be web-based, so to visit the website and run the simulation, a browser is required. Almost all modern browsers run JavaScript but there are some very low spec requirements to use it; as well as some for libraries I intend to use, such as P5.JS.

As of September 2018, this means that we support:

Browser	Current Version	Previous Version	Notes
Internet Explorer	v. 11	Not supported	No support for WebAudio
Microsoft Edge	v. 42	v. 41	
Chrome	v. 68	v. 67	
Chrome for Android	v. 68	v. 67	
Firefox	v. 61	v. 60	
Safari	v. 11	Not supported	
iOS Safari	v. 11.4	v. 11.2	
Opera	v. 54	v. 53	

Requirements for the P5.JS library

STAKEHOLDER REQUIREMENTS

There are some requirements dictated by my stakeholders that my software must meet. This will ensure that my software does what they need it to, and my simulation does what it has been designed and requested to do.

In terms of hardware requirements, it is vital that my software runs smoothly and efficiently on any of the computers at school. All of my stakeholders' personal computers, laptops and desktops, are at least as powerful as these and the ability to use the software in school or during lessons is a must. These computers all have standard peripherals of keyboards, mice, and monitors; so, the software must be able to be interacted with using these. The software should also be usable with a trackpad so that it can be run on a laptop.

The software requirements from the stakeholders follows a similar suit. As the simulation must run on the school computers, it must run well on Windows operating systems, specifically Windows 10. The computers all have access to both Google Chrome and Microsoft Edge web browsers; therefore, it must run on at least one of these

browsers. However, upon discussing with my stakeholders, most of them use only Google Chrome when at home. Therefore, the software will definitely work on Google Chrome, and I will aim to ensure it works on Microsoft Edge as well. The simulation itself must also conform to certain design requirements set by the stakeholders, however, these have already been stated and justified previously in my features of proposed solution.

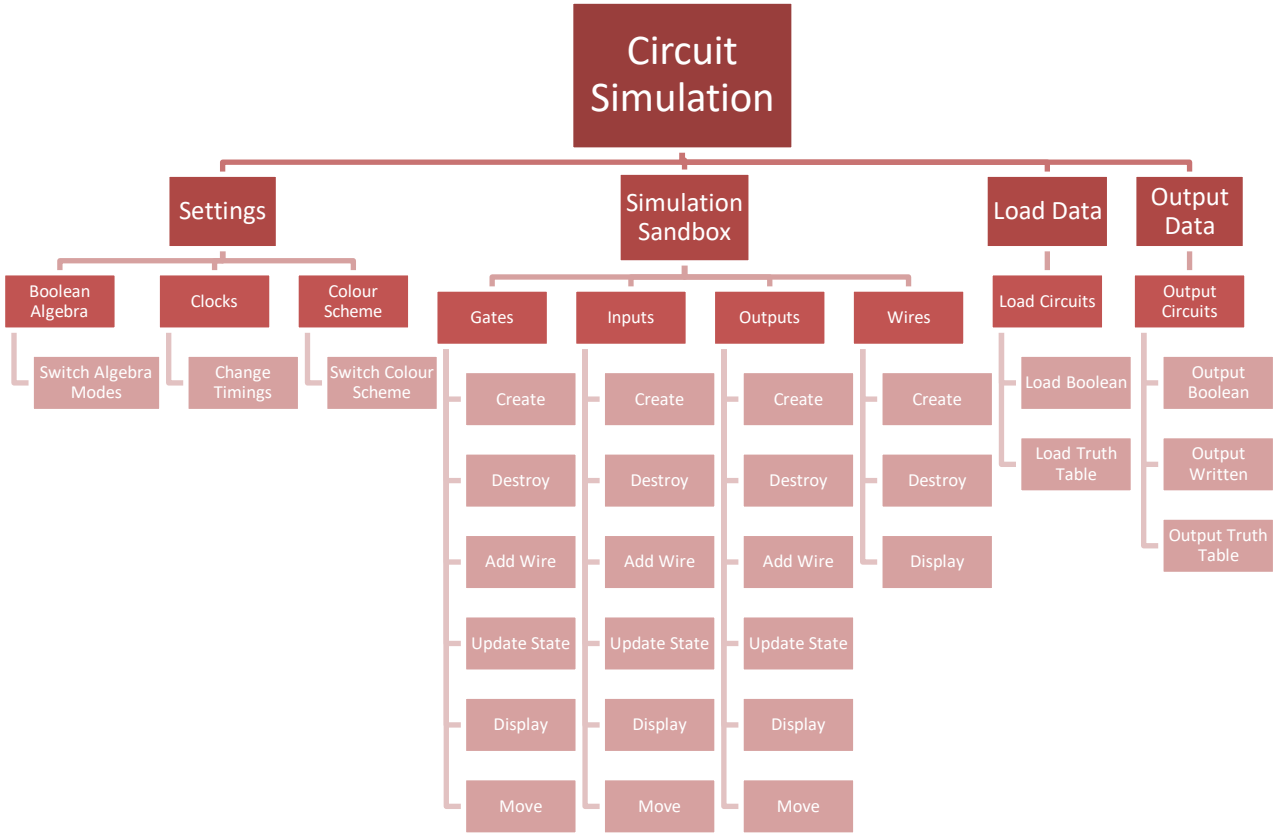
SUCCESS CRITERIA

Criterion	Justification	Evidencing
Allows the user to build their own logic circuits	The simulation is made to allow for users to build up logic circuits so that they can learn about how they work	Screenshot of the simulation with a user-built logic circuit
Correctly calculate logic for NOT, AND, OR, and XOR gates	The simulation must allow for the use of many types of logic gate so that truly complex logic can be built up, allowing the user the complete experience	Screenshot of each of the logic gates being used and behaving correctly to a given input
Allow the user to control the inputs to a circuit	This lets the user experiment with how different logic gates are affected and work with varying inputs	Screenshot showing the inputs to a circuit being controlled by the user
Allow the user to see the outputs of a circuit	Gives the users a clear understanding of how logic gates and their inputs interact with each other	Screenshot of the outputs of a circuit that are clear to see and understand
Allow the use of clocks as inputs to the circuit	The ability to use a wide range of components in the logic circuits will allow the users to broaden their knowledge as much as possible	Screenshot of clocks being used as an input to a circuit
Allow the user to edit the speed of clocks	By ensuring that the components are interactive, the user will feel like they have more control over the running of the logic circuit and understand its nuances more	Screenshot showing the process of a user editing the speed of a clock
Allow the user to generate a Boolean expression of their circuit	The ability to generate a Boolean expression from the users own logic circuit will break down the disconnect between formal Boolean algebra and logic gates	Screenshot of the program displaying the correct Boolean expression of a user-built circuit
Allow for the use of multiple Boolean algebra symbol sets, and the ability to switch between them	This allows for the user to use the symbols they are most comfortable with as well as experimenting with and learning new symbols for logic gates	Screenshot of Boolean algebra in multiple forms as well as a screenshot of the interface to switch between them
Allow the user to edit a circuit while its running	By letting the user have control over and edit the simulation while	Screenshot of a user editing a currently running circuit

	its running will make the project seem more interactive and lead to greater retention of knowledge	
Allow the user to generate a truth table of a circuit	This gives the user another way to see and understand the interactions of the logic gates in a circuit	Screenshot of the program displaying a correct truth table of a user-built circuit
Allow the user to generate a circuit from a Boolean expression	This will not only allow users to recreate a circuit that they have created before; but also see how Boolean algebra and logic circuits are related	Screenshot of a circuit correctly generated by a Boolean expression input by a user
Allow the user to generate a circuit from a truth table	This will let users create a circuit that behaves in a particular way even if they still do not fully understand Boolean logic	Screenshot of a circuit correctly generated by a truth table input by a user
Drag and drop control of the logic gates	A simplistic and intuitive way of interacting with the simulation will make it usable and approachable to beginners	Screenshot of the drag and drop functionality in use
Uncluttered and readable interfaces	A UI that is easy to understand and use will allow the users to focus all their attention onto the logic circuit they are building	Screenshot of the user interfaces

DESIGN

SYSTEMS DIAGRAM



EXPLENATION OF SYSTEMS DIAGRAM

The use of a systems diagram to outline the scope of my project is vital. This is as it allows me to break down the solution into simple, well-defined sections that can be approached sequentially and completed independent of each other. It also provides a system to measure my progress so I can manage my limited time and ensure all areas of my solution get completed. Taking a top-down approach allows me to use logic and procedural thinking to ease the development of my project.

BREAKDOWN OF KEY COMPONENTS

Due to the scale and scope of the project, the systems diagram only includes a brief heading of each component that is required. Therefore, I will explain and justify the necessity of each component in more detail separately as to ensure the readability of the systems diagram.

SETTINGS

- **Boolean Algebra**
  - Switch Algebra Modes
    - » The ability to switch between Boolean algebra notations will allow users to become comfortable with using and translating between them, as well as ensuring the software is approachable to all users no matter their prior knowledge.
- **Clocks**
  - Change Timings
    - » Giving users the ability to change the timings on clocks that they have in their circuit will allow their creations to be far more complex and intricate. This is also a feature that cannot be found in current simulation software solutions.
- **Colour Scheme**
  - Switch Colour Scheme
    - » Allowing users to customise their interface makes the software more personal and ensures that users won't be put off the simulation by a dislike of the colour scheme. It also allows for users who require high contrast, or similar visual requirements, to make use of the software; as the software being accessible to all is of the upmost importance.

SIMULATION SANDBOX

- **Gates**
  - Create
    - » It is necessary that users have the ability to create logic gates as this allows them to build logic circuits and make use of the simulation.
  - Destroy
    - » Without the ability to destroy unwanted logic gates, the user interface would become overly cluttered and this would impact on the usability of the software. It is key that the interface can be cleaned by the user so they can clearly understand what components build up their circuit
  - Add Wire
    - » By allowing users to add wires between logic gates and other components, they are able to create complex logical circuits out of the basic gates. This allows them to bridge the understanding between logic circuits and Boolean expressions.
  - Update State
    - » The logic gates each have an internal state the holds their outputs, this is used to propagate the results of logical operations through the circuit. It is vital that there is a method to update the state of a gate as this allows for the circuit to be simulated accurately.
  - Display
    - » The gates must be displayed to the user so that they can see what circuit they are building, as well as showing the state of each gate so they can see the results of logical operations and understand how this logic propagates through the circuit.
  - Move
    - » For the users to be able to properly interact with simulation they must be able to move the logic gates around. This will be using drag-and-drop functionality as it is the most

intuitive to new users and therefore makes the software approachable to everyone regardless of prior usage.

- **Inputs**
  - Create
    - » It is necessary that users have the ability to create inputs as this allows them to build and control logic circuits and make use of the simulation.
  - Destroy
    - » Without the ability to destroy unwanted inputs, the user interface would become overly cluttered and this would impact on the usability of the software. It is key that the interface can be cleaned by the user so they can clearly understand what components build up their circuit.
  - Add Wire
    - » By allowing users to add wires between inputs and other components, they are able to create complex logical circuits out of the basic components. This allows them to bridge the understanding between logic circuits and Boolean expressions.
  - Update State
    - » The inputs each have an internal state the holds their outputs, this is used to supply values to the input of gates in the circuit. It is vital that there is a method to update the state of an input as this allows user interactability within the simulation.
  - Display
    - » The inputs must be displayed to the user so that they can see what circuit they are building, as well as showing the state of each input so they can see what values they are passing to gates in their circuit.
  - Move
    - » For the users to be able to properly interact with simulation they must be able to move the inputs around. This will be using drag-and-drop functionality as it is the most intuitive to new users and therefore makes the software approachable to everyone regardless of prior usage.
- **Outputs**
  - Create
    - » It is necessary that users have the ability to create outputs as this allows them to build and see the results of logical circuits.
  - Destroy
    - » Without the ability to destroy unwanted outputs, the user interface would become overly cluttered and this would impact on the usability of the software. It is key that the interface can be cleaned by the user so they can clearly understand what components build up their circuit.
  - Add Wire
    - » By allowing users to add wires between outputs and other components, they are able to create complex logical circuits out of the basic components. This allows them to bridge the understanding between logic circuits and Boolean expressions.
  - Update State
    - » The outputs each have an internal state the holds the result of the logical operation passed to them, this is used so the overall result of the circuit. It is vital that there is a

method to update the state of an output as this allows for the circuit to be simulated accurately.

- Display
  - The outputs must be displayed to the user so that they can see the result of the circuit they are building and understand how differing gates and inputs affect the overall output of a logical circuit.
- Move
  - For the users to be able to properly interact with simulation they must be able to move the outputs around. This will be using drag-and-drop functionality as it is the most intuitive to new users and therefore makes the software approachable to everyone regardless of prior usage.

- **Wires**
  - Create
    - The user must have the ability to create wires that connect components together. This allows for the creation of logic circuits out of basic logic gates. Building full circuits from simple gates allows the user to understand how Boolean logic is built up into complex statements.
  - Destroy
    - Without the ability to destroy unwanted wires, the user wouldn't be able to rebuild or edit parts of their circuit. This would inhibit their learning as they wouldn't be able to see how small changes to the circuit can have big effects on the output.
  - Display
    - The wires must be displayed to the user as this allows them to visually understand which components are joined together. This will help them trace the flow of logic through the circuit.

LOAD DATA

- **Load Circuits**
  - Load Boolean
    - The ability to load a circuit from a Boolean expression will not only allow user to import their previous work, or that of another user; but also allow them to see the circuit representation of any Boolean expression and bridge the understanding between Boolean logic and logic circuits.
  - Load Truth Table
    - The ability to load in logic circuits from a user generated truth table will allow users to generate a circuit with a given functionality even if they do not know a Boolean form of it. This allows the software to be approachable and usable to users of all skill levels

SAVE DATA

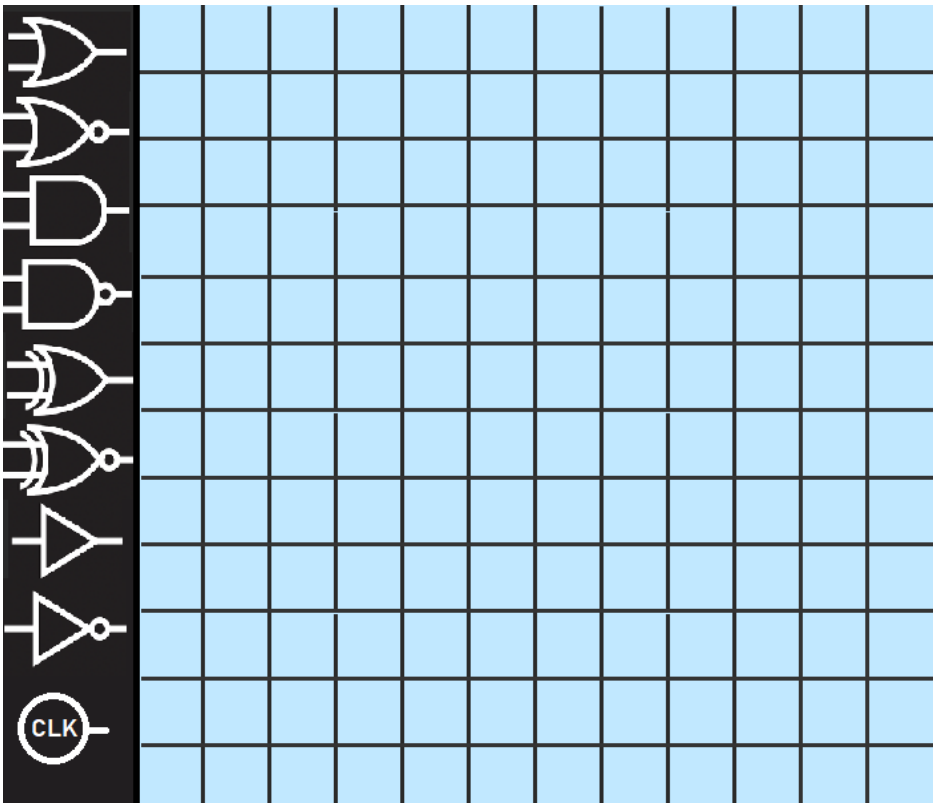
- **Output Circuits**
  - Output Boolean
    - Giving users a Boolean representation of their circuit is vital to help bridge the gap between understanding of logic circuits and Boolean expression as outlined by my

stakeholders. It also allows users a way to save any work they have done as the can import their circuit again using the Boolean expression that represents it.

- Output Written
  - The ability for users to output a written representation of the circuit they have built makes the software far more approachable to new users, as no understanding of Boolean algebra is required to understand the circuit they have built. They can simply read the expression like a sentence and be able to understand the effect of the circuit.
- Output Truth Table
  - By giving users the ability to generate a truth table from any circuit that they have designed and built, they can easily see the functionality of the circuit even if they lack understanding of Boolean logic. This also allows users a concise way to share the functionality of any circuit they have designed.

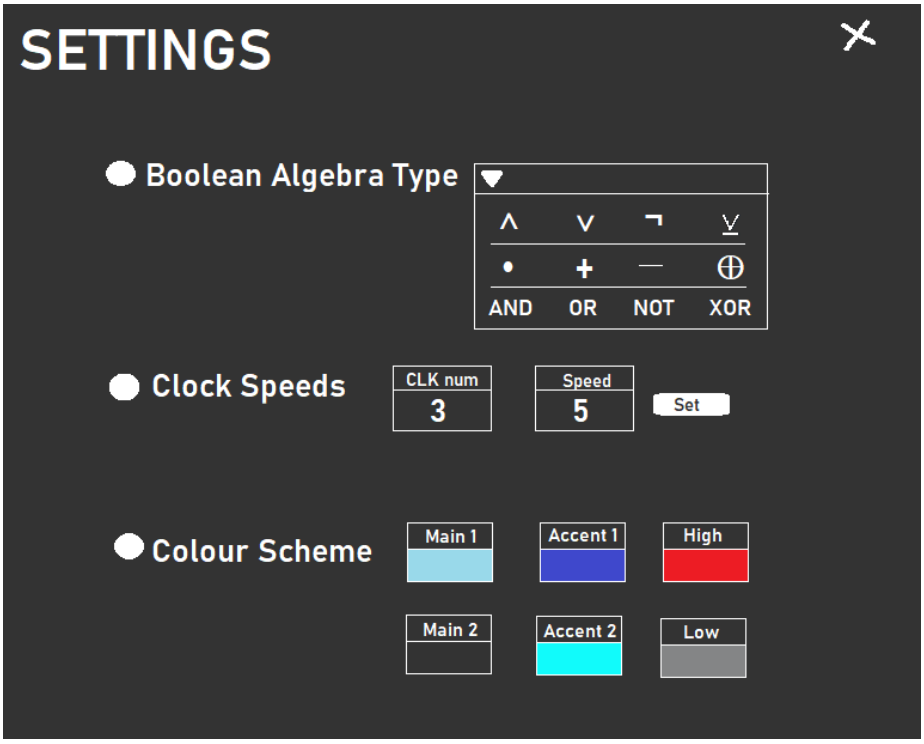
MOCK INTERFACES

SIMULATION SANDBOX



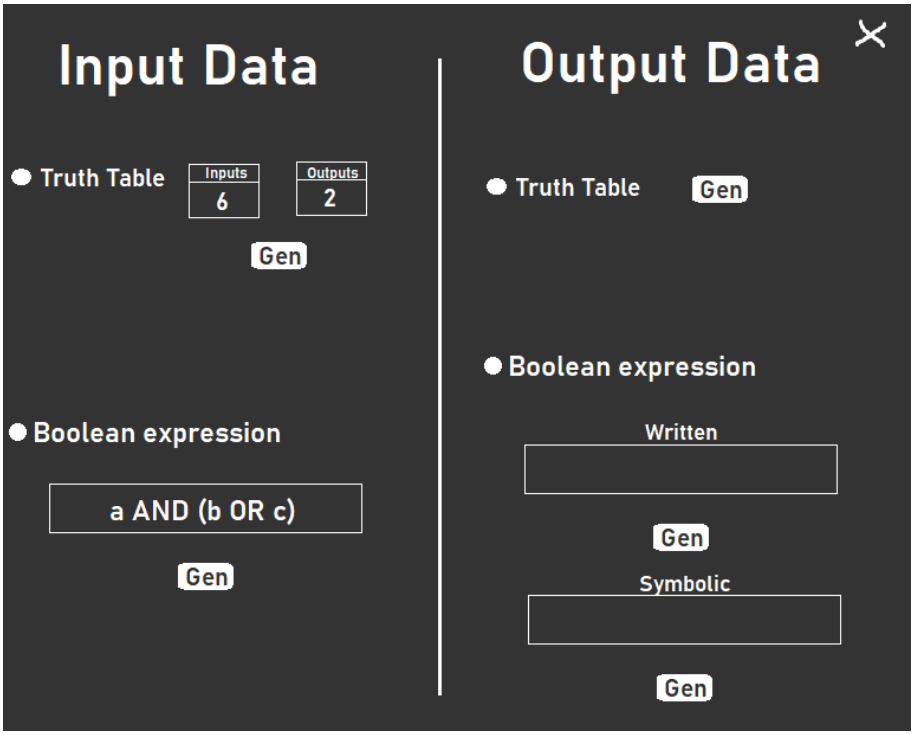
The simulation sandbox is the main screen of the software, the place where users can build, edit, and run their logical circuits. The sandbox makes use of contrasting colours and vertical divisions between the circuit area and the logic gate generators. This ensures that the user can easily distinguish between the differing parts of the screen. The background of the sandbox has a grid-pattern design, as this makes it easy for a potential user to maintain a sense of direction when scrolling around the sandbox. The unified colour scheme of the gates allows users to easily understand and quickly see and logical components that they have placed into the sandbox. This ensures that they have a full understanding of what circuit they have built so they can focus on learning how the gates interact with each other.

SETTINGS



The settings screen is a pop-up screen that will occur on user demand. This screen will allow them to make alterations to how the simulation looks and runs; catering it to their specific needs and ensuring that they get the most use out of it. The Boolean algebra symbol set can be selected from a list of several available flavours so that the user is not in a position where they don't understand the representation of their circuit due to a lack of knowledge of a specific symbol set. This allows the software to be approachable and usable by any user even if they don't have any prior knowledge. The contrasting colour scheme of the text to the background allows it to be easily read so is appropriately accessible.

DATA I/O



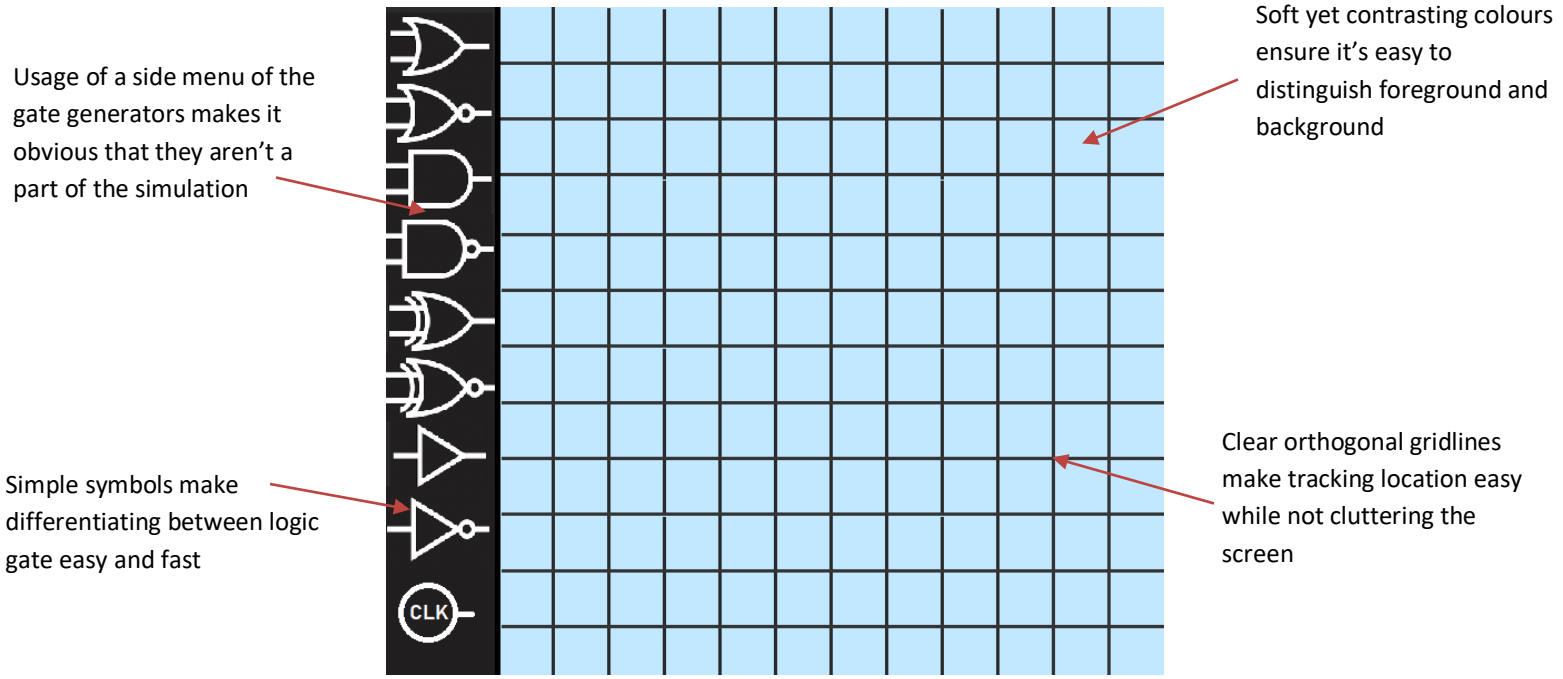
The Data I/O screen is a pop-up screen that will appear on user demand. The screen allows the user to interact with the software by inputting and receiving data. The user may input data about a circuit they wish for the simulation to create. There is the choice for the user to either input a truth table for this circuit or a Boolean expression. This ensures that users can interact with the software

even if they have limited knowledge of areas such as Boolean expressions; making the software more approachable to beginners. The screen also allows users to receive data that the software has output. This will be information about circuits that the user has built. Again, the user can choose to receive this information as either a truth table, or as a Boolean expression. The screen also gives a choice between receiving the expression in written English or in a symbolic grammar of the users choosing.

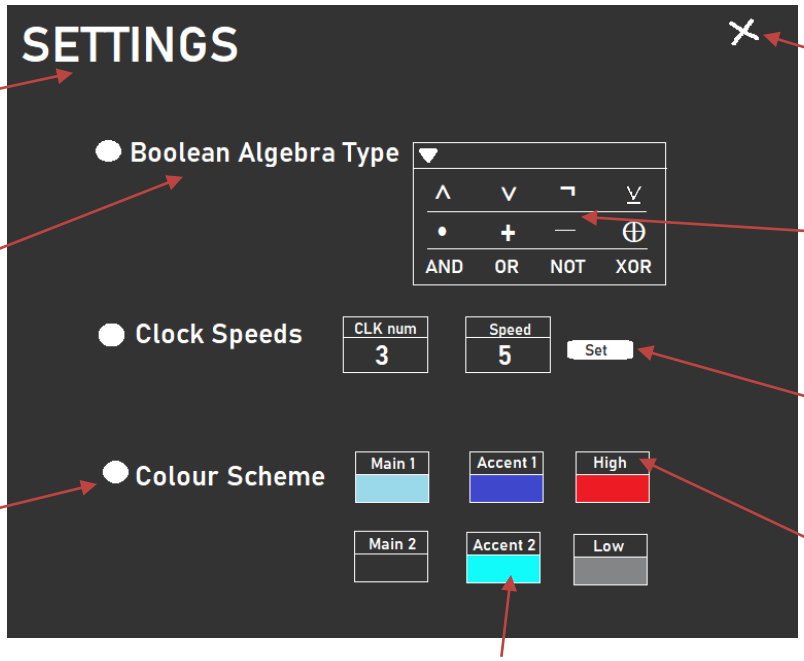
USABILITY FEATURES

The screen will be the connection between the simulation and the user controlling it, this means that it is vital that the GUI of my software is concise and easy to understand. I will make use of standard practices and common features so that any user can intuitively understand what is being displayed on screen, even if they have not used the software before. To ensure and justify that my GUI's are up to standard, I will analyze my mock interfaces and annotate them with their usability features.

SIMULATION SANDBOX



SETTINGS



Large text makes it easy to read the page title even for people with visibility issues

Simple 'X' symbol to close pop-up window

Bold text allows user to easily see what each option is

Intuitive drop-down menu allows for selection out of multiple options

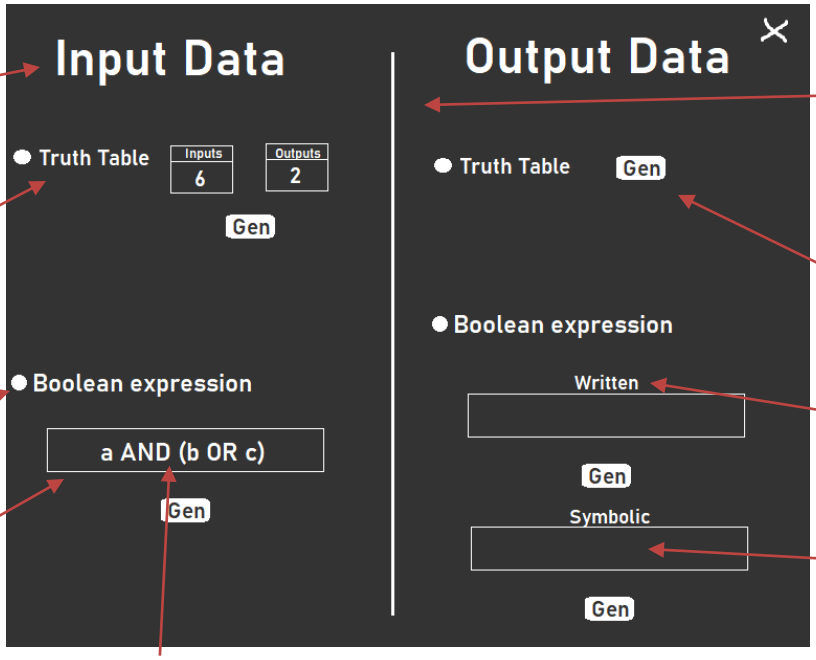
Large button with large text makes it obvious and easy to use

Bullet points make reading the options easy

Labelled input boxes make it obvious what data is expected

Visual display of the colours makes it easy to see and compare choices

DATA I/O



Large title is easy to read even for users with visibility issues

Dividing line makes clear differentiation between data input and output sections

Bold contrasting text makes options easy to read

Large button with large text makes it obvious and easy to use

Bullet points make reading the options easy

Labels make it obvious what different outputs mean

Text boxes allow for easy data input

Text boxes allow for easily understood data output

Pre-written example input makes it clear what format of data the input expects

VALIDATION OF INPUTS

The software allows the user to input data into the simulation as to make the software interactive. This data that the user enters must be validated to ensure that only permitted inputs are passed on to the software. This is done by a use of sanitisation and input validation.

Input Field	Validation Check	Justification
Boolean Algebra Type	Lookup table	The use of a drop-down list ensures that only a valid symbol set can be selected out of the options available
Clock Speeds	Presence check, Range check, Lookup table, Format Check	Both boxes must be filled in for a speed to be set. The speed must be within a valid range and the clock id must be the id of a clock that exists. The speed must be an integer
Colour Scheme	Format check	The input must be a valid colour value for it to be assigned as a colour to elements of the software
Truth Table	Presence check, Range check, Format check	Both boxes must be filled in for the table to be created. Both values must be positive and less than a max possible value
Boolean Expression	Presence check, Format check	The input must be given for a circuit to be generated from it. The input must be correctly formatted such that it can be split into inputs and gates and a circuit be generated from it.

Many actions require button presses, such as generating outputs or submitting inputs. This ensures that the user can only give valid inputs to the software as a button press is innately sanitised. This allows the software to be more usable and approachable as limited knowledge is required to use the simulation effectively.

ALGORITHMS

My software will make use of a large number of different algorithms to allow it to run correctly and efficiently. By planning and writing out the important algorithms now, I will be able to save time and effort in the development of my code, this will ensure that I can keep to my project deadlines. The designing of vital algorithms now will also ensure that the separate areas of my code will work in unison. By planning my algorithms in pseudocode, it will be easier to develop these algorithms into my software during programming. My algorithms will also be individually explained and justified in any case that isn't intuitively obvious from the code.

SETTINGS

SET CLOCK SPEED

In the settings, users are able to set the speed value for specific clocks in their circuits. This works by the user choosing a clock via its id and dictating a speed for it. This speed will be for how many frames the clock maintains its current state.

```
1  function setClockSpeed(id, speed)
2      circuit.components.get(id).setSpeed(speed)
3  end function
```

Line 2 in the function selects the clock with the given id from the components hash table, it then sets its speed to the given speed. The function assumes that the id and speed given are valid values as the sanitisation is taken care of by the input validation.

SIMULATION SANDBOX

CREATE GATE

To be able to build complex circuits, the user must have the ability to add new logic gates. So, it is necessary to have the functionality to create new gate objects. The following code creates a new gate of a specified type, such as AND, and adds it to the component hash table with the gates ID as the key.

```
1  function createGate(type)
2      gate = new Gate(type)
3      circuit.components.set(gate.getID(), gate)
4  end function
```

CREATE INPUT

For the user to be able to build complex circuits that are interactable, they must be able to add inputs to them. These inputs can either be togglable or push buttons.

```
1  function createInput(type)
2      inp = new Input(type)
3      circuit.components.set(inp.getID(), inp)
4      circuit.inputs.append(inp.getID())
5  end function
```

The function creates a new input object and adds it to both the component hash table as well as a list of all the inputs.

CREATE OUTPUT

To give the user the ability to see and understand the working of their circuit, they must be able to see the output of it. This entails that they must be able to create and add outputs to their circuits.

```
function createOutput()
    out = new Output()
    circuit.components.set(out.getID(), out)
    circuit.outputs.append(out.getID())
end function
```

DESTROY GATE

The user must have the ability to remove unwanted gates from the simulation to ensure the sandbox doesn't become over-cluttered. It is vital that when a gate is removed all of the circuit is updated so that any dependencies on the gate don't lead to an error. The function first gets the list of inputs and outputs of the specified gate to be deleted, it then removes the gate from the components hash table effectively removing it from the simulation. Following this it loops through each gate that was an input to the deleted gate and removes the deleted gate from its list of outputs. The function then loops through each of the gates that had the deleted gate as an input and removes the deleted gate from the list of inputs.

```
function destroyGate(ID)
    gate = cricuit.components.get(ID)

    inps = gate.getInputs()
    outs = gate.getOutputs()

    circuit.components.delete(ID)

    for i=0 to inps.len()
        outgoing = circuit.components.get(inps[i]).getOutputs()
        outgoing.remove(ID)
        circuit.components.get(inps[i]).setOutputs(outgoing)
    end for

    for i=0 to outs.len()
        incoming = circuit.components.get(outs[i]).getInputs()
        incoming.remove(ID)
        circuit.components.get(outs[i]).setInputs(incoming)
    end for
end function
```

DESTROY INPUT

The user must have the ability to remove any inputs that they do not want, as this will ensure that the simulation if fully interactable. To remove an input, it must itself be deleted, but it is vital that we remove any dependencies on it, as if those are not accounted for, we may run into logical errors.

```
function destroyInput(ID)
    inp = circuit.components.get(ID)

    outs = inp.getOutputs()

    circuit.components.delete(ID)
    circuit.inputs.remove(ID)

    for i=0 to outs.len()
        incoming = circuit.components.get(outs[i]).getInputs()
        incoming.remove(ID)
        circuit.components.get(outs[i]).setInputs(incoming)
    end for
end function
```

The function removes the input from both the components hash table, and the list of inputs. It then deletes the input from the circuit. Following this it loops through every component that takes it as an input and removes itself from the components list of inputs.

DESTROY OUTPUT

Giving the user the ability to remove unwanted outputs makes sure that the user has complete control over their circuit as well as preventing the screen from becoming overcluttered. This ensures that the software is easily understandable and will also allow it to run on lower spec hardware.

```
function destroyOutput(ID)
    out = circuit.components.get(ID)

    inps = out.getInputs()

    circuit.components.delete(ID)
    circuit.outputs.remove(ID)

    for i=0 to inps.len()
        outgoing = circuit.components.get(inps[i]).getOutputs()
        outgoing.remove(ID)
        circuit.components.get(inps[i]).setOutputs(outgoing)
    end for
end function
```

ADD WIRE

For the user to be able to build circuits they must be able to connect up components. The components will be connected visually with a wire so that the user can see and understand the connection, but the components are also connected logically so the software can process logic throughout the circuit.

```
function addWire(ID_A, ID_B)
    circuit.wires.append([ID_A, ID_B])

    outs = circuit.components.get(ID_A).getOutputs()
    ins = circuit.components.get(ID_B).getInputs()

    outs.append(ID_B)
    ins.append(ID_A)

    circuit.components.get(ID_A).setOutputs(outs)
    circuit.components.get(ID_B).setInputs(ins)
end function
```

The function takes in the two ID's of the components that will be connected together. It then adds component B to the list of outputs of component A, and adds component A to the list of inputs to component B.

UPDATE STATE (GATE)

Each component has an internal state, analogous to either high and low or on and off, that allows for logical operations to occur. This state must be updated to allow for logic to flow through the circuit and be shown to the user.

```
function updateState(ID)
    inps = circuit.components.get(ID).getInputs()
    vals = []
    for i=0 to inps.len()
        vals[i] = circuit.components.get(inps[i]).getState()
    end for
    newState = circuit.components.get(ID).evaluate(vals)
    circuit.components.get(ID).setState(newState)
end function
```

The function gets the list of the inputs to the specified gate. It then loops through each of these inputs and adds its state to a list called 'vals'. The new state is then calculated by the output of the gates calculate function, such as XOR, with the 'vals' as the argument. The state of the gate is then updated to this new state.

UPDATE STATE (INPUT)

The inputs must have the ability to change state otherwise the circuits that users built wouldn't be interactive. Inputs will be either togglable or push buttons. These will be updated differently.

```
function updateState(ID)
    type = circuit.componets.get(ID).getType()
    if type == "toggle" then
        if circuit.componets.get(ID).mouseOver() and mousedClicked then
            circuit.componets.get(ID).flipState()
        end if
    else
        if circuit.componets.get(ID).mouseOver() and mousedDown then
            circuit.componets.get(ID).setState(1)
        else
            circuit.componets.get(ID).setState(0)
        end if
    end if
end function
```

The toggle button checks whether it has been clicked on, and if it has its state flips. The push button checks whether the mouse is being held down over the button, and if it is the state is set to one, and its set to zero otherwise.

UPDATE STATE (OUTPUT)

The outputs must have their state be updatable for them to display the output of the circuit. This allows for the effects of the different logic gates and inputs to be seen by the user.

```
function updateState(ID)
    inps = circuit.components.get(ID)
    state = 0
    for i=0 to inps.len()
        if circuit.components.get(inps[i]).getState() == 1 then
            state = 1
        end if
    end for
    circuit.components.get(ID).setState(state)
end function
```

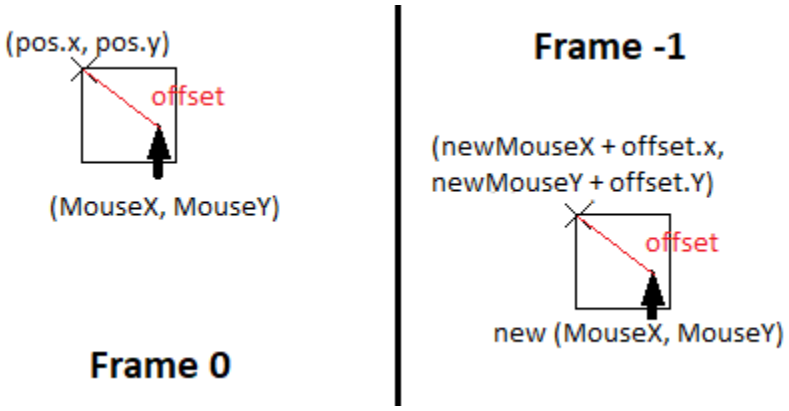
The function loops through every input to the specified output, and if the state of any of these inputs is one, then the output has its state set to one as well, if not then the state is set to zero.

MOVE COMPONENT

Each component will make use of drag-and-drop functionality. This allows the user to uniquely layout their circuit in an intuitive and easy to use manor.

```
function dragComponent(ID)
    if circuit.components.get(ID).dragging = true then
        offset = circuit.components.get(ID).getOffset()

        pos.x = offset.x + MouseX
        pos.y = offset.y + MouseY

        circuit.components.get(ID).setPosition(pos)
    end if

    if mouseOver and mousePressed then
        pos = circuit.components.get(ID).getPosition()

        off.x = pos.x - MouseX
        off.y = pos.y - MouseY

        circuit.components.get(ID).setOffset(off)
        circuit.components.get(ID).dragging = true
    else
        circuit.components.get(ID).dragging = false
    end if
end function
```

The function would be run for each component on each frame. It initially checks if the component is being dragged, if it is then it sets the new position of the component to the position of the mouse on the screen plus the offset that the mouse was to the root of the component when it began dragging. After this it checks whether the mouse is pressed down over the component, if it is it set the component to being dragged and saves the offset that the mouse is to the root of the component.



DATA I/O

LOAD BOOLEAN

A successes criterion of my software is that users can load circuits into the sandbox from Boolean expressions, this allows users to bridge the gap between Boolean algebra and logic circuits. The function should create a circuit from a given Boolean expression. The function assumes that the Boolean function is a valid input as it will have been sanitised during the validation of inputs.

```
function loadBoolean(expression)
    exp = []
    word = ""

    for i=0 to expression.len()
        if expression[i] == " " then
            exp.append(word)
            word = ""
        else
            word += expression[i]
        end if
    end for

    for i=0 to exp.len()
        if exp[i] in ["AND", "OR", "NOT", "XOR"] then
            createGate(exp[i])
        else
            createInput("toggle")
        end if
    end for

    createOutput()
end function
```

The function starts by creating an empty list and an empty string. It then loops through each letter in the given expression and either adds it to the string or, if the character is a space, adds the string to the list and resets the string to empty. This has the effect of populating the list with each word in the expression. It then loops through this list, and if the element is a gate, it creates a new gate, and if it's an input it creates a new input. It also creates an output for the circuit.

LOAD TRUTH TABLE

The user can generate a circuit from a truth table. This allows users with even no knowledge of Boolean expressions to generate circuits with specific functionality, ensuring that the software is inclusive to users of all levels.

```
function loadTruthTable(tTable)

    outs = tTable.len(1)
    ins = log(2,tTable.len(0))

    for i=0 to ins
        createInput("toggle")
    end for

    for i=0 to outs
        g = new Gate("OR")
        out = new Output()

        addWire(g.getID(), out.getID())

        for j=0 to tTable.len(0)
            if tTable[j,i] == 1 then
                a = new Gate("AND")
                addWire(a.getID(), g.getID())
                for n=0 to ins
                    onOff = ((1<<n) && j) >> n # 1 if the input is on otherwis
                    if onOff == 1 then
                        addWire(circuit.inputs[n].getID(), a.getID())
                    else
                        ng = createGate("NOT")
                        addWire(circuit.inputs[n].getID(), ng.getID())
                        addWire(ng.getID(), a.getID())
                        circuit.components.set(ng.getID(), ng)
                    end if
                end for
            end if
            circuit.components.set(a.getID(), a)
        end for
        circuit.components.set(g.getID(), g)
        circuit.components.set(out.getID(), out)
    end for

end function
```

The function uses the Sum-Of-Products approach to generating a Boolean circuit out of a truth table. This works but finding the product (ANDing) of each arrangement of the inputs that sets the output high and summing (ORing) them together. The function starts by calculating the number of inputs and outputs given the length of the 2D array ‘tTable’ which has the form that each element is a list of whether each output is high or low (e.g. [0,1]) given the inputs are high or low depending on the binary of the current index (e.g., index 2 is index 10 in binary therefore the first input is high and the second is low). It then creates the necessary number of inputs. After this, it starts a loop over each output and creates the output as well as the OR gate that sums the products. In then loops through

each possible arrangement of inputs and checks if the output is high for it. If it is, then it creates a new AND gate and connects it to the output. It then loops through each input and uses bit manipulation to calculate whether the input is high or low. If it is high then it connects the input to the AND gate, but if it is low, it creates a not gate and

Sensor Inputs

A	B	C	Output	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\bar{A}BC = 1$
1	0	0	0	
1	0	1	1	$A\bar{B}C = 1$
1	1	0	1	$AB\bar{C} = 1$
1	1	1	1	$ABC = 1$

$$Output = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

connects the input to the not gate and the not gate to the AND gate. It then adds each new gate to the components hash table. This then repeats for each output.

OUTPUT BOOLEAN

After designing and creating a circuit, the user is given the ability to generate a Boolean expression that is equivalent to the circuit. This allows the users to understand the duality between logic circuits and Boolean expressions, as well as giving them an easy way to share circuits they have created.

```
function generateBoolean()
    expression = ""
    for i=0 to circuit.outputs.len()
        out = circuit.outputs[i]
        tree = new Tree(out)
        tree = buildTree(tree, out)
        tree.root = new Gate("OR")
        expression += out + " = " + tree.inOrderTraversal() + ", "
    end for
    return expression
end function

function buildTree(tree, comp)
    inps = comp.getInputs()
    for i=0 to inps.len()
        child = circuit.components.get(inps[i])
        tree.node(comp).addChild(child)
        if child.getInputs().len() != 0 then
            buildTree(tree, child)
        end if
    end for
    return tree
end function
```

The function starts by initialising an empty string, it then loops through each output and generates a tree that represents the circuit. This is aided by an auxiliary function 'buildTree' that works recursively to add each element's inputs as child nodes. It then replaces the output with an or gate to represent how any of the output's inputs being high will set it high. The tree is then converted into a string using in order tree traversal and is added to the overall expression. This is repeated for each output.

OUTPUT TRUTH TABLE

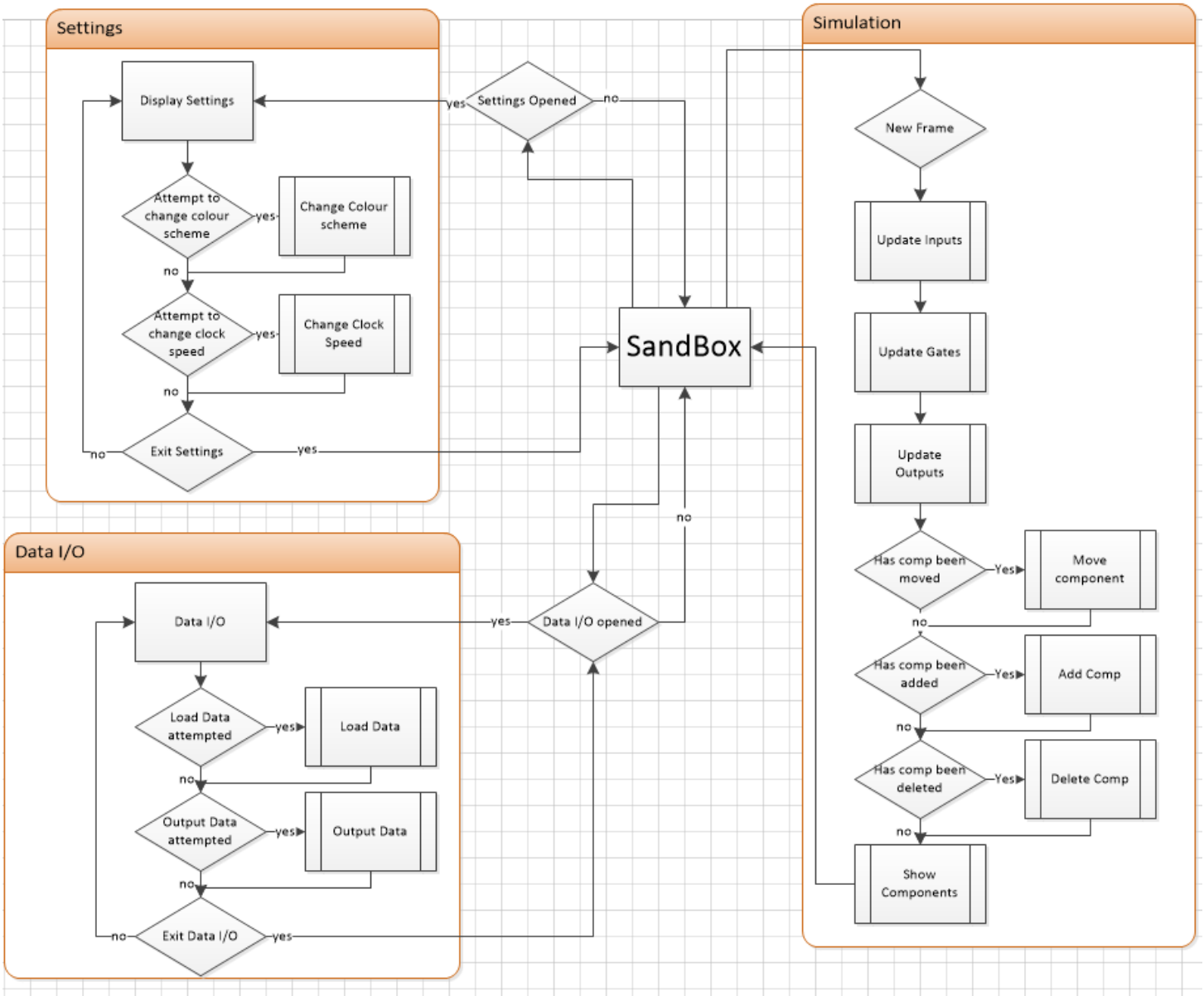
The user can generate a truth table from the circuit that they design. This allows them to easily see the full functionality of their circuit as well as being easily able to share the functionality of their circuit even if they don't have a full understanding of Boolean expressions. This ensures that the software is usable by beginners and experts alike.

```
function generateTruthTable()
    tt = []
    inps = []
    exp = generateBoolean()
    exp = exp.split(",") # changes to array where the string is cut at commas

    for i=0 to exp.len()
        exp[i] = exp[i].split(" ") # converts string a array of each word
        for j=2 to exp[i].len()
            if exp[i,j] in ["AND", "OR", "XOR", "NOT"] then
                pass
            else if exp[i,j] not in inps then
                inps.append(exp[i,j])
            end if
        end for
        exp[i] = exp[i].join(" ") # converts the array back to a list
    end for


    for i=0 to 2**inps.len()
        out = []
        for n=0 to inps.len()
            inps[n] = i.toBinary()[n]
        end for
        for j=0 to exp.len()
            out[j] = calc(exp[j],inps)
        end for
        tt[i] = out
    end for

    return tt
end function
```

The function starts by generating a Boolean expression for the circuit. It then finds and records all the required inputs. The function then loops through all possible combinations of these inputs and calculates whether each output would be high or low, recording this in a two-dimensional array.

SUB-PROCEDURE DIAGRAM

The program is made up of many smaller procedures and functions. This enables the software to be programed faster and more efficiently as code can be reused in multiple locations and editing to a single part of the program would affect any other areas unintentionally. By using a diagram to lay out how these sub-procedures will interact and be accessed will make it easier to develop the software as well as ensuring that all interacting areas of the software are compatible.



TESTING OF ALGORITHMS

To ensure that during iterative development I am able to test my software and find any logical or syntaxial errors I will produce a set of tests for my algorithms. These tests will include the test data, as well as a trace table of the function where applicable, along with its expected output. This will make it quick and easy to test my algorithms during iterative development as I will be able to enter the test data and compare the output to the expected

output; and if they don't match up I will be able to use the trace table to see where the error has occurred and therefore I will be able to promptly fix the bug

TEST DATA

Algorithm	Test Data	Expected Output	Justification
Set Clock Speed	(ID, 10) (ID, 1) (ID, 0) (ID, "fast")	Set clock speed to 10 Set clock speed to 1 Don't change clock speed Don't change clock speed	The speed of 10 is a valid input so the speed should be changed to 10, the speed of 1 is also a valid input but it is boundary data so useful to test for, 0 is an invalid speed so no change should occur, the same is true for the string "fast"
Create Gate	"OR" "Toggle" 123	Create an OR gate Don't create any gate Don't create any gate	The type "OR" is a valid type so should create a gate with type "OR", however the type "Toggle" is an invalid type for a gate but a valid type for an input so it should be tested to ensure no gate is created. 123 is erroneous data as type should be a specific string so should be tested
Create Input	"Toggle" "AND" 123	Create a toggle switch Don't create any input Don't create any input	The type "Toggle" is a valid input so we should expect a toggle switch to be created. "AND" is a valid type for a gate but not for an input so should be tested to ensure no input is created. The type 123 is the incorrect data type so should be tested to ensure no input is created
Destroy Gate	ID "test"	Delete the gate with the ID Don't delete any gate	The function should only accept valid ID's of gates, so when tested with a gates ID it should delete the gate. However, when tested with an invalid ID, such as that of another component or an entirely fabricated one, no action should be taken

Destroy Input	ID "test"	Delete the input with the ID Don't delete any input	When given a valid ID of an input, the algorithm should delete the input with that ID. But when given an invalid ID, no action should be taken
Destroy Output	ID "test"	Delete the output with the ID Don't delete any output	If the function is supplied with a valid ID of an output, then the output should be deleted. However, if the ID is not a valid output ID then no actions should be taken
Add Wire	(ID_1, ID_2) (ID_1, ID_1) ("data", ID) ("inp", "test") (ID_output, ID) (ID, ID_input)	Create a wire between the components Create a wire from component to itself Don't create a wire Don't create a wire Don't create a wire Don't create a wire	When the function receives two distinct valid ID's, then it should create a wire between the two components, however if the function receives the same ID twice it should connect the output of the component to its own input, this must be tested to ensure that it doesn't crash. If the algorithm only receives one valid ID or it doesn't receive any ID's then no actions should be taken. If the first ID is one of an output then no wire should be created as outputs don't have outputs. Equivalently if the second ID is that of an input then no wire should be created as inputs can't have inputs
Update Gate	ID "test"	Update the state Don't update the state	The update state function should update the state of a gate if a valid gate ID is supplied to it. However, if an invalid ID is supplied to it then no action should occur
Update Input	ID (toggle) ID (push) "test"	Update the state Update the state Don't update the state	If the function receives a valid ID for an input, either a toggle or a push, then it should update the state of the input depending on whether specific conditions are met. If the supplied ID is

			invalid then no actions should be taken
Update Output	ID	Update the state	If supplied with a valid ID for an output, then the algorithm should update the state of the output. But if an invalid ID is received then no action should be taken by the function
	“test”	Don’t update the state	
Move Component	ID	Update the position	The function should receive the ID of a component in the sandbox, if it does then it should update the position of the component if it has been dragged. However, if an invalid ID is received then no action should occur
	“test”	Don’t update the position	
Load Boolean	“A AND (B OR C)”	Create the Boolean circuit	The algorithm converts appropriate Boolean expression from strings to a circuit in the sandbox. If it receives a valid circuit then it should create that circuit. Just an input is still a valid expression so should be tested for to ensure that no errors occur due to lack of gates. Just a gate is an invalid expression so should be tested to ensure no circuit is built. An empty string is not a valid expression so should lead to no circuit being created. An integer is an invalid expression so no circuit should be built
	“A”	Create the Boolean circuit	
	“AND”	Don’t create a circuit	
	“”	Don’t create a circuit	
	123	Don’t create a circuit	
Load Truth Table	[[0], [1], [1], [0]]	Create an XOR circuit	The function expects to take a 2D array as inputs, so the first test data should create a circuit with the functionality of the XOR gate, this should be tested as the inner length of the array is only one. The second input is a 2D with an inner length greater than one so should be tested to
	[[0,1], [0,0], [0,0], [1,0]]	Create an AND circuit along with a NAND circuit	
	[]	Create no circuit	

	[[2,1], [0,0]]	Create no circuit	ensure that the function still works with data. The third input is an empty list which could be seen as a 2D array with outer length 0, so no circuit could have this behavior so no circuit is created.
	[1,1]	Create no circuit	The next input is a correct 2D array, but the elements in the inner arrays must be either 0 or 1 so no circuit should be created.
	"data"	Create no circuit	The next input is not a 2D array but just a single array so no circuit should be created instead of creating an AND circuit.
	1100	Create no circuit	The next data inputs are strings and integers so incorrect data type so should be tested to ensure no action occurs with erroneous data

TRACE TABLES

Destroy Gate

ID	Inps	Outs	i	Outgoing	Incoming
ID	[ID1, ID2, ID3]	[ID4, ID5]	0	[ID, ID6]	
ID	[ID1, ID2, ID3]	[ID4, ID5]	0	[ID6]	
ID	[ID1, ID2, ID3]	[ID4, ID5]	1	[ID7, ID]	
ID	[ID1, ID2, ID3]	[ID4, ID5]	1	[ID7]	
ID	[ID1, ID2, ID3]	[ID4, ID5]	2	[ID9, ID, ID8]	
ID	[ID1, ID2, ID3]	[ID4, ID5]	2	[ID9, ID8]	
ID	[ID1, ID2, ID3]	[ID4, ID5]	0		[ID, ID_10]
ID	[ID1, ID2, ID3]	[ID4, ID5]	0		[ID_10]
ID	[ID1, ID2, ID3]	[ID4, ID5]	1		[ID_11, ID_12, ID]
ID	[ID1, ID2, ID3]	[ID4, ID5]	1		[ID_11, ID_12]

The initial loop from row 0 to 5 removes the ID of the deleted gate from the output list (**outgoing**) of all its inputs (**inps**). The second loop from row 6 to 9 removes the ID of the deleted gate from the input list (**incoming**) of all the gates it outputs to (**outs**), this ensures that no errors occur within the other components which had depended on the deleted gate.

Add Wire

Wires	Outs	Ins
[[ID1, ID2], [ID_A, ID4], [ID5, ID_B]]	[ID4]	[ID5]

[[ID1, ID2], [ID_A, ID4], [ID5, ID_B], [ID_A, ID_B]]	[ID4]	[ID5]
[[ID1, ID2], [ID_A, ID4], [ID5, ID_B], [ID_A, ID_B]]	[ID4, ID_B]	[ID5, ID_A]

The first row is where the function selects the wires array as well the outputs of component A and the inputs to component B. In the second row, a wire has been added between component A and B and this is added to the wires array. In the third row, the outputs of component A have component B added to them and the inputs to component B have component A added to them.

Update State (Gate)

Inps	Vals	i	newState
[ID1, ID2, ID3]	[]		
[ID1, ID2, ID3]	[1]	0	
[ID1, ID2, ID3]	[1,0]	1	
[ID1, ID2, ID3]	[1,0,1]	2	
[ID1, ID2, ID3]	[1,0,1]		0

The first row selects the inputs of the component to up updated, rows 1 through 3 then loop through each of the inputs and the vals array is appended with their state. In row 4 the new state is calculated from the input values to the gate, in this case a state of zero as the gate is an AND gate.

Load Boolean

Expression	Exp	Word	i	Exp[i]
“A AND B”	[]	“”	0	
“A AND B”	[]	“A”	0	
“A AND B”	[“A”]	“”	1	
“A AND B”	[“A”]	“A”	2	
“A AND B”	[“A”]	“AN”	3	
“A AND B”	[“A”]	“AND”	4	
“A AND B”	[“A”, “AND”]	“”	5	
“A AND B”	[“A”, “AND”]	“B”	6	
“A AND B”	[“A”, “AND”, “B”]		0	“A”
“A AND B”	[“A”, “AND”, “B”]		1	“AND”
“A AND B”	[“A”, “AND”, “B”]		2	“B”

From rows 1 to 7 the expression is broken down into words which are individually stored in the array exp. This array is then looped through and depending on the word either a gate or an input is created, in this case two inputs would be ANDed together.

Load Truth Table

tTable	Ins	Outs	i	j	n	onOff
[[1,0], [1,0], [0,0], [1,1]]	2	2	0	0		
[[1,0], [1,0], [0,0], [1,1]]	2	2	0	0	0	0
[[1,0], [1,0], [0,0], [1,1]]	2	2	0	0	1	0
[[1,0], [1,0], [0,0], [1,1]]	2	2	0	1	0	1
[[1,0], [1,0], [0,0], [1,1]]	2	2	0	1	1	0
[[1,0], [1,0], [0,0], [1,1]]	2	2	0	2		
[[1,0], [1,0], [0,0], [1,1]]	2	2	0	3	0	1

[[1,0], [1,0], [0,0], [1,1]]	2	2	0	3	1	1
[[1,0], [1,0], [0,0], [1,1]]	2	2	1	0		
[[1,0], [1,0], [0,0], [1,1]]	2	2	1	1		
[[1,0], [1,0], [0,0], [1,1]]	2	2	1	2		
[[1,0], [1,0], [0,0], [1,1]]	2	2	1	3	0	1
[[1,0], [1,0], [0,0], [1,1]]	2	2	1	3	1	1

From lines 1 to 7 the logic for the first output is created. The algorithm loops through the first element of each sub-list in "tTable" and if the element is 1, then it loops for each input to that circuit (in this case there are two inputs as the list has length 4). It then calculates whether that input should be set high or low for this current position in the list, and either connects and the input or the NOTed input to an AND gate which is connected to the circuits main OR gate. If the first element in the sub list is 0, such as in line 5, then no gate is connected so no loop is required. From line 8 to the end it repeats this same process but for the second output, using the second element in each sub-list.

VARIABLES AND DATA STRUCTURES

Throughout my program I will make use of a vast number of variables. I will also be taking an object-oriented approach to my software so I will have many classes as well. By laying out the key variables and classes that I will use, it will make my development much easier and faster as I can reference back to it when programming to see what variables I will need to make use of.

CLASS DIAGRAMS

The classes that I will use will all have attributes and methods. By setting out the attributes and methods that these classes will poses here, it will be far easier to implement the classes when it comes to programming my software. The explanation and justification of the class methods is covered in the algorithms section where, following the use of them, their effect and necessity was described in detail.

CIRCUIT

- **Attributes**
  - Components: Hash table
  - Inputs: Array of integers
  - Outputs: Array of integers
  - Wires: 2D array of integers
- **Methods**
  - getInputs()
  - getOutputs()
  - setInputs()
  - setOutputs()

GATE

- **Attributes**
  - Inputs: Array of integers
  - Outputs: Array of integers
  - ID: Integer
  - Type: String
  - Position: Vector
  - Offset: Vector
  - Dragging: Boolean
  - State: Integer
- **Methods**
  - getInputs()
  - getOutputs()

- getID()
- getPosition()
- getOffset()
- getState()
- mouseOver()
- evaluate()
- setInputs()
- setOutputs()
- setPosition()
- setOffset()
- setState()

INPUT

- **Attributes**
  - Outputs: Array of integers
  - ID: Integer
  - Type: String
  - Position: Vector
  - Offset: Vector
  - Dragging: Boolean
  - State: Integer
- **Methods**
  - getOutputs()
  - getID()
  - getPosition()
  - getOffset()
  - getState()
  - getType()
  - mouseOver()
  - setOutputs()
  - setPosition()
  - setOffset()
  - setState()
  - flipState()

OUTPUT

- **Attributes**
  - Inputs: Array of integers
  - ID: Integer
  - Position: Vector
  - Offset: Vector
  - Dragging: Boolean
  - State: Integer
- **Methods**
  - getInputs()
  - getID()
  - getPosition()
  - getOffset()
  - getState()
  - mouseOver()
  - setInputs()
  - setPosition()
  - setOffset()
  - setState()

KEY VARIABLES

Variable	Data Type	Justification and Validation
circuit	Circuit	The circuit variable will be used to store the components in a user-built circuit as well as the connections between them, this is essential as it allows for the flow of logic throughout the circuit to be calculated

sandboxWidth	Int	The width of the sandbox area will be stored as an integer. This will allow the area to be drawn as well as have the size of the sandbox be edited by the user to allow for a more interactive and immersive feel
sandboxHeight	Int	The height of the sandbox area will be stored as an integer. This will allow the area to be drawn as well as have the size of the sandbox be edited by the user to allow for a more interactive and immersive feel
colourScheme	Array of colours	By storing the user selected colours to be used by the software in a variable, it allows the users to mix and match colours to their preference, this ensures accessibility even to people with visual problems
frameCount	Int	This stores the current frame count, in each 'frame' the logic of the circuit will be calculated and displayed to the user. It is vital to keep a record of the frame count as this allows both clocks to work and for users to select a slower speed for lower spec hardware
MouseX	Int	Storing the x position of the mouse within the window allows it to be used to detect when the mouse is being placed over specific objects, or when the mouse has moved, or when it is in certain regions of the screen
MouseY	Int	Storing the y position of the mouse within the window allows it to be used to detect when the mouse is being placed over specific objects, or when the mouse has moved, or when it is in certain regions of the screen
zoomScale	Float	This stores the scale factor by which the sandbox has been zoomed into or out off. This allows users to work on circuits of any size and therefore allows the software to be approachable to users of any skill level

FURTHER TEST DATA

To ensure that the software I produce is effective in solving the initial problem, it will have to be tested for its functionality. This acceptance testing will be carried out by a mixture of myself as well as the stakeholders to ensure it meets everyone's satisfaction. By having a plan set out for this post-development testing it will allow it to be carried out much more efficiently and ensure no areas are missed. As some of the criteria to be tested could be seen as subjective, multiple people may be required to test specific areas to ensure that an accepted objective decision can be concluded

Criterion	Evaluated By	Explanation/Justification
Allows the user to build their own logic circuits	Stakeholders	The stakeholders will create logic circuits using the software to ensure the software is easily usable
Correctly calculate logic for NOT, AND, OR, and XOR gates	Me	I will create circuits using these gates and test the circuit to ensure that the logic is correct
Allow the user to control the inputs to a circuit	Stakeholders	The stakeholders will create circuits with multiple different inputs to ensure that they are easily usable
Allow the user to see the outputs of a circuit	Stakeholders	The stakeholders will run circuits and observe the output to ensure it is easy to use and understand
Allow the use of clocks as inputs to the circuit	Stakeholders	The stakeholders will create circuits using clocks to ensure that they are easy to implement and use
Allow the user to edit the speed of clocks	Stakeholders	The stakeholders will create circuits using clocks and edit the speed of those clocks to ensure it is intuitive to use
Allow the user to generate a Boolean expression of their circuit	Me	I will create circuits and then generate the Boolean expression of them to ensure it is correct and easy to use
Allow for the use of multiple Boolean algebra symbol sets, and the ability to switch between them	Me	I will test to see whether it is simple to switch between algebra sets
Allow the user to edit a circuit while its running	Me	I will create circuits and edit them while the simulation is running to ensure that the circuits work as expected
Allow the user to generate a truth table of a circuit	Me	I will create a circuit then generate a truth table from the circuit to ensure it is correct and the system is intuitive to use
Allow the user to generate a circuit from a Boolean expression	Me	I will input a Boolean expression and ensure that the correct circuit is generated and that the system is easy to use
Allow the user to generate a circuit from a truth table	Me	I will input a truth table and ensure the created circuit is correct as well as the system for inputting the table is easy

		to use
Drag and drop control of the logic gates	Stakeholders	The stakeholders will create and edit circuits using the drag and drop controls to ensure that they feel intuitive
Uncluttered and readable interfaces	Stakeholders	The stakeholders will use the software and navigate the different interfaces to ensure they are uncluttered and readable

DEVELOPMENT

During my development I will take an iterative approach as this will allow me to work on specific parts of my software at a time, as well as having a product to show my stakeholders. This will allow me to gain their feedback and input on the software. It will also allow me to set attainable and recordable goals while sticking to a workable timeline.

ITERATION ONE

DEVELOPMENT

During my first Iteration I will focus on the main building blocks of the program. This will consist of implementing the logic gates, inputs, and outputs; as well as ensuring that they properly interact with each other. This will require the creation of classes for these components with the necessary procedures to allow for their correct function.

The first class I will create will be an overarching parent class that all components will inherit from. This class will allow for the drag and drop functionality of the components to ensure that the software is intuitive to use.

```
3  class Draggable{
4
5    constructor(x,y,w,h){
6      this.pos = createVector(x,y) // a vector of the position
7      this.w = w // the width of the comp
8      this.h = h // the height of the comp
9      this.offset = createVector(0,0) // the intital offset is 0
10     this.dragging = false
11   }
```

The constructor for this class will take four values, the x and y position of the component as well as the width and height of it. The class will have 5 attributes that are a vector called pos the stores the location of the component. The width and the height of the component (w, h). A vector called offset that is initialized to (0,0) will be used to store the location of the object that the mouse began dragging from. A Boolean called dragging also stores whether or not the object is in its dragging state.

```
  //a function that returns a bool dependant on whether the mouse is over the comp
  mouseOver(){
    return mouseX > this.pos.x && mouseX < this.pos.x + this.w && mouseY > this.pos.y && mouseY < this.pos.y + this.h
  }

  //a function that is run each frame and updates the position of the comp
  updatePos(){
    if (this.dragging){
    this.pos.x = this.offset.x + mouseX
    this.pos.y = this.offset.y + mouseY
    }
  }

  //a function run when the component is clicked on
  pressed(){
    if (this.mouseOver()){
      //set the offset to the position the mouse is over the comp
      this.offset.x = this.pos.x - mouseX
      this.offset.y = this.pos.y - mouseY
      this.dragging = true
    }
  }

  //a function run when the mouse is depressed
  released(){
    this.dragging = false
  }
```

The draggable object also has several functions that allow for the drag-and-drop capabilities to be used. A mouse over function returns a Boolean of whether or not the mouse is over the object. The update position function checks whether the object is being dragged, and if it is, it updates the position of the object to the mouse position plus the offset. The pressed function is called when the mouse is clicked, it checks whether the mouse is over the object, if it is then it calculates the mouse offset and sets the dragging state to true. The released function is called when the mouse is released and sets the dragging state to false.

The gate class will inherit from the draggable class, this will allow it to be moved using the mouse by the user. The class will hold the functions and attributes required to simulate a logic gate.

```
//a general class for gate components
class Gate extends Draggable{
  constructor(x, y, w, h, type, maxInputs=10){
    super(x,y,w,h) //initialise the draggable object
    this.inputs = [] //create an array for inputs to the gate
    this.outputs = [] //create an array for outputs to the gate
    this.maxInputs = maxInputs //have a maximum num of inputs such as 2 for XOR
    this.type = type //define the gate type such as "AND"
    this.state = 0 //set the state of the gate to 0 (low)
    this.calcState = null //store the calculated state so it doesnt need to be re calculated in the same frame
  }
```

The constructor for the class will take the four arguments required for a draggable object, as well as two others. A required argument called type, and an optional argument called maxInputs whose initial default value is set to an arbitrary 10. The constructor firstly initialises the draggable object and then defines 6 of its own attributes. These include two arrays for the inputs and outputs of the gate. An integer that stores the maximum number of inputs the gate can receive. The type of the gate is stored as a string. The state of the gate is an integer and is initially set to 0. A variable calcState is also set to null, the purpose of this variable is for optimisation as well as the implementation of short circuits and is explained in the update function.

```
24  //update the state of the gate depending on the inputs to it and its type
25  updateState(){
26    //first update all of the input nodes
27    let inps = []
28    for (i=0; i<this.inputs.length; i++){
29      if (this.inputs[i]!=this){
30
31        if (this.inputs[i].calcState != null){
32          inps.push(this.inputs[i].calcState)
33        }
34        else{
35          this.inputs[i].updateState()
36          inps.push(this.inputs[i].state)
37        }
38
39      }else{inps.push(this.state)}
40    }
41    //then update this node using the states of its inputs
42    this.state = (this.inputs.length>0) ? funcs.get(this.type)(inps) : this.state
43    this.calcState = this.state
44  }
```

The gate class has a function that updates its state. This function works recursively, calling itself on the inputs to the gate, in a similar manner to traversing a graph. The function initially sets an array for the state of all the inputs. It then loops through each input of the gate, if the input is not the gate itself, it checks whether the state of that gate has been calculated already this frame. If the value has been calculated then it adds this value to the input array, however, if it has not been calculated this frame then the update state function is called on the child. The now calculated state is pushed onto the input array. If the input is the gate itself then the current state of the gate is put on the array and the function is not called again. The state is then updated to the required value using the calculated inputs and the activation function of the gate, depending on the precondition of having a positive number of inputs. The function then also sets the calcState variable for this frame.

In the update state function the activation function of the gate is called to calculate its new state. All of the required activation functions are defined outside of the class in a hash map, where the keys are the gate type and the values are the activation function.

```
1  //define the function of each gate
2  let funcs = new Map()
3  funcs.set("AND",(inps)=>{return inps.reduce((a,b)=>a&b)})
4  funcs.set("OR",(inps)=>{return inps.reduce((a,b)=>a|b)})
5  funcs.set("XOR",(inps)=>{return inps.reduce((a,b)=>a^b)})
6  funcs.set("NOT",(inps)=>{return +!inps[0]})
7  funcs.set("NAND",(inps)=>{return +!inps.reduce((a,b)=>a&b)})
8  funcs.set("NOR",(inps)=>{return +!inps.reduce((a,b)=>a|b)})
9  funcs.set("NXOR",(inps)=>{return +!inps.reduce((a,b)=>a^b)})
```

The functions are defined inline and work by taking an array as an input, this will be the array of inputs to the gate. It then uses the reduce prototype to work through the array repeatedly applying the logic of the gate. The NOTed versions of the gate work by inverting the result of the original gates.

```
46   //display the gate
47   show(){
48     textAlign(CENTER,CENTER)
49     fill(255)
50     rect(this.pos.x, this.pos.y, this.w, this.h)
51     fill(0)
52     stroke(5)
53     text(this.type,this.pos.x+this.w/2,this.pos.y+this.h/2)
54   }
55
56   //the function called each frame that updates all parts of the gate
57   update(){
58     this.updatePos()
59     this.show()
60     this.calcState = null
61   }
```

The gate class also has functions that allow for the displaying of the gates on screen. As this is only the first iteration of the software, the overall aesthetics of the program are not as important as the functionality of the simulator, for this reason I have not spent time on beautifying the visuals. The gate also has an update function that is called each frame. This calls the functions required to allow for the drag and drop of the gate, as well as displaying the gate each frame and resetting the calcState to null.

The output class will allow for the creation and implementation of logical outputs in circuits. This will make the functionality of a user-built circuit very easy to see and understand.

```
//define a output class
class Output{
  constructor(x,y,w,h){
    super(x,y,w,h)
    this.state = 0 //the state of the output
    this.inputs = [] //the list of inputs to the output
  }
```

The constructor function for an output only takes the four arguments required by the parent draggable class. It then initialises the draggable class and defines a state, which is set to zero, and an array for inputs. Unlike the gate class the output doesn't require an array of outputs as it only takes inputs.

```
  //update the state of the output
  updateState(){
    //initially loop through each node that inputs to it and update its state
    let inps = []
    for (i=0; i<this.inputs.length; i++){

      if (this.inputs[i].calcState != null){
        inps.push(this.inputs[i].calcState)
      }
      else{
        this.inputs[i].updateState()
        inps.push(this.inputs[i].state)
      }
    }
    //then update this node using the states of its inputs
    this.state = (this.inputs.length>0) ? funcs.get("OR")(inps) : this.state
  }
```

The output class has an update state function very similar to the gate class. It loops through all of its children and loads their state into an array. The function doesn't have to check whether the input is itself as an output component cannot be and input. It then updates the state of the output using an OR activation function, so if any of its inputs are high, the output will be high. The function doesn't need to set a calcState variable as no component will take the output component as an input.

```
  //display the output
  show(){
    textAlign(CENTER,CENTER)
    fill(255)
    rect(this.pos.x, this.pos.y, this.w, this.h)
    fill(0)
    stroke(5)
    text(this.state,this.pos.x+this.w/2,this.pos.y+this.h/2)
  }

  //the function called each frame that updates all parts of the output
  update(){
    this.updatePos()
    this.show()
  }
```

The output class also has similar functions to the gate class for showing the output and updating it each frame. The display function, such as the one in the gate class, is a placeholder while I work on the more necessary functionalities of the software.

To allow for connection to be built up, a circuit class is required. This class will store all the components that are in the circuit as well as the connections between them, allowing for the flow of logic through the circuit.

```
1  //a circuit class that stores data about all of the components
2  class Circuit{
3    constructor(){
4      this.gates = [] //stores all the gates in the circuit
5      this.inputs = [] //stores all the inputs in the circuit
6      this.outputs = [] //stores all the outputs in the circuit
7      this.wires = [] //stores all the wires in the circuit
8    }
```

The constructor for the circuit takes no arguments, however, it defines four arrays. One for the gates in the circuit, one for the inputs, one for the outputs, and one for the wires. The wires array is a 2d array that stores sets of two components, the first element being the output component and the second element receiving the first as an input.

```
10   //add a new gate to the circuit by creating a gate and addding it to the gate list
11   addGate(x,y,w,h,type){
12     this.gates.push(new Gate(x,y,w,h,type))
13   }
14
15   //add a new output to the circuit
16    addOutput(x,y,w,h){
17     this.outputs.push(new Output(x,y,w,h))
18   }
19
20   //add a new output to the circuit
21    addInput(x,y,w,h,type){
22     this.inputs.push(new Input(x,y,w,h,type))
23   }
```

The circuit also contains functions that allow the for the adding of new components to the circuit. The functions for adding a gate, an output, and an input take the arguments required to create those components and pass them to the constructor function of the class.

```
25   //add wires between compnents
26   addWire(compA, compB){
27     this.wires.push([compA,compB]) //add the wire to the list of wires
28     compA.outputs.push(compB) //add compB as an output of compA
29     compB.inputs.push(compA) //add compA as an input of compB
30   }
31 }
```

The circuit class also contains a function for adding a wire between two components. The function takes the components as arguments and adds them to the wire array. It also adds the components into the respective input and output arrays of the components.

```
//display wires on the screen
function displayWire(wire){
  push()
  strokeWeight(5)
  stroke(255)
  line(wire[0].pos.x+wire[0].w, wire[0].pos.y+wire[0].h/2, wire[1].pos.x, wire[1].pos.y+wire[1].h/2)
  pop()
}
```

I have also defined a function that can display the wires. As this is only the first iteration the design is not overly important, so this will be revamped in the future. The function does make it clear where the wires are so it is suitable for the first iteration.

```
let circuit
function setup() {
  createCanvas(windowWidth,windowHeight);
  circuit = new Circuit()
  circuit.addGate(100,80,50,50,"OR")
  circuit.addGate(100,200,50,50,"AND")
  circuit.addInput(10,10,50,50,"toggle")
  circuit.addInput(10,150,50,50,"push")
  circuit.addOutput(200,80,50,50)
  circuit.addOutput(200,200,50,50)

  circuit.addWire(circuit.inputs[0], circuit.gates[0])
  circuit.addWire(circuit.inputs[1], circuit.gates[0])
  circuit.addWire(circuit.inputs[0], circuit.gates[1])
  circuit.addWire(circuit.inputs[1], circuit.gates[1])
  circuit.addWire(circuit.gates[0], circuit.outputs[0])
  circuit.addWire(circuit.gates[1], circuit.outputs[1])
}
```

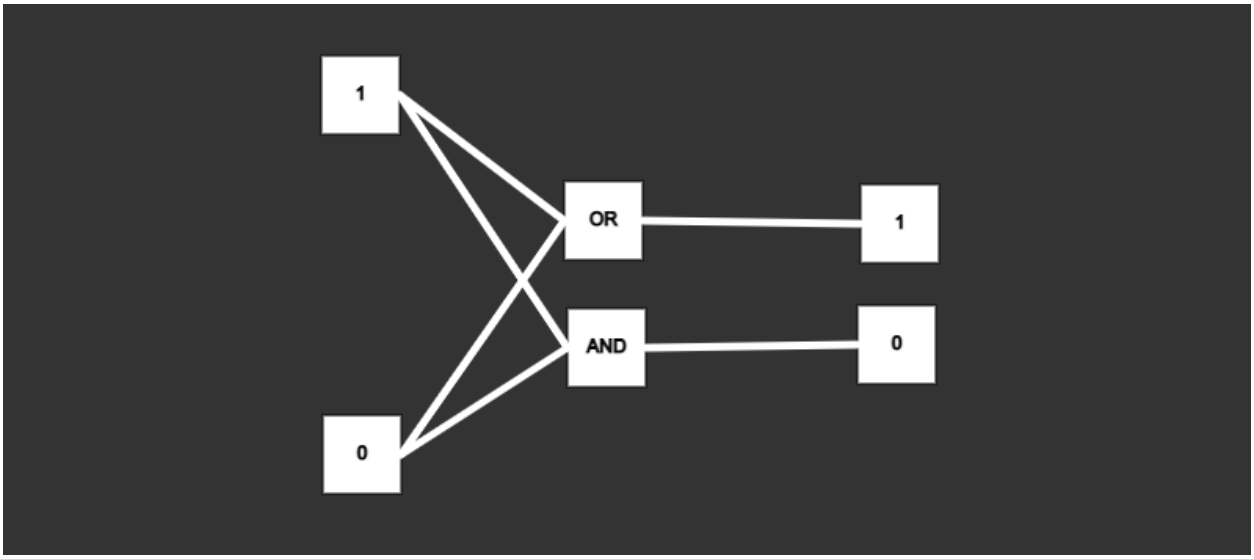
To test the functionality of the components I have set up a simple circuit to see how it runs and can ensure the logic is correctly calculated. The setup function is run once at the start of code execution, this initially creates a canvas object on the page that can be controlled by the program. A circuit object is then initialized, and has two gates, two inputs, and two outputs added to it. Wires are then connected between the components.

```
function draw() {
  background(51);

  //update the state of the comps in the circuit
  circuit.gates.forEach(gate => gate.updateState())
  circuit.inputs.forEach(input => input.updateState())
  circuit.outputs.forEach(output => output.updateState())

  //update the position etc of the comps in the circuit
  circuit.gates.forEach(gate => gate.update())
  circuit.inputs.forEach(input => input.update())
  circuit.outputs.forEach(output => output.update())
  circuit.wires.forEach(displayWire)

}
```

The draw function is called every frame. This firstly resets the canvas to uniform flat colour. Then the update state function is called for each gate, input, and output. After this the update function is called for every gate, input, and output; as well as the display wire function being called for each wire.

```
//called when the mouse is pressed
function mousePressed(){
  circuit.gates.forEach(gate => gate.pressed()) //calls the pressed function for all the gates
  circuit.inputs.forEach(input => input.pressed()) //calls the pressed function for all the inputs
  circuit.outputs.forEach(output => output.pressed()) //calls the pressed function for all the outputs
}
//called when the mouse is released
function mouseReleased(){
  circuit.gates.forEach(gate => gate.released()) //calls the released function for all the gates
  circuit.inputs.forEach(input => input.released()) //calls the released function for all the inputs
  circuit.outputs.forEach(output => output.released()) //calls the released function for all the outputs
}
```

The mouse pressed function is called each time a mouse press event occurs, in this function the pressed function of each gate, input, and output is called. Similarly, the mouse released function is called when the mouse button is released. It in turn calls the released function of each gate, input, and output.



The program produces the desired logic circuit, and the correct outputs are seen given any inputs. The components can be moved around using drag-and-drop. Both the inputs work, making use of both toggle controls and push controls. This early success gives me confidence to work on more interactive areas of the program, such as the user ability to add gates and wires to the simulation.

```
<!--the sidebar containing the components-->
<div id="sideBar">

  <!--Gates-->
  <div id="gates">
    <p id="AND" onclick="clickGate('AND')">AND</p>
    <p id="OR" onclick="clickGate('OR')">OR</p>
    <p id="XOR" onclick="clickGate('XOR')">XOR</p>
    <p id="NAND" onclick="clickGate('NAND')">NAND</p>
    <p id="NOR" onclick="clickGate('NOR')">NOR</p>
    <p id="XNOR" onclick="clickGate('XNOR')">XNOR</p>
    <p id="NOT" onclick="clickGate('NOT')">NOT</p>
  </div>

  <!--Inputs-->
  <div id="inputs">
    <p id="toggle" onclick="clickInp('toggle')">Toggle</p>
    <p id="push" onclick="clickInp('push')">Push</p>
  </div>

  <!--Outputs-->
  <div id="outputs">
    <p id="output" onclick="clickOut()">Output</p>
  </div>
```

To allow for the user to add components to the simulation, I have added a sidebar to the screen with options to add each component. These are currently using just a label as a stand in for the images that the final product will have. Each element also has an onclick function that allows for the creation of the component.

```
64 function clickGate(type){
65   //try 50 times to add the gate
66   for(let i=0; i<50; i++){
67     var accept = true
68     var pos = createVector(random(0,windowWidth-sideBar.width),random(0,height))
69     //check if gate is overlapping another component
70     for(let comp of [...circuit.gates,...circuit.inputs,...circuit.outputs]){
71       if(overlap({pos:pos, w:50, h:50}, comp)){accept=false; break}
72     }
73       //if accepted add the gate
74     if(accept){circuit.addGate(pos.x,pos.y,50,50,type);return}
75   }
76 }
```

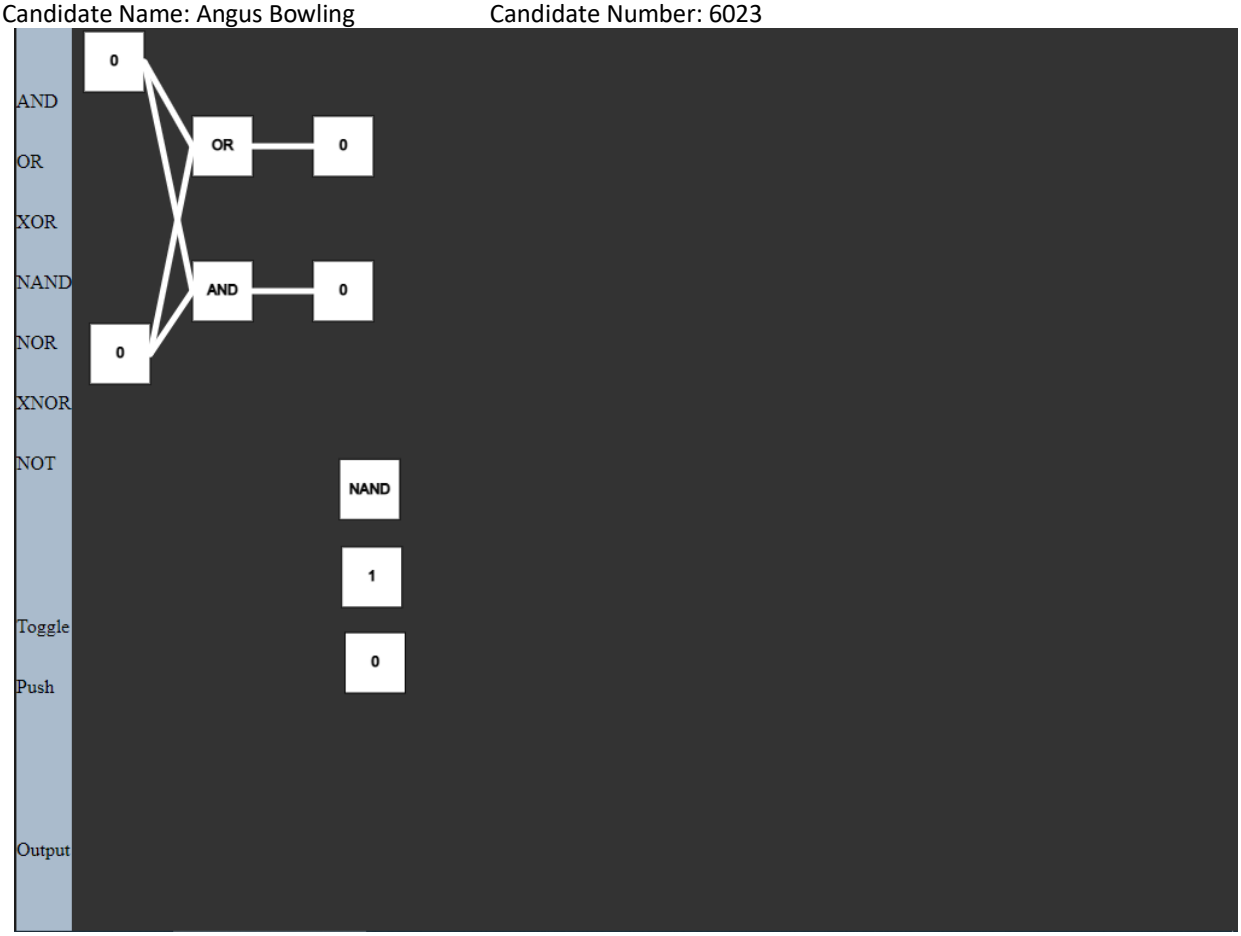
The click gate function is called when a gate element is clicked in the sidebar. It takes an argument which is the type of the gate. It then loops 50 times, and attempts to add a gate to the sandbox in a position that doesn't overlap with another component. If it is successful the gate is added however, is none of the fifty attempts work then no gate will be added.

```
77 function clickInp(type){
78   //try 50 times to add the input
79   for(let i=0; i<50; i++){
80     var accept = true
81     var pos = createVector(random(0,windowWidth-sideBar.width),random(0,height))
82     //check if input is overlapping another component
83     for(let comp of [...circuit.gates,...circuit.inputs,...circuit.outputs]){
84       if(overlap({pos:pos, w:50, h:50}, comp)){accept=false; break}
85     }
86       //if accepted add the input
87     if(accept){circuit.addInput(pos.x,pos.y,50,50,type);return}
88   }
89 }
90 function clickOut(){
91   //try 50 times to add the output
92   for(let i=0; i<50; i++){
93     var accept = true
94     var pos = createVector(random(0,windowWidth-sideBar.width),random(0,height))
95     //check if output is overlapping another component
96     for(let comp of [...circuit.gates,...circuit.inputs,...circuit.outputs]){
97       if(overlap({pos:pos, w:50, h:50}, comp)){accept=false; break}
98     }
99       //if accepted add the output
100    if(accept){circuit.addOutput(pos.x,pos.y,50,50);return}
101  }
102 }
```

The click input function is similar to the click gate function. It also takes a type as an argument and attempts to add an input to the sandbox, it tries at most 50 times at which point no input is added. The click output function is again similar to the previous two. However, it doesn't require a type to be passed as an argument.

```
42 function overlap(compA, compB){
43   return !(compA.pos.x > compB.pos.x+compB.w ||
44            compA.pos.x+compA.w < compB.pos.x ||
45            compA.pos.y > compB.pos.y+compB.h ||
46            compA.pos.y+compA.h < compB.pos.y)
47 }
```

To check whether two elements are overlapping, the overlap function can be called. The function ORs together the conditions that would make it impossible for the components to be overlapping and then returns the NOT of that.



With the sidebar added, the user has the ability to add gates, inputs, and outputs to the sandbox. While the aesthetics of the software is currently not top level, this can be developed in later iterations. The adding of components is working correctly so this takes the software closer to being truly interactive.

```
48  //add a wire from this gate to another
49  wireFrom(){
50    if (!wireDrag && (mouseX-(this.pos.x+this.w+5))**2 + (mouseY-(this.pos.y+this.h/2))**2 <= 25){
51      wireDrag = true
52      compFrom = this
53    }
54  }
55
56  //add wire to this gate from another
57  wireTo(){
58    if (wireDrag && (mouseX-(this.pos.x-5))**2 + (mouseY-(this.pos.y+this.h/2))**2 <= 25 && this.inputs.length < this.maxInputs){
59      wireDrag = false
60      circuit.addWire(compFrom, this)
61      compFrom = null
62    }
63  }
```

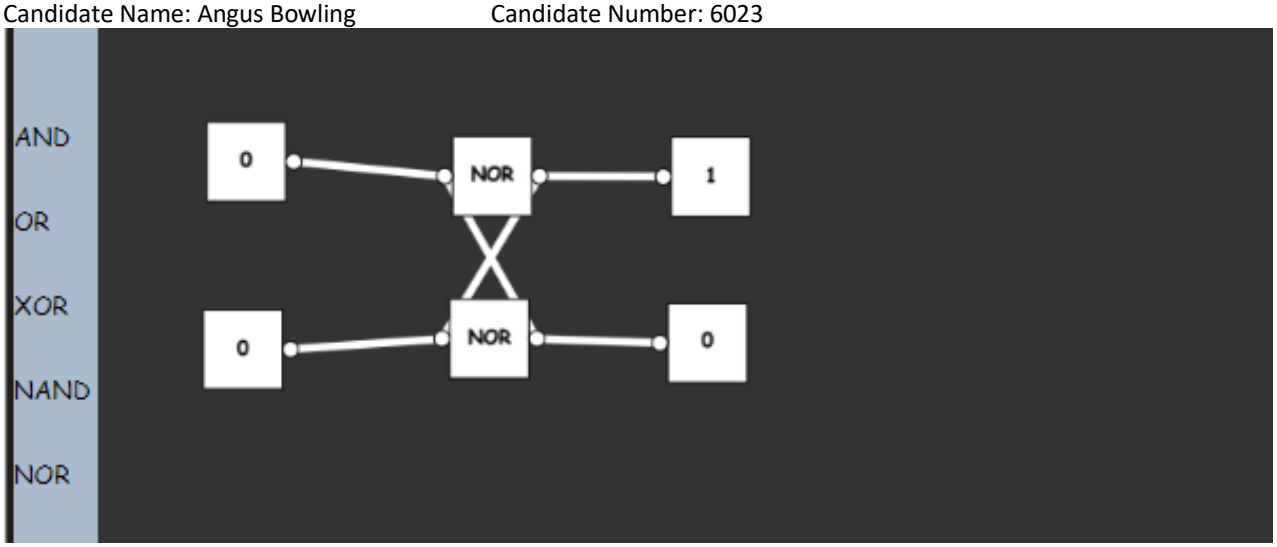
To allow for wires to be added to and from components, I have written functions that detect when the user click or release the mouse over a component's input or output area. For the wire from function, this sets the wire dragging Boolean to true and sets the component that the wire is coming from to itself. For the wire to function, the wire dragging Boolean is set to false, the wire is then added between the initial component and the end component, then the initial component variable is reset. The output class also has the wire to function but no wire from function as outputs don't output to any other components. The input class has the wire from function but not the wire to function for the opposite reason.

```
3  ▼ RangeError: Maximum call stack size exceeded
      at /gate.js:33:18
```

Upon testing user building of circuits in the sandbox, I tried to create a SR-NOR latch, when attaching a wire between the two NOR gates, the adding of the second wire caused the above error. This error comes from the update state function calling more times than the recursive stack can handle. This is because of the short circuit, and this short circuit wasn't caught by the 'calcState' check as short circuit goes through multiple components and not just directly from a component to itself.

```
25  updateState(){
26    if (this.calcState == null){
27
28      //initailly set calcState to the current state to allow for multi-comp short circuits
29      this.calcState = this.state
30
31      //first update all of the input nodes
32      let inps = []
33      for (let i=0; i<this.inputs.length; i++){
34        if (this.inputs[i]!=this){
35
36          if (this.inputs[i].calcState != null){
37            inps.push(this.inputs[i].calcState)
38          }
39          else{
40            this.inputs[i].updateState()
41            inps.push(this.inputs[i].state)
42          }
43
44        }else{inps.push(this.state)}
45      }
46      //then update this node using the states of its inputs
47      this.state = (this.inputs.length>0) ? funcs.get(this.type)(inps) : this.state
48      this.calcState = this.state
49    }
50  }
```

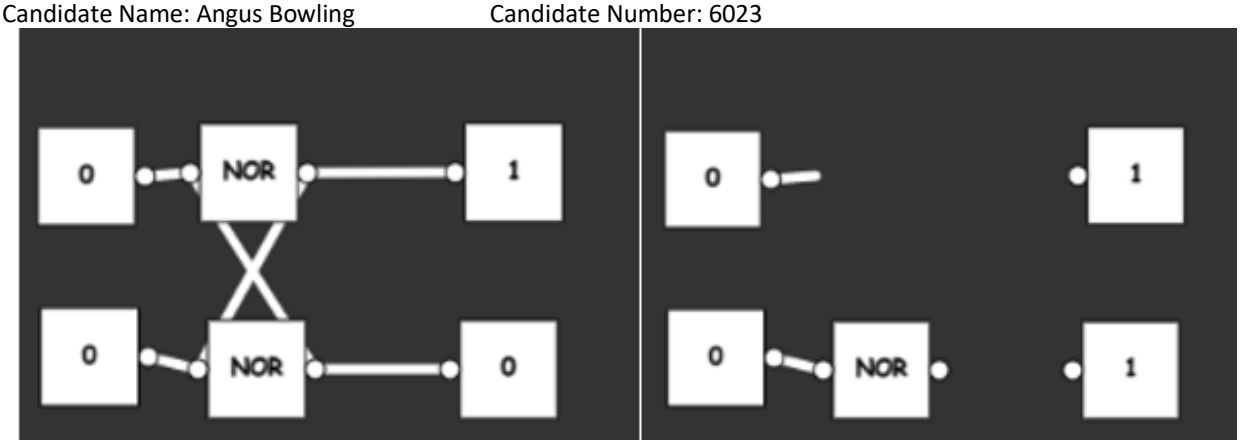
By adding the code at line 29, the value of calcState is set to the current state before any updating. This means that if in the recursive calls of the update state function, if this gate is ever called again, it will return the calcState which is the initial value. This will prevent the function being called recursively indefinitely and stop and max stack errors.



Having added this addition precaution to the update state function, multi-component short circuits are creatable. This is an example of the SR-NOR latch and it both doesn't cause any errors as well as functioning correctly from a logical sense

```
32    //delete a gate
33    delGate(gate){
34      //remove the gate from all its inputs and the correspnding wires
35      gate.inputs.forEach(comp => {
36        comp.outputs.splice(comp.outputs.indexOf(gate),1)
37        this.wires.splice(this.wires.indexOf([comp, gate]),1)
38      })
39      //remove the gate from all its outputs and corresponding wires
40      gate.outputs.forEach(comp => {
41        comp.inputs.splice(comp.inputs.indexOf(gate),1)
42        this.wires.splice(this.wires.indexOf([gate, comp]),1)
43      })
44      //remove gate from the circuit
45      this.gates.splice(this.gates.indexOf(gate),1)
46    }
47  }
48
```

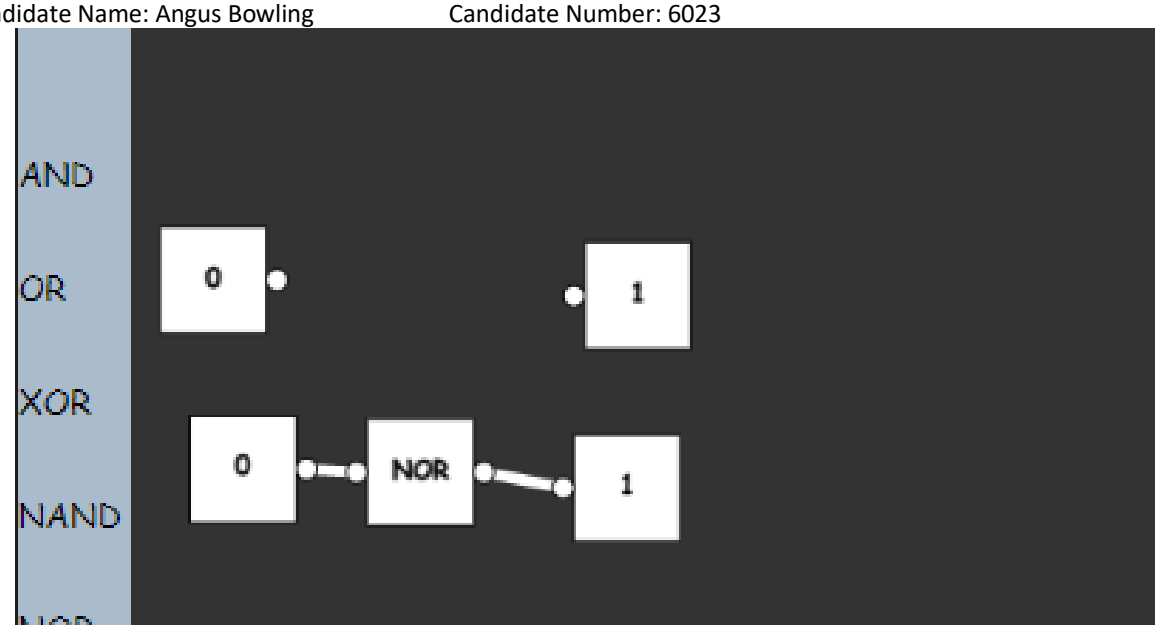
To allow for a fully interactive simulation, the user should be able to delete components. The delete gate function takes a gate as an argument and removes it from the simulation. It first loops through each of the inputs to the gate, and removes itself from the output of each element, as well as removing the wire between them. It then does this for the inputs of each of the gate's outputs. Finally, it removes itself from the circuits gate array



The image on the left shows a user-built circuit that features short circuiting, the image on the right shows the outcome of deleting one of the gates. As show the function isn't correctly working for short circuits, as not all the required wires are gone, some wires are gone that shouldn't be gone, and the state of the outputs is incorrect.

```
32  //delete a gate
33  delGate(gate){
34
35    //remove any wires which include the gate
36    for (var i=0; i<this.wires.length; i++){
37      if (this.wires[i].includes(gate)){
38        this.wires.splice(i,1)
39        i--
40      }
41    }
42
43    //remove the gate from all its inputs and the correspnding wires
44    gate.inputs.forEach(comp => comp.outputs.splice(comp.outputs.indexOf(gate),1))
45
46    //remove the gate from all its outputs and corresponding wires
47    gate.outputs.forEach(comp => comp.inputs.splice(comp.inputs.indexOf(gate),1))
48
49    //remove gate from the circuit
50    this.gates.splice(this.gates.indexOf(gate),1)
```

By removing any wires which contain the gate in its own independent loop, instead of in the other loops, we can avoid the error with the incorrect wire removal. This was caused by removing more than one element per loop which led to the indexing being incorrect for subsequent removals. By decrementing the index after removing a wire, the indexing is restored to the correct value.



Having made these changes, all the correct behavior is occurring except for the outputs staying high when it should be low. This is due to the condition of having a positive number of inputs to calculate the state of an output, and if the number of inputs is zero then the state is set to its current value, however, it should be set to zero.

```
23      //then update this node using the states of its inputs
24      this.state = (this.inputs.length>0) ? funcs.get("OR")(inps) : 0
25    }
```

This change is simply made in line 23 of the update state function of the output class and allows for empty outputs to be set to the correct low state.

```
54  delInput(inp){
55
56    //remove any wires which include the input
57    for (var i=0; i<this.wires.length; i++){
58      if (this.wires[i].includes(inp)){
59        this.wires.splice(i,1)
60        i--
61      }
62    }
63    //remove the input from all its outputs and corresponding wires
64    inp.outputs.forEach(comp => comp.inputs.splice(comp.inputs.indexOf(inp),1))
65
66    //remove gate from the circuit
67    this.inputs.splice(this.inputs.indexOf(inp),1)
68  }
69
70  //delete a gate
71  delOutput(out){
72
73    //remove any wires which include the gate
74    for (var i=0; i<this.wires.length; i++){
75      if (this.wires[i].includes(out)){
76        this.wires.splice(i,1)
77        i--
78      }
79    }
80
81    //remove the gate from all its inputs and the correspnding wires
82    out.inputs.forEach(comp => comp.outputs.splice(comp.outputs.indexOf(out),1))
83
84    //remove output from the circuit
85    this.outputa.splice(this.outputa.indexOf(out),1)
86  }
87 }
```

Similar functions can be made for the input and output classes. The delete input function doesn't need to loop through the inputs as there are none, the equivalent is true for the outputs in the delete output function. There was initially a typo in line 85 where instead of 'outputs', the software attempted to edit an array called 'outputa' which doesn't exist, however I corrected this upon running it.

```
68
69    //a function called on right clicking
70    rightMouse(){
71      if (this.mouseOver()){circuit.delGate(this)}
72    }
```

To allow for the user to delete components, each component class has a right mouse function added to it. This is called when the right mouse button is pressed. The function checks whether the mouse is over the component, and if it is, it calls the delete function on the component.

```
}else if (mouseButton==RIGHT){
  circuit.gates.forEach(gate => gate.rightMouse()) //calls the rightMouse function for all the gates
  circuit.inputs.forEach(input => input.rightMouse()) //calls the rightMouse function for all the inputs
  circuit.outputs.forEach(output => output.rightMouse()) //calls the rightMouse function for all the outputs
}
```

By adding this code to the mouse down event listener, it ensures that when the right button is pressed, the right mouse function will be called on all components, this will allow and components that need to be deleted to do so.

```
<!--the sandbox containing the simulation-->
<div id="canv" oncontextmenu="return false"></div>
```

By setting the oncontextmenu attribute of the div that contains the sandbox to return false, this means that the typical right mouse button context menu will not pop-up. This ensures that the user can easily use the software without a pop-up menu appearing every time they try to delete a component.

REVIEW

The first iteration of development has been completed. This includes the development of software as well as its testing, along with feedback from the stakeholders to ensure that the simulation meets their needs and criteria.

WHAT HAS BEEN DEVELOPED

During the first iteration many basic systems were developed. This lays a groundwork for polish and further development in the subsequent iterations. Having completed the logically functionality of the software, along with the ability to interact with and create as well as destroy circuits, the simulation is in a usable state.

This progress can be seen through the completion of many success criteria set out by the stakeholders in the analysis section

Criteria Met
Allows the user to build their own logic circuits
Correctly calculate logic for NOT, AND, OR, and XOR gates
Allow the user to control the inputs to a circuit
Allow the user to see the outputs of a circuit
Allow the user to edit a circuit while its running
Drag and drop control of the logic gates

As of yet there have been no changes or alterations made to the design other than minor differences in the workings of algorithms, such as removing the need for component ID's as the objects themselves can be passed as pointers. None of these have affected the user's interaction with the simulation so don't require stakeholder validation.

TESTING AND FEEDBACK

During the development in the iteration, tests on each specific procedure were run. Any times that an error, or an unexpected outcome occurred, this was noted and shown. The testing and solving of any errors were also documented throughout the iteration. Testing often involved logging values and outputs to the console to ensure correct functionality. This testing was carried out with the test data defined in the development section where applicable to ensure that the behaviour of the functions was correct.

I also request feedback from the stakeholders who used the software. This feedback was typically less technically oriented but based on the overall feel and ease-of-use of the simulation. They said that they were looking forward to some graphical work as the current stand-in graphics made the simulation unpleasant to use. They also requested that the area for adding wires to and from components would be larger as to make it easier to interact with the simulation. A suggestion that instead of buttons to add components to the sandbox, the user should simply be able to drag across from the sidebar, however, this request was deemed less important that the predefined success criteria so will be held in mind to add on if the time is available.

ITERATION TWO

DEVELOPMENT

During the first iteration, I developed the main functionality of the simulation itself. This allowed for the creation of and the running of logic circuits. During the second iteration I intend to work on the auxiliary menus, such as the data IO and the settings. I also want to work on the graphical aspects, as this will make the using the software far more appealing. I will also try to work on the requests of the stakeholders if I have time during the iteration.

```
import http.server, socketserver #import required libraries

def createHandler(port:int)->type:
    address : tuple[str,int] = ("",port) #set the port for the server to use
    requestHandler : type = http.server.SimpleHTTPRequestHandler #define a request hadler for http requests
    return socketserver.TCPServer(address, requestHandler) #define a TCP server using the address and the http request handler

def runServer(server:type)->None:
    address : int = server.server_address
    print("Serving at: localhost:{}".format(address[1]))
    server.serve_forever() #run the TCP server indefinitly

def main()->None:
    server : type = createHandler(1337)
    runServer(server)

if __name__ == "__main__":
    main()
```

The hosting of the site is done using Python. The script initially imports the required libraries, this being the http.server and the socketserver libraries. A create handler function is defined that takes a port number as an argument. It defines the address which is a tuple containing the port number and an empty string, such that the default host name will be used. It then creates a http request handler which uses the http protocol for communication. A socket server is then defined and returned, this server binds the http handler to the given address.

A run server function is then defined, it takes a server as an argument, it pulls the address from the server to be printed to the user. It then calls the server forever method on the server, this starts up runs the server indefinitely until the program is killed. The main function simply creates a server using the create handler function, and runs it using the run server function.

```
<!--Gates-->
<div id="gates">
  <img class="comp" id="AND" onclick="clickGate('AND')" src="/images/gates/AND.png"></img>
  <img class="comp" id="OR" onclick="clickGate('OR')" src="/images/gates/OR.png"></img>
  <img class="comp" id="XOR" onclick="clickGate('XOR')" src="/images/gates/XOR.png"></img>
  <img class="comp" id="NAND" onclick="clickGate('NAND')" src="/images/gates/NAND.png"></img>
  <img class="comp" id="NOR" onclick="clickGate('NOR')" src="/images/gates/NOR.png"></img>
  <img class="comp" id="XNOR" onclick="clickGate('XNOR')" src="/images/gates/XNOR.png"></img>
  <img class="comp" id="NOT" onclick="clickGate('NOT')" src="/images/gates/NOT.png"></img>
```

To make the simulation more graphically appealing, the gates in the sidebar have been replaced with images. This makes it obvious to see what component is being added to the sandbox

```
78    //display the gate
79    show(){
80      image(gateImg.get(this.type), this.pos.x, this.pos.y, this.w, this.h)
81      fill(255)
82      circle(this.pos.x-R, this.pos.y+this.h/2, 2*R)
83      circle(this.pos.x+this.w+R, this.pos.y+this.h/2, 2*R)
84    }
```

I also refactored the display code for the gates. Instead of displaying a simple box containing the gate type, the correct image is loaded and displayed onto the screen. The circles to add and receive wires have also been given a variable radius, such that it can easily be adjusted to meet the stakeholder's requirements.

```
16  function preload() {
17    //load images for gates
18    gateImg.set("AND", loadImage("/images/gates/AND.png"))
19    gateImg.set("OR", loadImage("/images/gates/OR.png"))
20    gateImg.set("XOR", loadImage("/images/gates/XOR.png"))
21    gateImg.set("NAND", loadImage("/images/gates/NAND.png"))
22    gateImg.set("NOR", loadImage("/images/gates/NOR.png"))
23    gateImg.set("XNOR", loadImage("/images/gates/XNOR.png"))
24    gateImg.set("NOT", loadImage("/images/gates/NOT.png"))
25  }
```

I have also written a preload function, that is run prior to loading the rest of the code. This populates a hash map with the images of the gates, using their type as the key. This allows any gate to access the required image using simply just the gate type.

```
9   //define colours such that user can edit them
10  let highColor = "#0000ff"
11  let lowColor = "#ffffff"
12  let dragColor = "#969696"
13  let bgColor = "#333333"
14  let sbColor = "#aabbcc"
```

I have also defined variables that hold each colour that is displayed, this allows them to be changed throughout the entire simulation, as well as allowing user to set preference for their own style.

```
54      <!--the settings and IO menu-->
55      <div id="IO"></div>
56      <div id="settings"></div>
```

To allow for auxiliary menus, such as the date input/output or the settings menu, divs are created and hidden such that they don't interfere with the main use of the sandbox. These can then have their styles updated by command to allow them to be shown to the user, allowing them to make changes or import/export data.

```
32  #IO, #settings{
33      display: none;
34      width:90vw;
35      height: 90vh;
36      position: absolute;
37      left: 5vw;
38      top: 5vh;
39      z-index: -1;
40      background: #000;
41      border-radius: 25px;
42  }
```

To allow for the divs to be initially hidden, they are set to have the display style of none, and are positioned at z-index of -1, so they are not drawn to the screen. By setting the position style to absolute, this will allow for the menus to appear above the sandbox when required, to center the menus positions are given in terms of the view dimensions.

```
<button id="ioButton" onclick="if(menu==null){openMenu('IO')}else if(menu!='IO'){closeMenu(menu); openMenu('IO')}else{closeMenu('IO')}">IO</button>
<button id="settingsButton" onclick="if(menu==null){openMenu('settings')}else if(menu!='settings'){closeMenu(menu); openMenu('settings')}else{closeMenu('settings')}">settings</button>
<!--The sidebar and canvas-->
```

To allow for the menus to be opened and closed, two buttons are defined, one for each menu. These buttons have an onclick method which calls a menu open function with the specified id. Depending on whether any other menus are open, it may also call the close menu function.

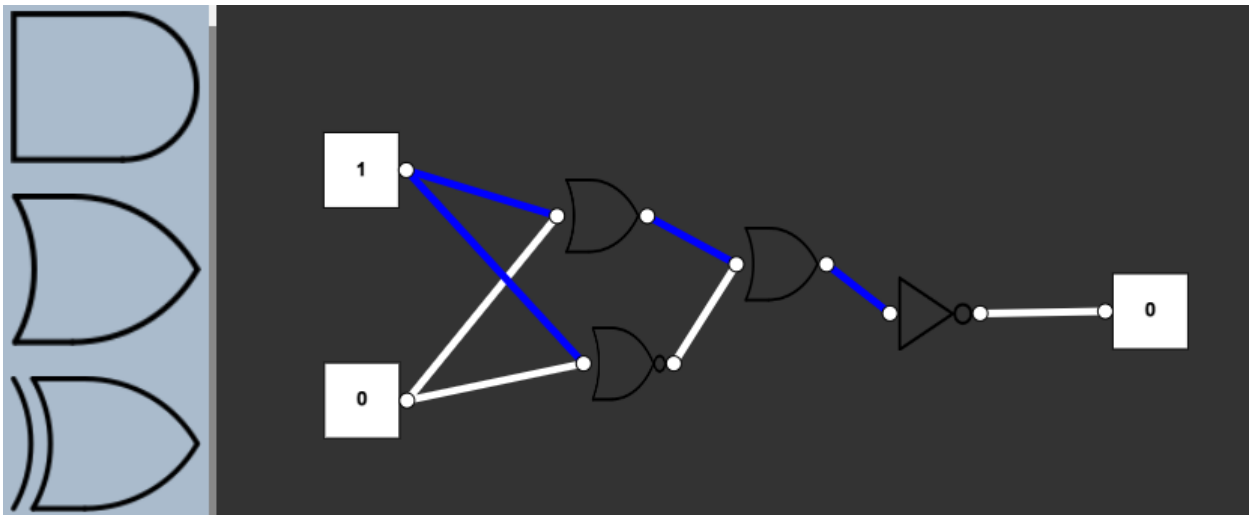
```
143  function openMenu(id){
144    document.getElementById(id).style.zIndex=1
145    document.getElementById(id).style.display="block"
146    menu=id
147    noLoop()
148  }
149
150  function closeMenu(id){
151    document.getElementById(id).style.zIndex=-1
152    document.getElementById(id).style.display="none"
153    menu=null
154    loop()
155  }
```

The open menu function is called when a menu wishes to open. It takes an argument id, which is the id of the menu in the DOM. It then sets the display style of the menu to block and the z-index to 1, this allows the menu to be shown, it then sets the variable menu to the id, this allows the program to know which menu is currently open. The function no loop is then called which pauses execution of the draw function.

The close menu function is very similar to the open menu function. It also takes the id of the menu as an argument; however, it sets the z-index to -1 and the display style to none. This hides the menu from view. It then sets the menu variable to null, symbolising that no menu is currently open. It then calls the loop function, restarting the execution of the draw function.

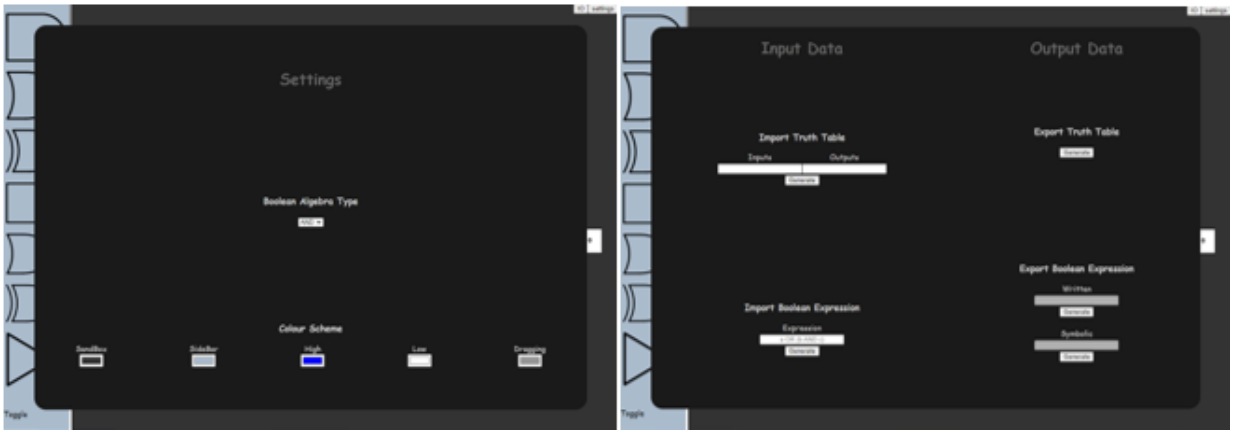
```
/*Custom Scroll Bar*/
::-webkit-scrollbar {
  width: 5px;
}
::-webkit-scrollbar-track {
  background: #f1f1f1;
}

::-webkit-scrollbar-thumb {
  background: #888;
}
::-webkit-scrollbar-thumb:hover {
  background: #555;
}

```

Having set the gates in the sidebar to be images, the sidebar is overflowing in the y direction. This causes the scroll bar to be shown, allowing the user the access to each component. However, the default scroll bar doesn't fit the aesthetic of the simulation, it is clunky and oversized. Due to this I have redefined the scroll bar to be thinner and consist of more muted and complementing colours.



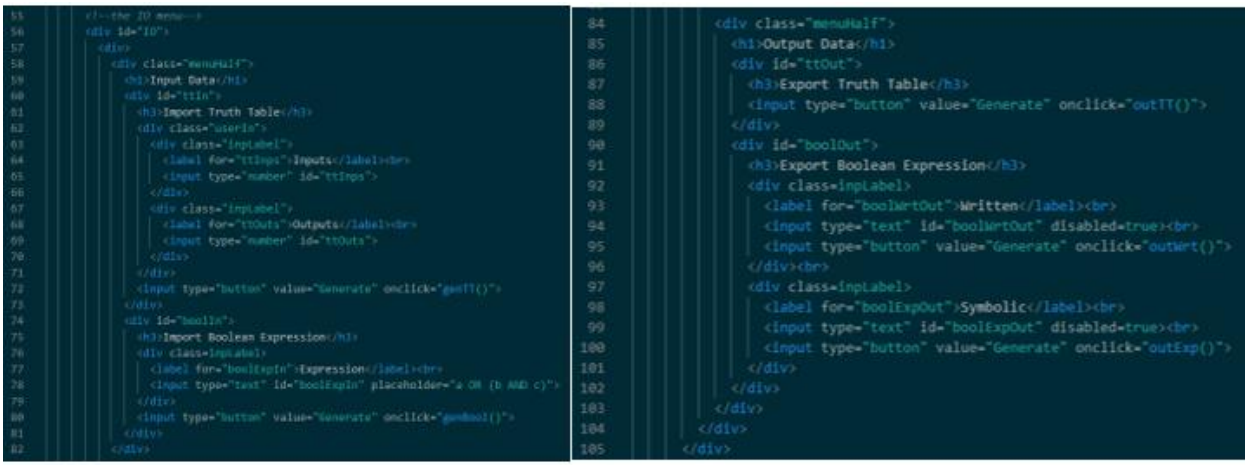
The images used for gates in the sandbox make the simulation more usable as a learning tool as the symbols are equivalent to what you would see in a circuit diagram. The use of the same symbols in the sidebar brings a sense of uniformity to the simulation and the simplified scrollbars allow for use of the sidebar without causing distraction from the simulation. Wires are now also coloured depending on the state of the gate they are coming from, this makes it obvious to see the flow of logic through the circuit.

The IO and settings buttons are also working, causing the menu to be opened or closed on demand. The menus themselves are still currently a work in progress, however the initial size, shape, and positioning of the menus is completed. The buttons themselves are also currently just stand-ins for what will later be icons, however it is important to focus on completing the functionality of my simulation before focusing on graphical aspects as there is a time constraint.



This is the menu pages filled out with their content. The content on the settings page is split out into two distinct parts; the Boolean algebra type, and the colour scheme. The Boolean algebra type is selectable using a drop-down menu, and the colours scheme can be changed using colour selection pallets. The data IO menu is split horizontally into two, the input and output sections. The input section is split, allowing truth tables and expressions. The

output section is also split, allowing for truth table as well as for expressions, which can be both written or using symbols.



```
<!--the IO menu-->
<div id="IO">
  <div>
    <div class="menuHalf">
      <h1>Input Data</h1>
      <div id="ttIn">
        <h3>Import Truth Table</h3>
        <div class="userIn">
          <div class="inpLabel">
            <label for="ttInps">Inputs</label><br>
            <input type="number" id="ttInps">
          </div>
          <div class="inpLabel">
            <label for="ttOuts">Outputs</label><br>
            <input type="number" id="ttOuts">
          </div>
        </div>
        <input type="button" value="Generate" onclick="genTT()">
      </div>
      <div id="boolIn">
        <h3>Import Boolean Expression</h3>
        <div class="inpLabel">
          <label for="boolExpIn">Expression</label><br>
          <input type="text" id="boolExpIn" placeholder="a OR (b AND c)">
        </div>
        <input type="button" value="Generate" onclick="genBool()">
      </div>
    </div>
    <div class="menuHalf">
      <h1>Output Data</h1>
      <div id="ttOut">
        <h3>Export Truth Table</h3>
        <input type="button" value="Generate" onclick="outTT()">
      </div>
      <div id="boolOut">
        <h3>Export Boolean Expression</h3>
        <div class="inpLabel">
          <label for="boolWrtOut">Written</label><br>
          <input type="text" id="boolWrtOut" disabled=true><br>
          <input type="button" value="Generate" onclick="outWrt()">
        </div><br>
        <div class="inpLabel">
          <label for="boolExpOut">Symbolic</label><br>
          <input type="text" id="boolExpOut" disabled=true><br>
          <input type="button" value="Generate" onclick="outExp()">
        </div>
      </div>
    </div>
  </div>
</div>
```

The data IO menu is created using divs, the menu is initially split into two halves for input and output. Each of these halves is then populated with children divs which allow for each induvial input, the inputs are made up of an input box, a label, and a button. This allows for controlled and correct input from the user and ensures that they know what each input is used for.

```
<!--the settings menu-->
<div id="settings">
  <div>
    <h1>Settings</h1>
    <div id="boolType">
      <h3>Boolean Algebra Type</h3>
      <select id="boolGrams"  onchange="setBoolType()">
        <option value="wrt">AND</option>
        <option value="logic">^</option>
        <option value="mech">+</option>
      </select>
    </div>
    <div id="colourScheme">
      <h3>Colour Scheme</h3>
      <div id="colCont">
        <div class="colLabel">
          <label for="sandBoxCol">SandBox</label><br>
          <input type="color" id="sandBoxCol" onchange="setColour('sandBoxCol')">
        </div>
        <div class="colLabel">
          <label for="sideBarCol">SideBar</label><br>
          <input type="color" id="sideBarCol" onchange="setColour('sideBarCol')">
        </div>
        <div class="colLabel">
          <label for="highCol">High</label><br>
          <input type="color" id="highCol" onchange="setColour('highCol')">
        </div>
        <div class="colLabel">
          <label for="lowCol">Low</label><br>
          <input type="color" id="lowCol" onchange="setColour('lowCol')">
        </div>
        <div class="colLabel">
          <label for="dragCol">Dragging</label><br>
          <input type="color" id="dragCol" onchange="setColour('dragCol')">
```

The settings menu is also created in a similar way. Using divs the menu is split into sections for the Boolean algebra type, along with the colour scheme. The type selection is a drop-down menu, this allows users to select from a

distinct and predefined list. Each colour is selected using a colour pallet which gives the user a visual display of what colour they are selecting, ensuring that changing a colour is intuitive and easy to do.

```
33  #IO, #settings{
34    display: none;
35    width:90vw;
36    height: 90vh;
37    position: absolute;
38    left: 5vw;
39    top: 5vh;
40    z-index: -1;
41    background: rgb(26, 26, 26);
42    border-radius: 25px;
43    min-height: 420px;
44    min-width: 560px;
45  }
46
47  #IO > div{
48    display: flex;
49    flex-direction: row;
50    height: 100%;
51    text-align: center;
52  }
53
54  #settings > div{
55    display: flex;
56    flex-direction: column;
57    height: 100%;
58    text-align: center;
59    justify-content: space-around;
60  }
```

To allow for the placing of the menus and the elements in them they are given the flex property. This allows the elements to fill up the available space, and grow or shrink where available. This ensures that the software is usable on a variety of systems. By defining the flex direction of the menus, the location of their children can be controlled. For the IO menu the flex direction is row, as this allows the input and output menus to be next to each other. The settings menu uses flex direction of column as this allows the elements to be spaced vertically through the menu.

```
208  function genBool(){
209    var exp = document.getElementById("boolExpIn").value
210    circuit = new Circuit()
211    inpMap = new Map()
212    circuit.addOutput(random(sideBar.width,width-sideBar.width), random(height), 50, 50)
213    try{
214      var tree = jsep(exp)
215      buildTree(tree, circuit.outputs[0])
216      closeMenu("IO")
217    }catch(error){alert("Invalid Expression")}
218
219  }
```

The gen bool function is called when the user clicks the button to generate a Boolean circuit form an expression. The function initially reads the content of the text box which contains the user's expression. It the initialises the circuit and a hash-map called 'inpMap', which will store a reference to each input. The function then adds an

output to the circuit. Following this, the function attempts to create an abstract syntax tree from the expression. It then calls the build tree function on the tree with the output as an argument. It then closes the IO menu. If an error occurs during this stage, the user is given the alert message “Invalid Expression”

```
221  function buildTree(exp, output){
222    if(exp.type == "UnaryExpression"){
223      circuit.addGate(random(sideBar.width,width-sideBar.width), random(height), 50, 50, exp.operator)
224      circuit.addWire(circuit.gates.at(-1), output)
225      buildTree(exp.argument, circuit.gates.at(-1))
226    }else if(exp.type == "BinaryExpression"){
227      circuit.addGate(random(sideBar.width,width-sideBar.width), random(height), 50, 50, exp.operator)
228      var newGate = circuit.gates.at(-1)
229      circuit.addWire(newGate, output)
230      buildTree(exp.left, newGate)
231      buildTree(exp.right, newGate)
232    }else if(exp.type == "Identifier"){
233      if(!inpMap.has(exp.name)){
234        //new input
235        circuit.addInput(random(sideBar.width,width-sideBar.width), random(height), 50, 50, "toggel")
236        inpMap.set(exp.name, circuit.inputs.at(-1))
237        circuit.addWire(circuit.inputs.at(-1), output)
238        return
239      }else{
240        //existing input
241        circuit.addWire(inpMap.get(exp.name), output)
242        return
243      }
244    }
245  }
```

The build tree function is a recursive algorithm which takes in a tree as well as an output for arguments. It then checks the type of the tree’s root. If the root is a unary expression, such as a NOT gate, then it creates the gate and adds it to the circuit, it then adds a wire between itself and the supplied output, finally it calls build tree on its child tree with itself as the output. If the root’s type is a binary expression, such as an OR gate, then it creates and adds the gate to the circuit. It then stores a reference to the new gate, and adds a wire between itself and the given output. Finally, it calls build tree on both its child trees with itself as the output. Initially I didn’t define a reference to the new gate but simply passed the build tree function the final gate in the gates array, however this was erroneous as a new gate could be created by the build tree function, so when it was called for the second time, the output was the wrong component. If the root type is an identifier, such as an input, then the function checks to see whether the input has already been created, by checking whether it is a key in the ‘inpMap’ hash-map. If it isn’t, so it’s a new input, then the function creates a new input and adds it to the circuit, it also adds the reference to the input into the ‘inpMap’, it then adds a wire between the input and the given output. It then calls return, allowing the recursive routine to unwind. If the input already exists, then it adds a wire between the input and the given output. It then returns allowing the recursion to unwind from the recursive stack.

```
35  //deifne operations for parser
36  jsep.addUnaryOp("NOT")
37  jsep.addBinaryOp("AND",2)
38  jsep.addBinaryOp("NAND",2)
39  jsep.addBinaryOp("OR",1)
40  jsep.addBinaryOp("NOR",1)
41  jsep.addBinaryOp("XOR",1)
42  jsep.addBinaryOp("XNOR",1)
```

The creation of the abstract syntax tree (AST) is completed using the library called JSEP (JavaScript Expression Parser). This library gives use of an expression parser that can be used to convert an expression, such as one supplied by the user, into a tree representation. To allow the parser to convert the Boolean expression into a tree, I define each Boolean function as an operator, giving them the appropriate priority where required. By converting the Boolean expressions into a tree, calculations can be carried out over them, such as creating the circuit with the given expression representation.

```
247  function outTT(){
248    var tt = []
249    for (var i = 0; i<2**circuit.inputs.length; i++){
250
251      var out = []
252      for (var n=0; n<circuit.inputs.length; n++){
253        circuit.inputs[n].calcState = ((i>>n)&1)
254      }
255
256      circuit.gates.forEach(gate => gate.updateState())
257      circuit.outputs.forEach(output => output.updateState())
258
259      circuit.outputs.forEach(output => out.push(output.state))
260
261      circuit.gates.forEach(gate => gate.update())
262      circuit.outputs.forEach(output => output.update())
263
264      tt.push(out)
265    }
```

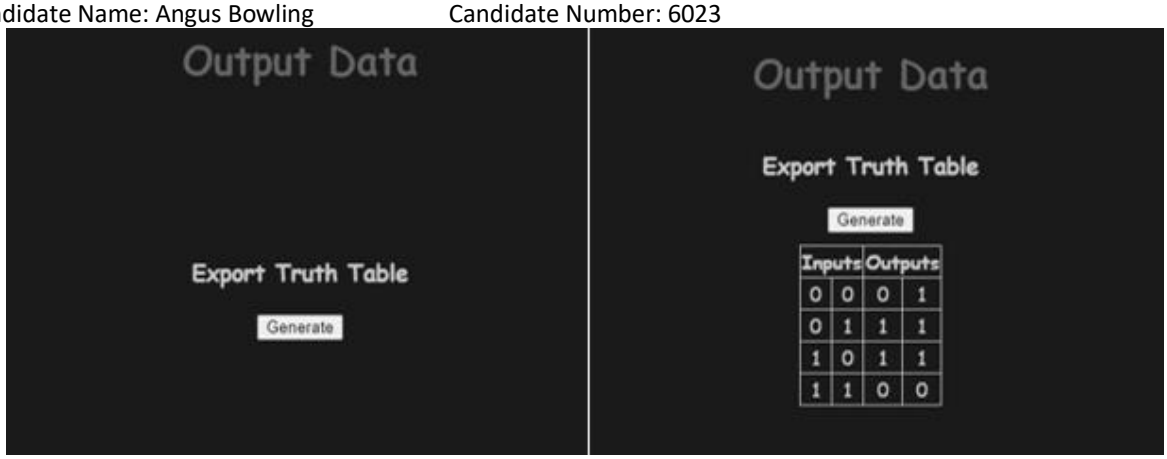
The output truth table function is called when the user presses the output truth table button on the data IO menu. The function initially defines an array called ‘tt’ which will be the representation of the truth table. It then loops between zero and two raised to the power of the number of inputs. This is the number of possible permutations of the input states. For each cycle of this loop, an array called ‘out’ is initialised, then each input is cycled through and its state set to either high or low. The circuit then has the update state functions called on the components, such that the outputs have the correct value. The values of the output are then pushed to the out array, the components are then reset using the update function such that their calcState values are reset to null. The out array is then pushed onto the truth table array. This logically calculates the truth table for the circuit, however, does not yet display this information to the user.

```
247  function outTT(){
248
249    //set the css of the table
250    document.getElementById("outputTruthTable").setAttribute("style", "display:block;")
251    var elementStyle = "overflow-y: auto;"
252    var style = document.getElementById("ttOut").currentStyle || window.getComputedStyle(document.getElementById("ttOut"))
253    elementStyle += " padding: 0 "+style.marginLeft+";"
254    elementStyle += " margin: 0;"
255    document.getElementById("ttOut").style = elementStyle
256    //define referance to the table
257    var tab = document.getElementById("outputTruthTable").firstElementChild
258    for (var i=tab.rows.length-1; i>0; i--){tab.deleteRow(i)}
259    //set the colspan of the input and output headers
260    tab.rows[0].firstElementChild.setAttribute("colspan", circuit.inputs.length)
261    tab.rows[0].lastElementChild.setAttribute("colspan", circuit.outputs.length)
262
```

Having rewritten parts of the output truth table function, having tested its logic in creating truth tables from the circuit, the function now initially set the style properties of the table. This starts by setting the display of the div containing the table to block, allowing it to be displayed on screen. The padding and margins of the div are then set, such that the scroll-bar appears in a more elegant position. A reference to the table is then set, and any existing rows are deleted, resetting the table. The column span of the input and output headers is then set to the number of inputs and outputs. This code all ensure that the table is easily readable by the user, so they can understand the data represented by the truth table

```
    //create a table row
    var row = tab.insertRow(tab.rows.length)
    //add the input values to the row
    for (var n=0; n<circuit.inputs.length; n++){
      row.insertCell(n).innerHTML=((i>>(circuit.inputs.length-n-1))&1)
    }
    //add the output values to the row
    for (var n=0; n<circuit.outputs.length; n++){
      row.insertCell(n+circuit.inputs.length).innerHTML=out[n]
    }
  }
```

The function then runs the same as before. However, instead of appending the output states to 2D array as before, the states are pushed into the table. For each permutation of inputs, the input values are first pushed onto the new row of the table, then the output states are pushed onto the new table row. This allows for the outputs of each gate to be displayed next to the permutation of inputs that they result from, easily show the user the functionality of the circuit they have created.



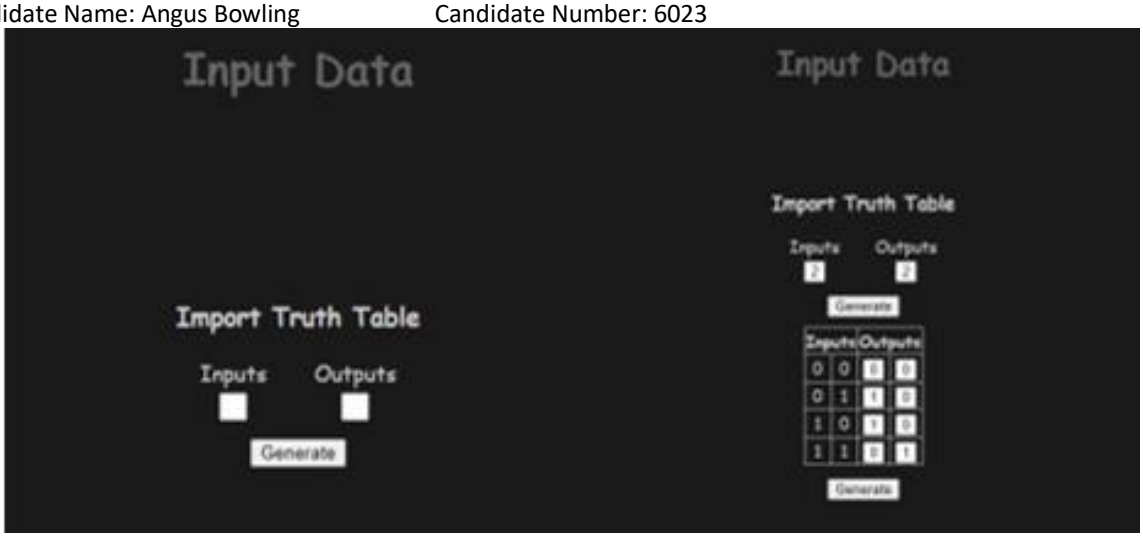
This show the data IO menu before and after pressing the export truth table function button. The user circuit was a XOR gate and a NAND gate, and the correct truth table for this circuit can be seen displayed to the user.

```
125
126   #outputTruthTable > table{
127     margin: 10px auto 0 auto;
128   }
129
130   #inputTruthTable > table{
131     margin: 10px auto 0 auto;
132   }
133
134   table, th, td {
135     border: 1px solid #CCC;
136     border-collapse: collapse;
137   }
```

The CSS for the tables is as shown, this sets a 10 pixels margin on the top of each table as well as setting the left and right margins to auto, while setting the bottom margin to be 0 pixels. By setting the top margin, a buffer between the table and the element above it (a button) is created, spreading out the elements on the menu and making it more readable. By setting the left and right margins to auto, the table will be centered in its container.

```
function genTT(){
  //set the css of the table
  document.getElementById("inputTruthTable").setAttribute("style", "display:block;")
  document.getElementById("ttIn").setAttribute("style", "overflow-y: auto;")
  //define referance to the table
  var tab = document.getElementById("inputTruthTable").firstElementChild
  for (var i=tab.rows.length-1; i>0; i--){tab.deleteRow(i)}

  var inputs = +document.getElementById("ttInps").value
  var outputs = +document.getElementById("ttOuts").value
  //set the colspan of the input and output headers
  tab.rows[0].firstElementChild.setAttribute("colspan", inputs)
  tab.rows[0].lastElementChild.setAttribute("colspan", outputs)

  for (var i = 0; i<2**inputs; i++){
    //create a table row
    var row = tab.insertRow(tab.rows.length)
    //add the input values to the row
    for (var n=0; n<inputs; n++){
      row.insertCell(n).innerHTML = ((i>>(inputs-n-1))&1)
    }
    //add the output values to the row
    for (var n=0; n<outputs; n++){
      var inp = document.createElement("input")
      inp.setAttribute("type", "number")
      inp.setAttribute("max", 1)
      inp.setAttribute("min", 0)
      //restrict values between 0 and 1
      inp.setAttribute("onchange", "this.value = this.value % 2")

      row.insertCell(n+inputs).innerHTML = inp.outerHTML
    }
  }
}
```

The generate truth table function creates a table that allows the user to fill in a truth table. The function starts by setting the display of the table's container to block, allowing it to be displayed. It also sets the overflow-y to auto, allowing a scroll-bar to appear if necessary. It then defines a reference to the table and deletes any existing rows, essentially resetting the table. It then gets the number of input and outputs from the user input boxes, casting the values to integers using the unary plus operator, using these values to set the column span of the input and output headings. It then loops through each permutation of the inputs and pushes those specific input values onto the new row of the table, it then adds the sufficient number of numerical input boxes to the table for the user to define the output values for the given inputs. These input boxes have a function such that if the input isn't either 0 or 1, then it is set to one of those using modular arithmetic.



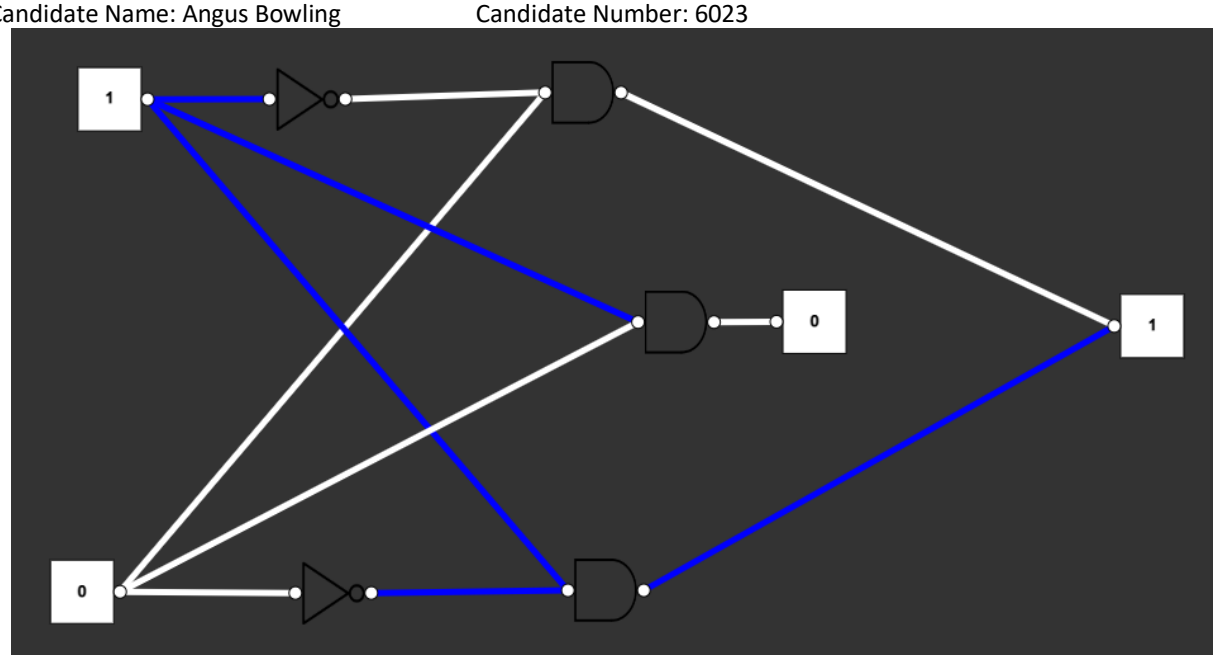
This shows the data IO menu before and after pressing the generate truth table button. The user can input the number of inputs and outputs, in this case 2 and 2. Then the user can fill in the value of each output. Finally, a new button appears allowing the user to build the circuit from the truth table.

```
328  //build the circuit from user truth table
329  function buildTT(){
330   //set referance to table
331    var tab = document.getElementById("inputTruthTable").firstElementChild
332    var inputs = +tab.rows[0].firstElementChild.outerHTML.match(/colspan=\"(\d+)\"/)[1]
333    var outputs = +tab.rows[0].lastElementChild.outerHTML.match(/colspan=\"(\d+)\"/)[1]
334
335    var tt = []
336    //loop through each row
337    for (var i=1; i<tab.rows.length; i++){
338      var out =  []
339      //loop through eeach col
340      for (var j=inputs; j<inputs+outputs; j++){
341        if (tab.rows[i].children[j].firstElementChild.value != ''){
342          out.push(+tab.rows[i].children[j].firstElementChild.value)
343        }else{
344          alert("Please fill in all inputs")
345          return
346        }
347      }
348      tt.push(out)
349    }
350    tableToCir(tt)
351    closeMenu("IO")
352  }
```

Upon pressing the new button, the build truth table function is called. It initially defines a reference to the table as well as storing the number of inputs and outputs. This is done using REGEXs by finding the number after the word colspan in the outerHTML of the input header and the output header. The function then initialise an array that will store the truth table's values. Each output is then looped through for each row of the table, and pushed onto to truth table array. A check is made to see if each output is filled in, and if its not an alert is raised to the user and the function returns. The function then calls the table to circuit function on the truth table and subsequently closes the data IO menu.

```
function tableToCir(tt){
  circuit = new Circuit
  //add inputs and NOT gates
  for (var i=0; i<Math.log2(tt.length); i++){
    circuit.addInput(random(sideBar.width,width-sideBar.width), random(height), 50, 50, "toggle")
    circuit.addGate(random(sideBar.width,width-sideBar.width), random(height), 50, 50, "NOT")
    circuit.addWire(circuit.inputs.at(-1), circuit.gates.at(-1))
  }
  //add outputs
  for (var i=0; i<tt[0].length; i++){
    circuit.addOutput(random(sideBar.width,width-sideBar.width), random(height), 50, 50)
  }
  //loop through truth table
  for (var i=0; i<tt.length; i++){
    for (var j=0; j<tt[i].length; j++){
      if (tt[i][j] == 1){
        //add AND gate if output is high
        circuit.addGate(random(sideBar.width,width-sideBar.width), random(height), 50, 50, "AND")
        var andGate = circuit.gates.at(-1)
        circuit.addWire(andGate, circuit.outputs[j])
        //connect wire between either Input or NOT and AND gate
        for (var n=0; n<Math.log2(tt.length); n++){
          if((i>>(Math.log2(tt.length)-n-1))&1 == 1){
            circuit.addWire(circuit.inputs[n], andGate)
          }else{
            circuit.addWire(circuit.gates[n], andGate)
          }
        }
      }
    }
  }
}
```

The table to circuit function takes a 2D array as an argument. It firstly initialises the circuit object, following this it loops through the number of inputs and creates an input and a corresponding not gate. It adds theses to the circuit and also connect a wire between them. The function then loops through the number of outputs and adds these to the circuit. The function then loops through the truth table, and for each value that is 1 (the output being high for the given input), the function adds an AND gate to the circuit as well as connecting this AND gate to the corresponding output. The function then loops through each input for the given output, and if it is 1 (the input is high) a wire is created between the input and the AND gate, if it is 0 (the input is low) then a wire is created between the corresponding NOT gate and the AND gate.



The circuit produced from the prior user inputted truth table is as follows. This shows the correct function of the algorithm as the two inputs and outputs were created, along with corresponding NOT gates. The function then added AND gates for each time the output is supposed to be high, and connected wires from either the input or the NOT gate to the AND gate dependent on the permutation of the inputs.

```
177   input[type=number]{
178   |  width: 1em;
179   }
180
181   /*hide buttons on  numerical inputs*/
182   /* Chrome, Safari, Edge, Opera */
183   input::-webkit-outer-spin-button,
184   input::-webkit-inner-spin-button {
185   |  -webkit-appearance: none;
186   |  margin: 0;
187   }
188   /* Firefox */
189   input[type=number] {
190   |  -moz-appearance: textfield;
191   }
```

To ensure that the table takes up a reasonable and appealing amount of space on screen, the width of the numerical inputs is set to 1em (the width of one character). However, due to the spin buttons on numerical inputs, these buttons were covering up the input when the user moussed over it, preventing them from focusing on the input box and inputting values. To fix this I disabled the spin buttons, to do this on Firefox based browsers I set the appearance of the numerical input boxes to that of a text field. For other browsers I set the web-kit appearance of the buttons to none, and set the margins to zero, removing their width.

```
417  function cirToTree(comp){
418
419    //Unary operators
420    if (comp.type == "NOT"){
421      return {type:"Unary", comp:comp.type, child:cirToTree(comp.inputs[0])}
422
423    //Binary operators
424    }else if (["AND", "OR", "XOR", "NAND", "NOR", "XNOR"].includes(comp.type)){
425
426      //invalid circuit if gate has less than two inputs
427      if(comp.inputs.length < 2){
428        errCir = true; return
429
430      //for 2 inputs call recursively
431      }else if(comp.inputs.length == 2){
432        return {type:"Binary", comp:comp.type, left:cirToTree(comp.inputs[0]), right:cirToTree(comp.inputs[1])}
433
434      //for >2 inputs, call recursively with first input and a dummy gate with remaining inputs
435      }else{
436        var leftNode = cirToTree(comp.inputs[0])
437        //create dummy gate
438        var compCopy = Object.assign({}, comp)
439        //remove first input
440        compCopy.inputs.shift()
441        return {type:"Binary", comp:comp.type, left:leftNode, right:cirToTree(compCopy)}
442      }
443
444    //inputs
445    }else{
446      return {type:"Input", id:circuit.inputs.indexOf(comp)}
447    }
448  }
```

When I add the functionality to generate the Boolean representation of the user-built circuit, I will need a way of representing the circuit as an object in code. For this purpose, I have developed an auxiliary function, circuit to tree. The function is recursive and take a component as an input; it then checks the type of the component to tell which of three categories the component is part of. If the component is a unary operator, such as a NOT gate, then then an object is returned which has three attributes: a type of unary, the original component, and a child which is the circuit to tree function called with the input to the unary operator. If the type is a binary operator, then a similar process to the unary operator occurs, except the object has a left and a right child; each of which are created using recursive calls to the function. If the number of inputs to a Boolean operator is less than one, a variable errCIr is set to true, indicating that the users circuit is invalid. If the number of inputs is exactly two, then the process runs as normal. However, if it's more than two, the function creates a copy of the gate; it then removes the first element of the components input array, and calls circuit to tree on the components first child and also on the copy of the component. This recursive calling ensures that even gates with multiple inputs, such as an OR gate with three, is decomposed to the gate with inputs of the original first input and then a gate combining the other inputs, such as an OR between the first input and an OR gate of the other two inputs. If the type is not an operator, then it must be an input, in this case a slightly different object is returned. It has a type of input and is given an id which is equal to the index of the input in the circuit's inputs array. No call is made to the function when an input object is received so the recursion can begin to unwind.

```
124  function clickGate(type){
125    var maxInputs
126    if(type=="NOT"){maxInputs = 1}
127    //try 50 times to add the gate
128    for(let i=0; i<50; i++){
129      var accept = true
130      var pos = createVector(random(0,windowWidth-sideBar.width),random(0,height))
131      //check if gate is overlapping another component
132      for(let comp of [...circuit.gates,...circuit.inputs,...circuit.outputs]){
133        if(overlap({pos:pos, w:50, h:50}, comp)){accept=false; break}
134      }
135        //if accepted add the gate
136      if(accept){circuit.addGate(pos.x,pos.y,50,50,type,maxInputs);return}
137    }
138  }
```

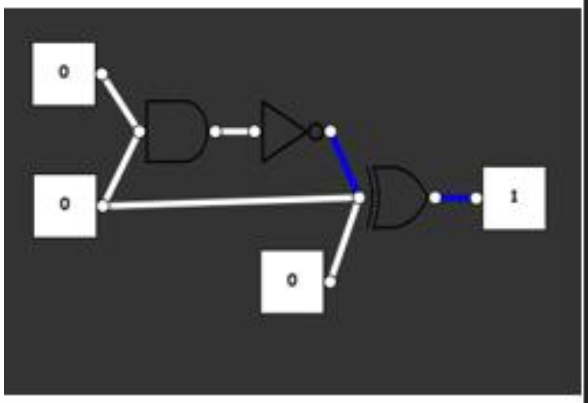
To ensure that no logical errors occur from only looking at a single child of a unary operator, a restriction to a single input must be set on all unary gates, in this case NOT gates. By adding a check to the click gate function, if a NOT gate is created it is passed the max inputs argument of one. This limits the gate to having only a single input and prevents the user from adding a more than one wire to it.

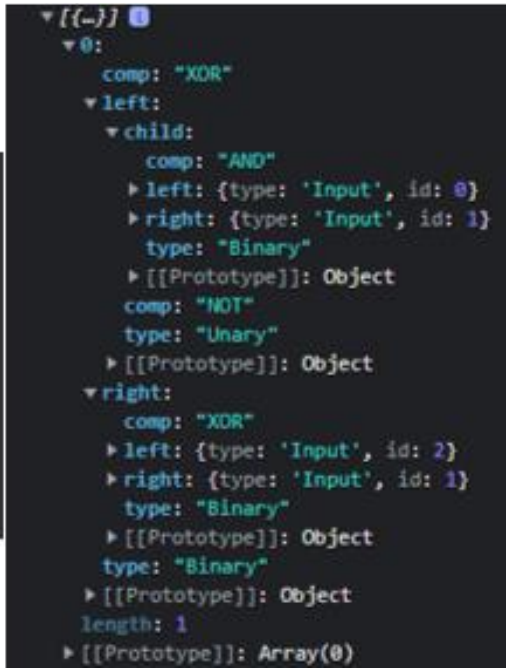
```
10   //add a new gate to the circuit by creating a gate and addding it to the gate list
11   addGate(x, y, w, h, type, maxInputs=10) {
12     this.gates.push(new Gate(x, y, w, h, type, maxInputs));
13   }
14
```

For consistency this change has also been carried across to the add gate function of the circuit class. This is required as the circuit class's wrapper of the gate's constructor function is called by the click gate function, so the max input argument must be accepted by the wrapper function.

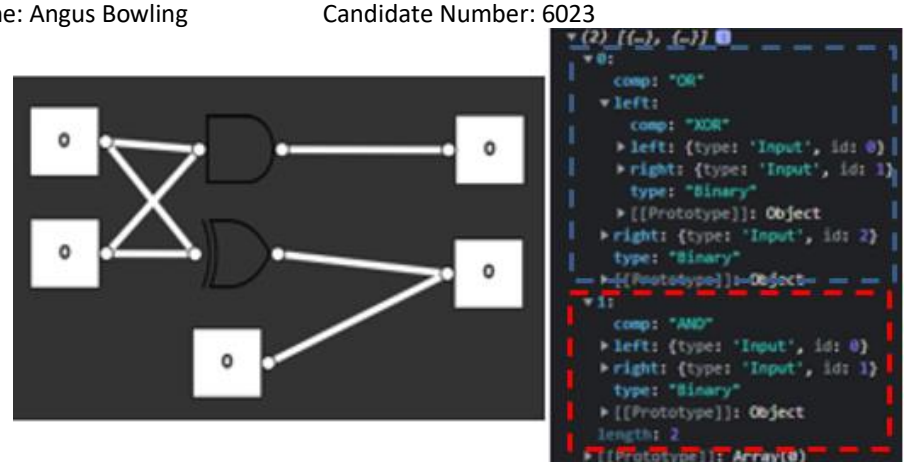
```
390  function outWrt(){
391    //define an array of trees
392    var trees = []
393    //loop through each output
394    for (const out of circuit.outputs){
395      errCir = false
396      //skip outputs with no inputs
397      if(out.inputs.length<1){
398        continue
399      //for outputs with one input
400      }else if(out.inputs.length==1){
401        var tree = cirToTree(out.inputs[0])
402      //for outputs with >1 inputs
403      }else{
404        //create a dummy OR gate as for the output
405        var tree = cirToTree({type:"OR", inputs:out.inputs})
406      }
407      //add tree to array if no errors are present in the circuit
408      if (!errCir){trees.push(tree)}
409    }
410    console.log(trees)
411  }
```

The out written function is called when the user wants to generate a written Boolean representation of their circuit. I first initialises an array called trees, which holds the tree representation of the circuit for each output. It then loops through each output in the circuit, firstly setting the errCir Boolean to false. It then checks how many inputs the output has; if its is less than one, the output is skipped. If it is exactly one, then the circuit to tree function is called on this input and the returned tree is stored. If it is multiple inputs, then the circuit to tree function is called with a dummy object, that has type of OR and the inputs of the output. If no error is found when creating the tree, then the tree is pushed onto the trees array. For testing purposes, the trees array is logged to the console.

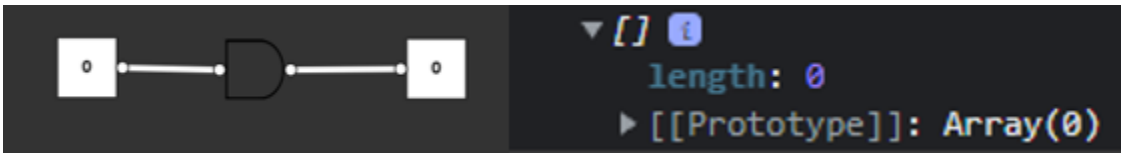




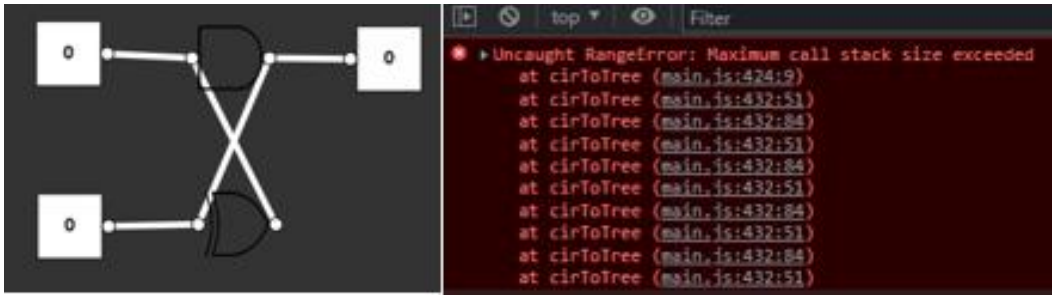
The following circuit, which although small does contain all the paths of the circuit to tree function, gives the correct output. This features a binary gate with more than two input, which you can see the XOR gate's inputs are broken down into the NOT gate and a new XOR gate containing the other two inputs. It contains a unary operator that is correctly distinguished, and the different object format can be seen in the output where the NOT gate has only one child. The circuit also features the same input being used in multiple places, and you can see that the gate with ID two occurs twice in the output as expected in the correct locations.



For this circuit, the use of multiple outputs is tested, this can be seen working with the two different trees being highlighted in blue and red in the output. It also shows the creation of an or gate in the blue output, as the output has more than one input. Consistency between the input ID's can also be seen across the two outputs so the expressions for multiple outputs will be logically input-consistent.



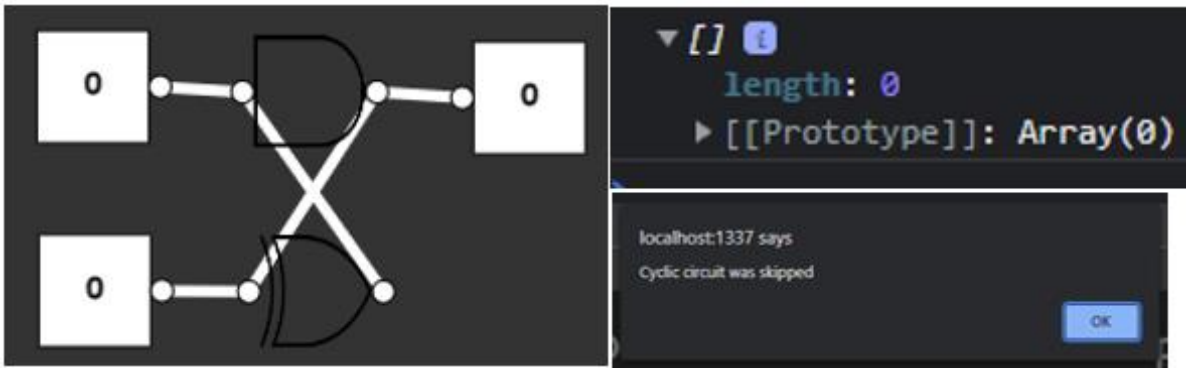
The following circuit is invalid as the AND gate only has a single input, however, as it is a Boolean operator it requires at least two. The expected output for this would be an empty array as when the output written function calls the circuit to tree function, it checks the value of the errCir variable; and as the AND gate only has one input this will be set to true, and the tree won't be pushed to the array.



The following circuit has a closed loop in it, this means the circuit cannot be displayed as a tree but only as a graph; due to this no finite Boolean expression can represent the value of the output. The expected output of this would be an empty array as it's an invalid circuit, however, an error was raised due to a recursive stack overflow; this means that the function is being recursively called indefinitely with no returns being met. The reason for this is the loop causes the function to be called on a component that has been met higher in the recursive flow. A component is eventually being reached from itself, so the cycle continues and causes an error.

```
394     //Loop through each output
395     for (const out of circuit.outputs){
396       try{
397         errCir = false
398         //skip outputs with no inputs
399         if(out.inputs.length<1){
400           continue
401         //for outputs with one input
402         }else if(out.inputs.length==1){
403           var tree = cirToTree(out.inputs[0])
404         //for outputs with >1 inputs
405         }else{
406           //create a dummy OR gate as for the output
407           var tree = cirToTree({type:"OR", inputs:out.inputs})
408         }
409       }catch(RangeError){
410         errCir = true
411         alert("Cyclic circuit was skipped")
412       }
413       //add tree to array if no errors are present in the circuit
414       if (!errCir){trees.push(tree)}
415     }
```

Having added a try-catch statement between lines 396-409 and 409-412, any range errors raised due to maximum recursive depth being reach will be caught; and instead of trying to create a tree of the circuit, an alert will be made to the user informing them that the circuit was skipped, and no tree will be pushed to the trees array.



Having added the new checks, when running the output written with the given circuit no errors occur. The output is an empty array as expected and a message is sent to the user informing them that the cyclic circuit was skipped. By ensuring that no user-built circuits can cause an error to occur, even if they are not possible to represent as an expression, means that even new and unexperienced users will feel comfortable using all functionalities of the software; and by informing them of why no representation was show gives them a learning opportunity.

REVIEW

The second iteration of the software has been completed. This was primarily work on the menu systems as well as their functionality. This development included the testing of functions by myself as well as the stakeholders to ensure that they were happy with the software.

WHAT HAS BEEN DEVELOPED

The second iteration focused heavily on the creation, population, and functionality of the auxiliary menus. This includes both the data IO menu and the settings menu. These both needed to be constructed in HTML and CSS, allowing them to be hidden or shown to the user, as well as the placement of the elements inside these menus along with their graphical design. As per the stakeholder's request, I worked heavily on improving the graphics of the simulation, as this makes the software more enjoyable to use. This included adding images for the gates in both the sidebar as well as the sandbox, adding colour representation of the high and low states of wires. I also gave the menus an appealing beveled design along with spacing out the content within them. Effort was made to ensure that the input boxes were correctly sized for their needs, restricting the numerical input boxes to slim sizes, and removing the side buttons such that the truth-tables were easy to fill in. More minimalistic scroll bars were introduced and the option for certain sections of menus to be scrollable was added to ensure that even when specific areas of the menus are very large, the rest of the menu remains readable and usable.

In terms of functional developments, the auxiliary menu features were also developed. In the settings menu the ability to set specific parts of the colour scheme to the user's choice was developed, allowing a more personalised feel for the simulation. The option to change the Boolean algebra type is also available; this ensures that the user can read their expressions using whichever symbols they feel more comfortable with or challenge themselves to learn a new symbol set. In the data IO menu, both inputting and outputting data have been developed. For the data input, the ability to create a circuit from a Boolean expression was implemented; this gives the user options to create circuits from any expression they read and be able to test its functionality. It also allows users to share circuits they have built between themselves allowing for a community feel around the simulation. The creation of circuits from a truth table is also possible. By defining the desired number of inputs and outputs, the user can then fill in a truth table and generate the corresponding circuit. This gives users the ability to create circuits with known functionalities even if they can't think of a Boolean representation of it. In terms of data output, the ability to generate a truth table for the user-built circuit has been developed. This allows for the complete functionality of multiple circuits to be quickly and easily seen and compared without having to manually run through each permutation of inputs. The logical backbone for outputting a user-built circuit as an expression has also been put into place with the ability to convert the circuit into an abstract tree.

Addition to the usability of the sandbox were also developed. This includes the stakeholders request of differently sized areas for adding wires, this has been set to a global variable so it could be added to the settings menu if desired. There were also checks put into place for the number of inputs that a unary operator can have, this ensures that users will only observe the correct and expected behaviour of gates such as the NOT.

Criteria met
Allow for the use of multiple Boolean algebra symbol sets, and the ability to switch between them
Allow the user to generate a truth table of a circuit
Allow the user to generate a circuit from a Boolean expression
Allow the user to generate a circuit from a truth table
Uncluttered and readable interfaces

There have still not of yet been any major diversions from the solution set out in the design section. Minor variations in programming technique were made, such as using a parser to generate a tree from a Boolean expression to generate the circuit from, or simply evaluating the value of the circuit for all permutations of inputs instead of creating an abstract representation of the circuit and running the inputs over that for generating a truth table as output. Due to the nature of these not affecting the interaction with the software itself, the differences can be abstracted away and don't require stakeholder approval.

TESTING AND FEEDBACK

Similar to the first iteration, during active development, all new functions are tested as they are written. This ensure that, not only do the functions logically run, but they also integrate with all the previous functions. If any errors, or unexpected outcomes, occurred during development they were documented along with screenshots of the behaviour where applicable. The development and implementation of solutions was then also documented and shown, typically along with an explanation for the original error.

As this iteration brought in lots of new features, along with corresponding interfaces, much feedback from the stakeholders was received. The feedback on the aesthetics was positive; focusing on the new menus, they said that they liked the sectioning of similar features into similar areas as this intuitive use of related features. Notes were made about the improved look of the tables after showing them images of before the width of input boxes was changed, they preferred the new design as it "flowed better" and made the "related inputs and outputs more obvious". Richard enjoyed the ability to create circuits from Boolean expressions as he found it quicker to make circuits than building up gate by gate; he suggested the option to have more than one output be added at a time, especially with the sharing of the same inputs between them.

ITERATION THREE

DEVELOPMENT

During this third iteration work will focus on completing the remaining success criteria along with polishing the software; making it easier and more intuitive to use, as well as nicer graphically. The main features to be developed this iteration is displaying a Boolean expression of a user-built circuit, and clocks. Finishing the Boolean expression shouldn't be too hard as the majority of the logic for this process was developed last iteration, all that remains is displaying it to the user. The addition of clocks to the software may be more challenging. The clocks will probably either be a type of input or will extend the input class, as the majority of the code for using clocks will already be present. The code for updating the state of the clock will need to be developed, as well as integrating the clock objects into all of the functions in the auxiliary menus. I would also like to add the option to change the speed of specific clocks; this may be done from the settings menu or by right clicking the clock.

```
431     var exp = ""
432     for(var tree of trees){exp += ", "+inputNames[trees.indexOf(tree)].toUpperCase()+"="+inOrder(tree)}
433     document.getElementById("boolWrtOut").setAttribute("value", exp.slice(2))
434   }
```

I have added the following three lines to the bottom of the output written function. The first line (431) initialises an empty string called exp. The second line is a for-loop, it loops through each expression in the trees array, and manipulates the exp string. It firstly appends a comma and whitespace to it, then it adds the input name with the

index of the current loop and converts this to upper case, finally it adds an equal sign and the result of a call to the in-order function with the current tree as the argument. It then sets the value of the output box to the exp variable, with the leading comma and whitespace removed

```
51    //define a string for the name of inputs
52    fetch("/data/inputNames.txt").then(res => res.text()).then(txt => inputNames=txt.split("\r\n"))
53
```

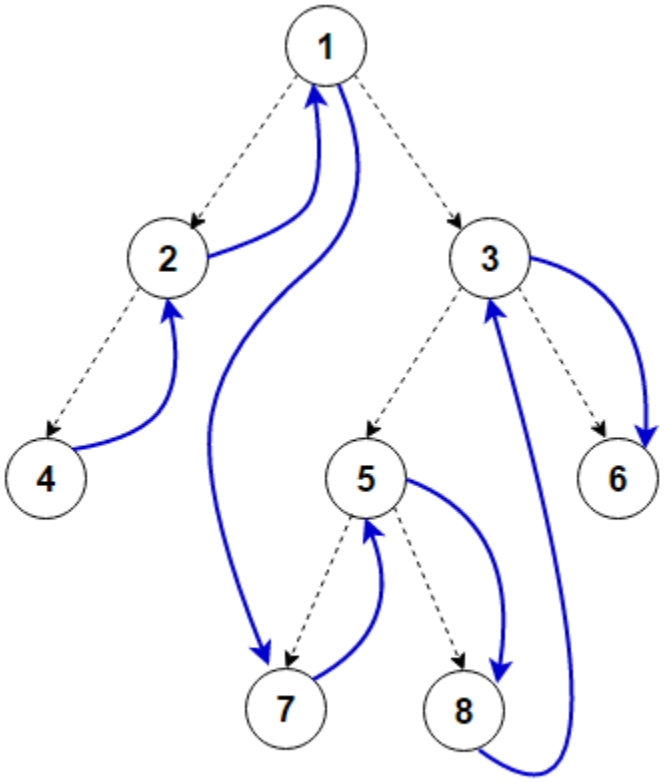
The input names array is defined in the preload function. A fetch request is made to a local file (/data/inputNames.txt), and the response from this is converted to text. This text form is then split into an array using the newline and carriage return characters, and the result is set to the input names function. This is called in the preload function as fetch is an asynchronous function. This leads to it returning a promise object, so this is manipulated using the 'then' prototype. This allows the rest of the code to run sequentially while the processing happens in the background, preventing the code from waiting for the response to the request.

```
1  from string import ascii_lowercase as letters #import a string of lowercase letters
2  with open("./data/inputNames.txt", "w") as file: #use files context manager
3    for num in range(0,10): #loop over integers from 0 to 9
4      for letter in letters: #loop over each letter in alphabet
5        file.write(letter+str(num)+"\n" if num>0 else letter+"\n") #write the concat of letter and int
6
```

The following python code generates the input names file. The first line imports a string containing all lower-case ascii characters and names it letters. The second line then uses the context manager for files, it then invokes the __enter__ function of the file context manager. The next two lines (3 and 4) are nested for loops which loop through each combination of digit and letter. Line five then calls the write function on the file, if the digit is zero, then the letter and a newline character are written, however, if the digit is not zero, the letter concatenated with the digit is written along with a newline character. After this the files context manager calls the __exit__ function on the file, saving it and closing it.

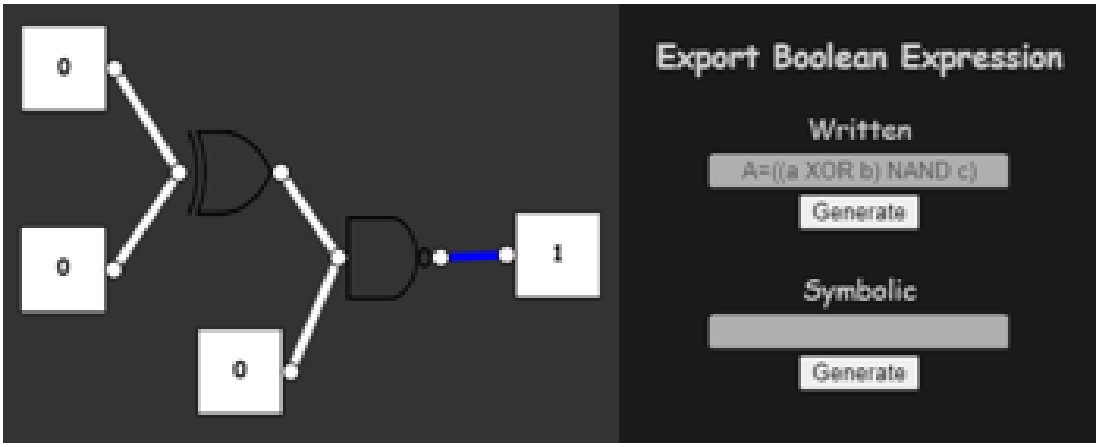
```
504  function inOrder(node){
505    if(node.type=="Input"){
506      return inputNames[node.id]
507    }else if(node.type=="Unary"){
508      return "("+node.comp+" "+inOrder(node.child)+")"
509    }else{
510      return "("+inOrder(node.left)+" "+node.comp+" "+inOrder(node.right)+")"
511    }
512  }
```

The in-order function is a simple in-order tree traversal. It takes in a node as an argument, if the node is and input, and therefore a leaf node, it returns the element at the node id's index in input names. It doesn't call itself, so recursion ceases at leaf nodes. If the node is a unary node, it returns the concatenation of its component (such as OR) and a call to the in-order function on its child node. This is also enclosed within brackets to make order obvious. The function call comes after the node's components because, in Boolean expression, the NOT operator comes before its operand (e.g. NOT a). if the type is a binary operator, then the function returns the concatenation of the in-order function called on the left child, followed by the node's component, followed by the in-order function called on the right child. This is also enclosed in brackets.



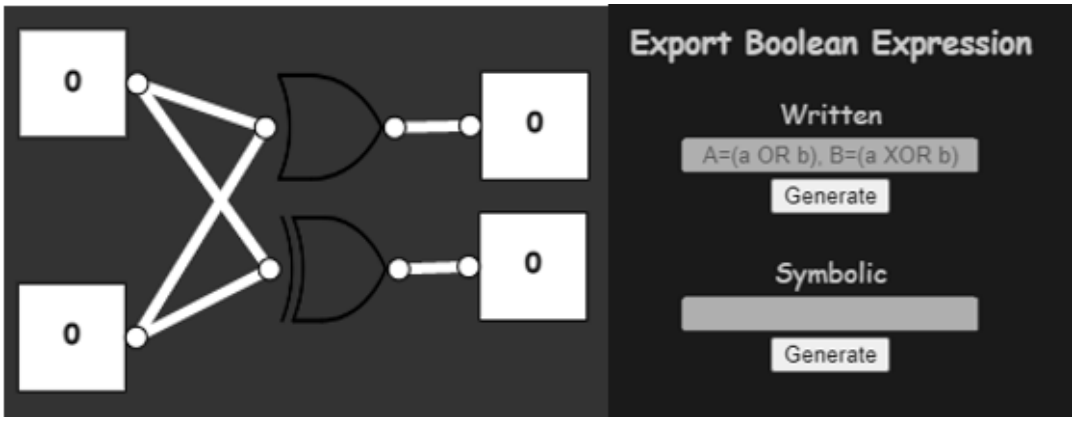
Inorder: 4, 2, 1, 7, 5, 8, 3, 6

The following image show how in-order tree traversal works. It visits the left child, then the node itself, then the right child. It works recursively and is called initially on the root node. It continually checks the left child until it reaches a node with no children; at this point it executes the operation on the node, in our case adding its type to a string. Unwinding then begins, the operation is executed on the parent node, then execution occurs on the right child. As it works recursively, the operation on a node and its two children may just be the result of a call on an even higher node, with the subtree being simply one of its children.



The following circuit then produces the correct Boolean expression when the user clicks the generate function. The brackets make the order of operations clear without having to use priority of operators. Inputs are assigned unique lowercase letters, from the input names array, if more than 26 inputs are required then digits are appended (e.g.

c3). Outputs are assigned uppercase letters, this is generated from setting a value from the input names array to upper case; again, if more than 26 outputs are required, digits are appended (e.g. D5)



For the following input, two outputs are present. This can be seen in the comma separated expression in the output. The outputs also both use the same inputs, and the inputs have the same names in both expressions of the output. This is the behaviour that we would expect to see.

```
54      //fill in wrtToExp
55      wrtToExp.set("AND", "[·, ∧]")
56      wrtToExp.set("OR", "[+, ∨]")
57      wrtToExp.set("XOR", "[⊕, ⊻]")
58      wrtToExp.set("NOT", "[', ¬]")
59  }
```

To allow for outputting the user-built circuit as a Boolean expression using symbols, I have created a hash-map which has key of the gate type, and values of the possible symbols. The value are arrays as the user can chose between symbol sets, the current symbol set selected will be the index of the array that is used.

```
514  function pureBool(node){
515    if(node.type=="Input"){return node}
516
517    if(node.type=="Binary"){
518      node.left = pureBool(node.left)
519      node.right = pureBool(node.right)
520
521      if(["NAND", "NOR", "XNOR"].includes(node.comp)){
522        var nodeCopy = Object.assign({}, node)
523        nodeCopy.comp = nodeCopy.comp.replace(/N/, '')
524        node.type = "Unary"
525        node.comp = "NOT"
526        node.child = nodeCopy
527      }
528      return node
529    }
530
531    if(node.type=="Unary"){
532      node.child = pureBool(node.child)
533      return node
534    }
535  }
536
```

Given that there are not symbols for the negated operators (NAND, NOR, XNOR), for the expression to be output using symbols, the tree must be altered to only contain the main operators. The pure bool function is recursive and takes in a node as an argument. If the type of the node is and input, then the node itself is returned, no recursive call is made as this is the base case. If the type is unary, then the function is called on the nodes child and then the node is return. If the node type is binary, then the function is recursively called on both the left and right child nodes. A check is then made to see if the node is a negated gate. If it is, then a copy of the node is made and this copy has its gate changed to the non-negated version (e.g. NOR -> OR) using a regular expression to remove the first N in the string. The new node is not simply a reference to the original node, but a shallow copy made using the object's assign primitive, therefore changes to the node or node-copy will not affect each other. The type of the original node is then set to unary; its component is set to a NOT gate, and its child is set to the altered copy of itself. This process changes a negated gate to a NOT gate, containing the non-negated gate as its input.

For example:

a XNOR b -> NOT (a XOR b)

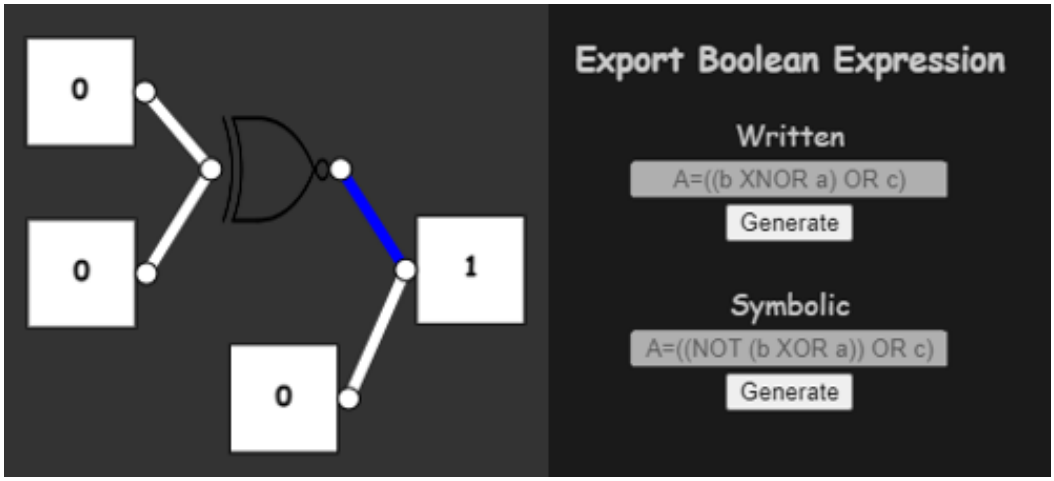
Or:

a NAND (b NOR c) -> NOT [a AND (NOT {b OR c})]

The first example simply replaces the XNOR gate with and XOR gate of the two inputs, and then encloses that in a NOT. The second example shows how this works recursively. Not only replacing the NAND, but also the embedded NOR gate into the component NOT and AND, and NOT and OR gates.

```
436  function outExp(){
437    //define an array of trees
438    var trees = []
439    //loop through each output
440    for (const out of circuit.outputs){
441      try{
442        errCir = false
443        //skip outputs with no inputs
444        if(out.inputs.length<1){
445          continue
446        //for outputs with one input
447        }else if(out.inputs.length==1){
448          var tree = cirToTree(out.inputs[0])
449        //for outputs with >1 inputs
450        }else{
451          //create a dummy OR gate as for the output
452          var tree = cirToTree({type:"OR", inputs:out.inputs})
453        }
454      }catch(RangeError){
455        errCir = true
456        alert("Cyclic circuit was skipped")
457      }
458      //add tree to array if no errors are present in the circuit
459      if (!errCir){trees.push(tree)}
460    }
461
462    var exp = ""
463    for(var tree of trees){
464      console.log(pureBool(tree))
465      exp += ", "+inputNames[trees.indexOf(tree)].toUpperCase()+"="+inOrder(pureBool(tree))
466    }
467    document.getElementById("boolExpOut").setAttribute("value", exp.slice(2))
468  }
```

The output expression function is called when the user attempts to output a symbolic Boolean expression of their user-built circuit. It currently is the same as the output written function except for two points. Instead of calling the in-order function on the tree during the for-loop, it first calls pure bool on the tree to remove any negated gates. It also outputs to the symbolic output box. The output is still in written form; however, this will allow me to ensure that the pure bool function is working as intended.



In this example you can see a circuit containing a XNOR gate, along with the outputs of the written and symbolic expressions. The written expression uses the default generated tree, which contains a XNOR component, however, the symbolic output uses the tree returned by pure bool. You can clearly see how the XNOR gate has been

replaced by and XOR gate contained as input to a NOT gate, both of which have symbol representations that can be used.

```
462     var exp = ""
463     for(var tree of trees){
464       exp += ", "+inputNames[trees.indexOf(tree)].toUpperCase()+"="
465       if (boolType == "wrt"){exp+=inOrder(pureBool(tree))}
466       else{
467         exp+=inOrder_symbolic(pureBool(tree))
468         i = (boolType=="mech") ? 0:1
469         exp = exp.replace(gateRX, x=>wrtToExp.get(x)[i])
470       }
471     }
472     document.getElementById("boolExpOut").setAttribute("value", exp.slice(2))
```

The following alterations were made to the output expression function. When appending to exp, a check is made to see which symbol set is currently selected. If written is selected then the output is calculated the same as for output written, however, pure bool has been called so the output won't necessarily be the same tree. If either mech or logic are selected, then the in-order symbolic function is called. The output from this function then has the replace prototype called on it. It takes in a predefined regex and replaces any component with the symbol from a hash map.

```
20    //define a regex of gate types
21    let gateRX = /AND|OR|XOR|NAND|NOR|XNOR|NOT/g
```

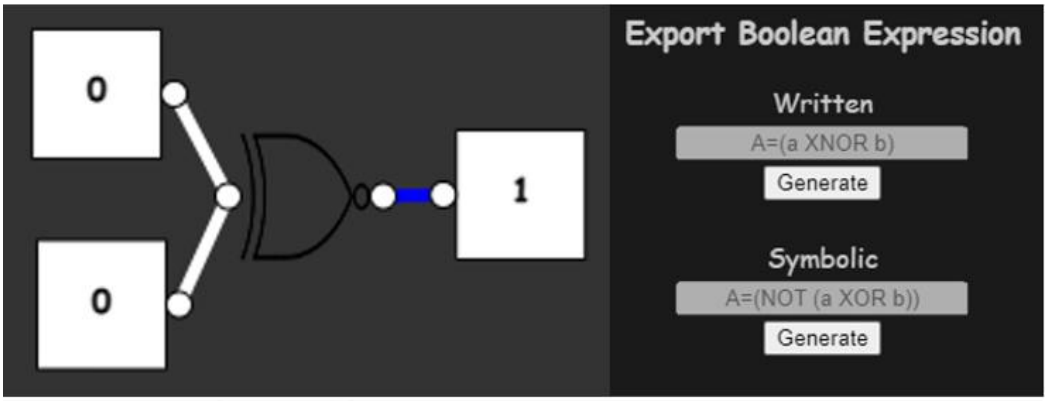
The regex is as follows, it checks for the string representation of and of the gates, the '|' (OR) control symbol means that any of the gates can be present, and the closing 'g' means that the string is checked globally, so more than one match may occur.

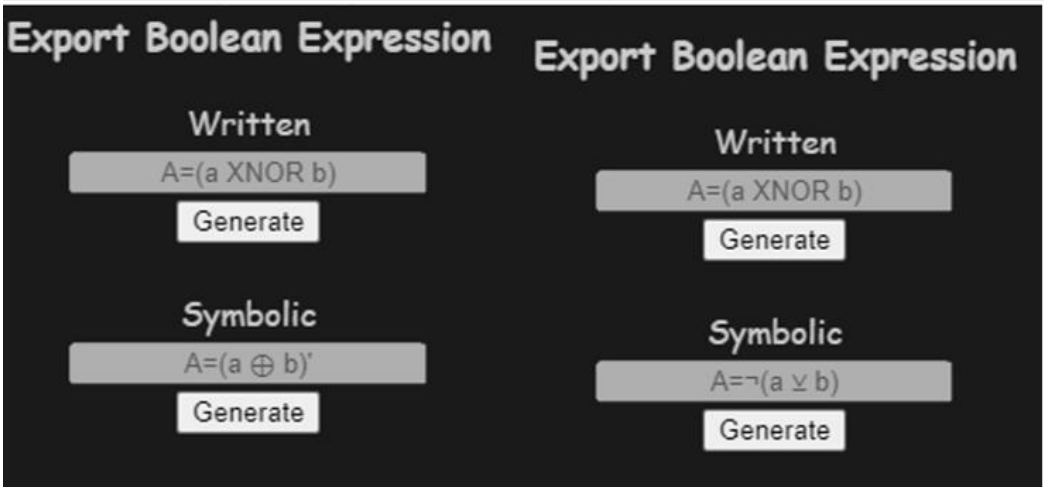
```
54      //fill in wrtToExp
55      wrtToExp.set("AND", ["·", "∧"])
56      wrtToExp.set("OR", ["+", "∨"])
57      wrtToExp.set("XOR", ["⊕", "⊻"])
58      wrtToExp.set("NOT", ["'", "¬"])
59    }
```

A syntaxial error was made when defining the symbol look up hash-map, instead of the values being an array of characters, it was a string of an array. This error has been resolved.

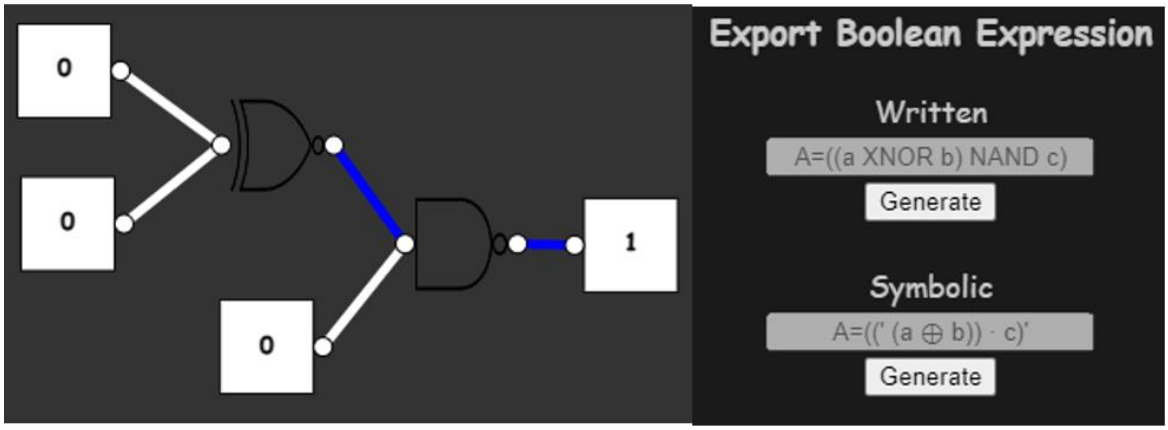
```
541  function inOrder_symbolic(node){
542    //get index of symbol in array from hashmap
543    var index = (boolType == "mech") ? 0:1
544
545    if(node.type=="Input"){
546      return inputNames[node.id]
547    }else if(node.type=="Unary"){
548      if(index==0){
549        //mechanics notation a'
550        return inOrder(node.child)+node.comp
551      }else{
552        //logical notation ¬a
553        return node.comp+inOrder(node.child)
554      }
555    }else{
556      return "("+inOrder(node.left)+node.comp+inOrder(node.right)+")"
557    }
558  }
```

The in-order symbolic function works the same as the standard in-order function. The exception is when using unary operators. The function checks whether the Boolean type is mechanical or logical, and depending on this either places the NOT symbol before or after the child node. This is to match the notation used by the symbol set.





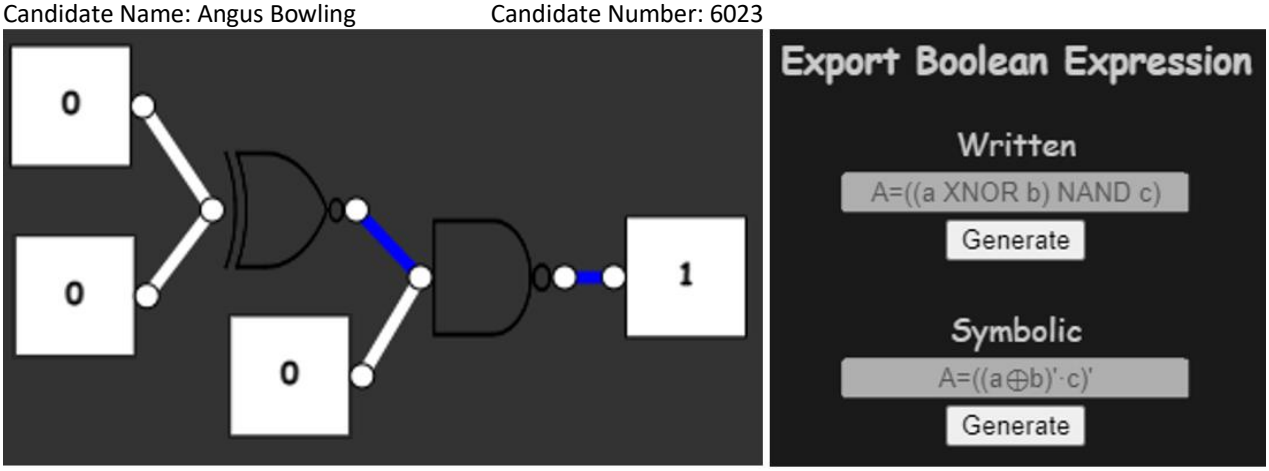
The above example shows a simple user-built circuit of a XNOR b. the three example outputs show the symbol sets of written, mechanical, and logical. While all have the same written output, you can see how each one has a unique symbolic output, with the correct symbols and symbol placement as expected.



The following example shows a user-built circuit with nested negated gates. The written output is correctly calculated, however, when the symbol set is either logical or mechanical (as shown), the output is incorrect. Although the first negation is correctly calculated, the subsequent on places the NOT in the incorrect place and adds whitespace between the elements. This is not the expected behaviour.

```
541  function inOrder_symbolic(node){
542    //get index of symbol in array from hashmap
543    var index = (boolType == "mech") ? 0:1
544
545    if(node.type=="Input"){
546      return inputNames[node.id]
547    }else if(node.type=="Unary"){
548      if(index==0){
549        //mechanics notation a'
550        return inOrder_symbolic(node.child)+node.comp
551      }else{
552        //logical notation ¬a
553        return node.comp+inOrder(node.child)
554      }
555    }else{
556      return "("+inOrder_symbolic(node.left)+node.comp+inOrder_symbolic(node.right)+")"
557    }
558  }
```

The error occurred due to incorrect calling from the in-order symbolic function. Although the initial call to in-order symbolic was made by the output expression function, the subsequent recursive calls by in-order symbolic were simply being made to in-order, this means that any negated gates past the root node would be treated as written output would treat them, and the whitespace that the written elements have between them was added to any subsequent elements. This has now been fixed by ensuring that in-order symbolic calls itself.



Now, given the above user-built circuit, the output show is created. This is the correct behaviour as the NOTs are placed in the correct locations and the whitespace is omitted.

```
247  function genBool(){
248    //get the user input expression
249    var exp = document.getElementById("boolExpIn").value
250    //exp = exp.replace(/\((.+)\)'/)
251    exp = exp.replace(symRX, x=>expToWrt.get(x)) //replace symbols with written
252    circuit = new Circuit()
253    inpMap = new Map()
254    circuit.addOutput(random(sideBar.width+50,width-sideBar.width-50), random(height-50), 50, 50)
255    try{
256      var tree = jsep(exp)
257      buildTree(tree, circuit.outputs[0])
258      closeMenu("IO")
259    }catch(error){alert("Invalid Expression")}
```

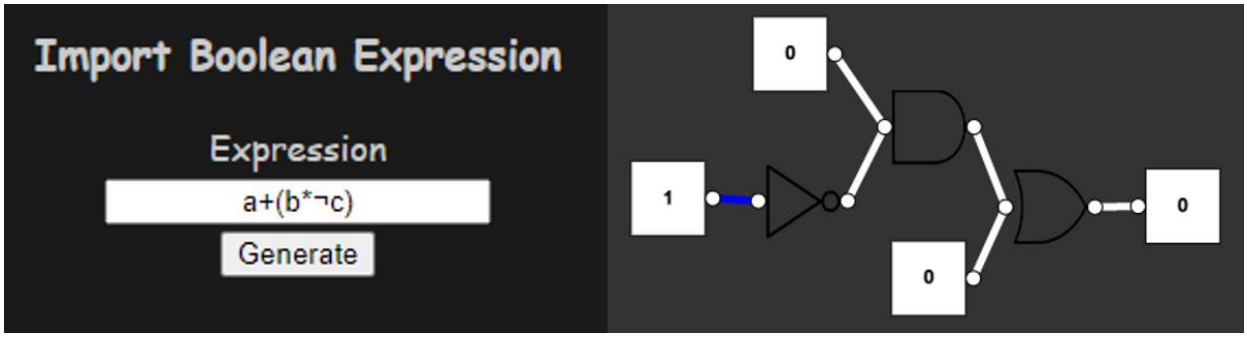
The generate Boolean function has been updated. On line 251 the expression is altered such that any matches from the symRX regex are replaced with an element in a hash-map with the replaced symbol as a key. This allows for users to use a mix-and-match of different symbols from different symbol sets. Ensuring that the software works correctly independent of the user's selection of symbols, such that users with any range of prior knowledge of Boolean expressions can use the feature.

```
20   //define a regex of gate types
21   let gateRX = /AND|OR|XOR|NAND|NOR|XNOR|NOT/g
22   let symRX = /·|∧|\*|\^|\/\\|x|×|\+|∨|\\\/|v|⊕|⊻|¬|-|!/gi
23   //deifne hash map of gate type to symbol
24   let wrtToExp = new Map()
25   let expToWrt = new Map()
```

The regex and the hash-map are defined in the above code. The symbol regex contains all symbols that a user may input in their expression and expect to have special function. These are conjoined using the '|' (OR) operator and the global and case-insensitive modifiers are set.

```
61    //fill in expToWrt
62    expToWrt.set("∧", " AND ")
63    expToWrt.set("·", " AND ")
64    expToWrt.set("*", " AND ")
65    expToWrt.set("^", " AND ")
66    expToWrt.set("×", " AND ")
67    expToWrt.set("x", " AND ")
68    expToWrt.set("/\\", " AND ")
69    expToWrt.set("∨", " OR ")
70    expToWrt.set("+", " OR ")
71    expToWrt.set("\\/", " OR ")
72    expToWrt.set("v", " OR ")
73    expToWrt.set("⊕", " XOR ")
74    expToWrt.set("⊻", " XOR ")
75    expToWrt.set("¬", " NOT ")
76    expToWrt.set("-", " NOT ")
77    expToWrt.set("!", " NOT ")
```

The following code populates the expression to written hash-map. It takes keys of the expected user-input symbols and the values are the corresponding written operator. This is enclosed in whitespace such that the parser can easily decompose the expression.



Having made these changes, the following user input, which has a mix of symbol, creates the circuit on the right. This is the correct circuit for the input and so the function is showing the expected behaviour.

```
1   let defaultPause = 100
2   //a class for clocks, as clocks will be used as inputs they extend the input class
3   class Clock extends Input{
4
5       constructor(x,y,w,h){
6         super(x,y,w,h,'')
7         this.type = 'clock'
8         this.pause = defaultPause
9       }
```

To allow for clocks components, I have created a clock class. As the clock acts in a similar way to the inputs, it extends the input class. The constructor takes in four arguments, the x and y position of the clock, and the width

and height of it as well. It then calls the constructor of the input class using super, following this it sets it type to clock and its pause duration to default pause. The default pause is a global variable that allows for the default pause time to be standardised.

```
11  //update the state of the clock
12  updateState(){
13    if (this.calcState == null){
14      this.state = (frameCount%this.pause==0) ? +!this.state : this.state
15      this.calcState = this.state
16    }
17  }
```

I have redefined the update state function for the clock class. If the calculated state is null, meaning it hasn't been called this frame, then it checks whether the modulo of the frame count and the pause duration is zero. If it is then it inverts the state, if not it remains the same. The calculated state is then set to the current state.

```
19  //display the clock
20  show(){
21    textAlign(CENTER,CENTER)
22    fill(255)
23    rect(this.pos.x, this.pos.y, this.w, this.h)
24    circle(this.pos.x+this.w+R, this.pos.y+this.h/2, 2*R)
25
26    image(clockImgOff, this.pos.x, this.pos.y, this.w, this.h)
```

I have also redefined the show function of the clock. It draws a rectangle and then places an image inside of it. This image is a square wave as it represents the clocks cycle.

```
27  //add a new clock to the circut
28  addClock(x,y,w,h){
29    this.inputs.push(new Clock(x,y,w,h))
30    //this.clocks.push(this.inputs.at(-1))
31  }
```

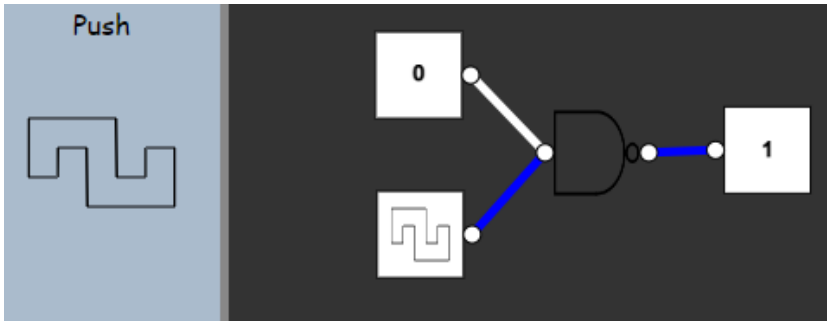
I have added a function to circuit class that allows for the adding of clocks to the circuit. It takes the arguments that are required to create a new clock, it then passes these to the constructor function of the clock class, and pushes the new clock onto the circuit's inputs array.

```
41          <!--Inputs-->
42          <div id="inputs">
43            <p id="toggle" onclick="clickInp('toggle')">Toggle</p>
44            <p id="push" onclick="clickInp('push')">Push</p>
45            <img id="clock" class="comp" onclick="clickClock()" src="/images/clock.svg">
46          </div>
```

The inputs section in the sidebar has also had a new button added to it, allowing for the creation of clocks. This is done by having an image object, which displays the clock image. Upon being clicked, the image calls the click clock function, allowing for a clock to be initialised and added to the sandbox.

```
192 function clickClock(){
193   //try 50 times to add the clock
194   for(let i=0; i<50; i++){
195     var accept = true
196     var pos = createVector(random(0,windowWidth-sideBar.width),random(0,height))
197     //check if clock is overlapping another component
198     for(let comp of [...circuit.gates,...circuit.inputs,...circuit.outputs]){
199       if(overlap({pos:pos, w:50, h:50}, comp)){accept=false; break}
200     }
201       //if accepted add the clock
202     if(accept){circuit.addClock(pos.x,pos.y,50,50);return}
203   }
204 }
```

The click clock function sets up a for loop that runs fifty times. Each time through the loop, a random position vector is then generated, and if an element in this position wouldn't be overlapping with any other component in the circuit, then it is added via a call to the add clock function. If not, the loop continues until either a clock is added, or 50 iterations occur.



A circuit built using a clock is show as well as the button in the sidebar to add a clock to the circuit. The clock switches states at a constant rate as expected, showing the correct behaviour.

```
19    //display the clock
20    show(){
21      fill(255)
22      rect(this.pos.x, this.pos.y, this.w, this.h)
23      circle(this.pos.x+this.w+R, this.pos.y+this.h/2, 2*R)
24
25      var img = (this.state==1) ? clockImgOn : clockImgOff
26
27      image(img, this.pos.x, this.pos.y, this.w, this.h)
28    }
29
30  }
```

To ensure that it is clear to the user what state a clock is in, there should be two different images for the clock; coloured depending on the users set high and low colours. To allow for this the show function of the clock selects between two different images for the clock depending on the state.

```
45  //get referance to clock images and set the colours
46  clockImgOff = loadImage("/images/clock.png", img => swapCol(img, color(255), color(lowColor)))
47  clockImgOn = loadImage("/images/clock.png", img => swapCol(img, color(255), color(highColor)))
48
```

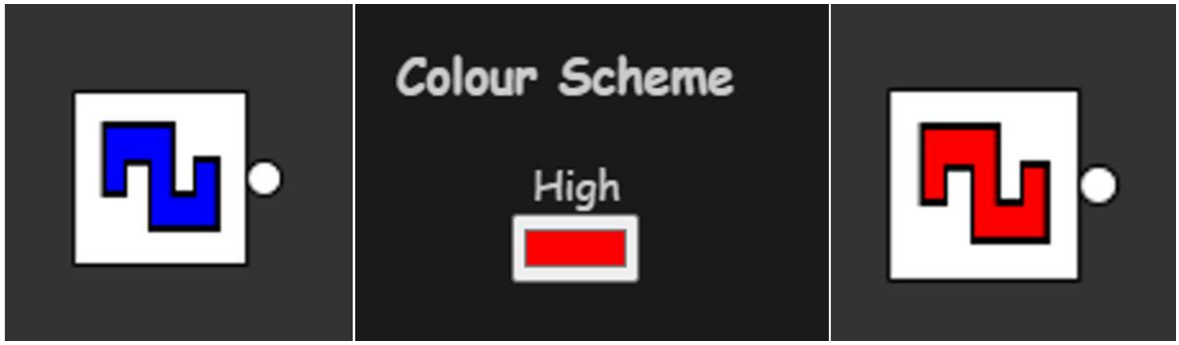
To allow for two different clock images, two images must be loaded in. this takes place in the preload function, initialising the two images, and passing the load image function a second argument which is a callback function. The callback function is called when the asynchronous function returns, allowing the rest of the code to execute without having to wait for the access of the images from storage.

```
606  function swapCol(img, find, rep){
607    img.loadPixels()
608    //loop through each pixel
609    for(let y=0; y<img.height; y++){
610      for(let x=0; x<img.width; x++){
611        if(img.get(x,y)[0]==find.levels[0] && img.get(x,y)[1]==find.levels[2] && img.get(x,y)[2]==find.levels[2]){img.set(x,y,rep)}
612      }
613    }
614    img.updatePixels()
615  }
```

The swap colour function takes an image along with two colours as arguments. It then loops through each pixel in the image. It then checks if the colour at that pixel matches the colour that should be replaced. If it is, then the colour at that pixel is replaced with the replacement colour. The code at line 607 and 614 are specific to the P5.JS library that handles the images, the load the pixels of the image into memory in the form of an array of 8-bit unsigned integers and then update the image to that of the altered array.

```
249        break
250      case "highCol":
251        clockImgOn = loadImage("/images/clock.png", img => swapCol(img, color(255), color(col)))
252        highColor = col
253        break
254      case "lowCol":
255        clockImgOff = loadImage("/images/clock.png", img => swapCol(img, color(255), color(col)))
256        lowColor = col
257        break
```

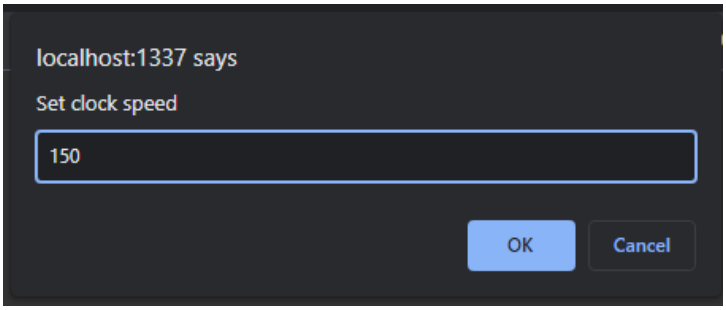
The set colour function has also been altered such that when a new high or low colour is set, the images for the clock are updated to reflect the new colour scheme.



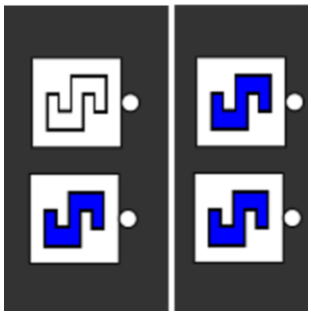
The swap colour function can be seen working in the above example. The high colour of the clock is initially blue, the user then sets the high colour to red, and this update is then seen in the clocks display.

```
31    //called when centre mouse button is pressed on the clock
32    centreMouse(){
33      if (this.mouseOver()){
34        //prompt user for speed and remove any non-digits
35        let nSpeed = prompt('Set clock speed',''+this.pause).replace(/[^0-9]/g,'')
36        //set the clocks pause to the new speed if its valid
37        this.pause = (nSpeed == null || nSpeed == "" || +nSpeed<minSpeed) ? this.pause : +nSpeed
38      }
39    }
40
41  }
```

To allow users to change the speed of their clocks, this function was added to the clock class. When the mouse centre button is pressed, this function is called, it checks initially whether the mouse is over the clock. If it is it prompts the user to input a speed for the clock. It then sets the speed of the clock to this new speed if it is valid. For a speed to be valid it must be an integer above the minimum speed. If the new speed is invalid then the clocks speed is unchanged.



Upon pressing the centre mouse button above a clock, this menu pops up. This prompts the user to input a speed, it shows the user the current clock speed and allows them to input a new one. The user can then either click ‘ok’ to update the speed or ‘cancel’ to leave the speed as is.



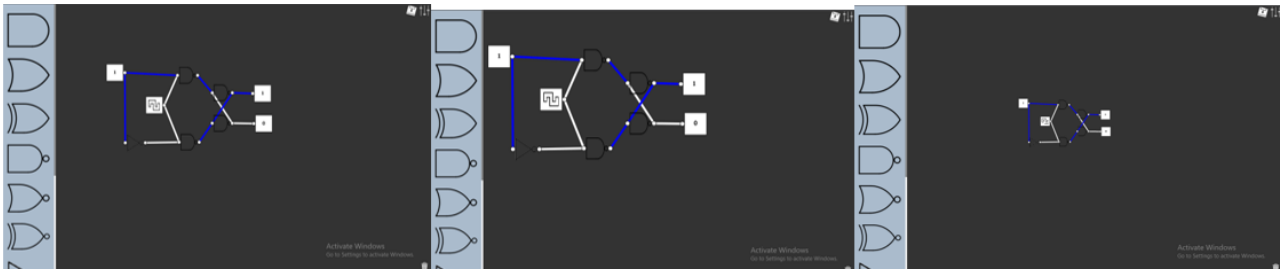
Two clocks can now be set to oscillate at different speeds, such that they can be in and out of phase with each other, and this difference can change. This is shown in the above image.

```
188   //called on mouse scroll
189   function mouseWheel(event){
190     if (mouseX>sideBar.width && menu==null){
191       zoom -= event.delta
192     }
193   }
```

By giving the user the ability to zoom in and out, they can build larger circuits as well as making the simulation more accessible for users with smaller displays. The function to allow for zooming is an event handler for the mouse wheel scroll event. It checks whether the mouse is over the sandbox and there is no menu open. If it is, then the variable zoom is deprecated by the mouse scroll delta.

```
110   //zoom to current level
111   translate(width/2,height/2)
112   scale(1+zoom/1000)
113   translate(-width/2, -height/2)
```

This code is run at the start of the draw loop. It translates the canvas' origin to the centre of the sandbox, it then scales the canvas by one plus one-thousandth of the zoom. This zooms in or out the canvas, the zoom is divided by 1000 as the delta that it is changed by on mouse scrolls is 100.



This is shown working above, giving users the ability to zoom the sandbox both in and out, without changing the sidebar or the auxiliary menus/buttons. However, when in a state with an altered zoom, the drag-and-drop functionality along with clicking on a component cease to work as expected.

```
142    //undo zoom to allow for mouse functionality
143    translate(width/2,height/2)
144    scale(1/(1+zoom/1000))
145    translate(-width/2, -height/2)
```

This is due to the mouse location variable no being changed by the zoom. Therefore, when the mouse appears to be above the component, the mouse coordinates don't match up with the actual component coordinates, only the apparent coordinates. The above code should undo the zoom at the end of the draw loop, such that the coordinate system is back in line.

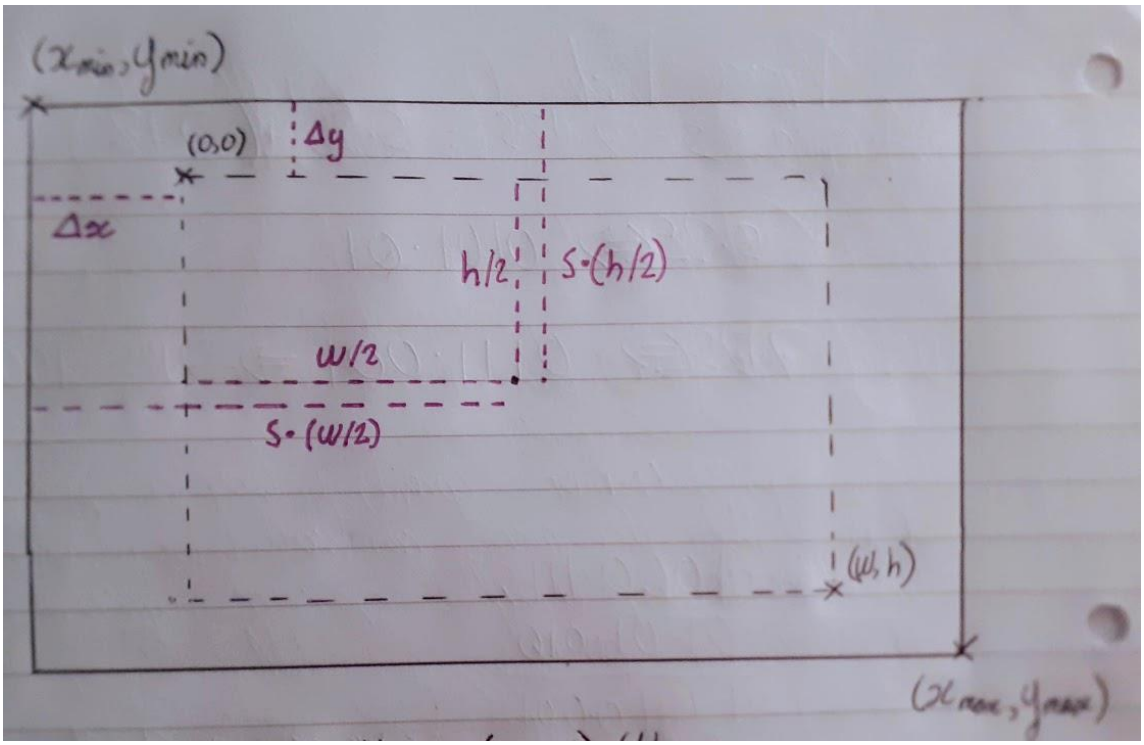
This can be shown working with the drawing of circles in the centre and corners before and after undoing the zoom. The white dots are drawn before undoing the zoom, and change location depending on zoom level, whereas the red dots are drawn after the zoom is undone, and they stay in the centre and corners no matter the level of zoom.

```
61   //a function to translate the mouse position to the zoomed position
62   function transformMouse(){
63     var s = 1/(1+zoom*zoomSF) //define the scale factor
64     var x = s*(mouseX-width/2) + width/2
65     var y = s*(mouseY-height/2) + height/2
66     return {x,y}
67   }
```

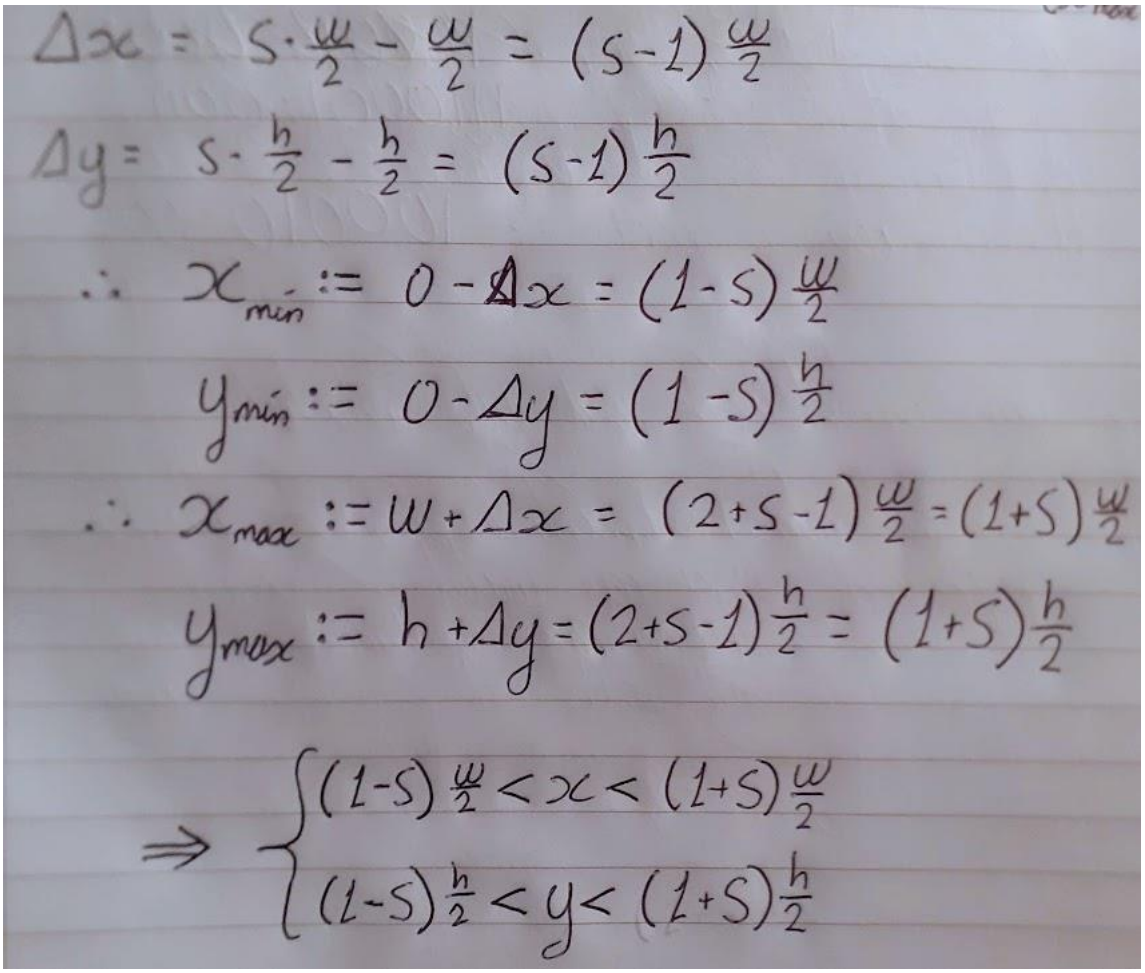
The scale function applies a transformation matrix to only the canvas itself, this means that the mouse position is still absolute and not given relative to the transformations. For this reason, I have created a function that returns the relative position of the mouse for a given scale. It first calculates the scale factor, using this it transforms the mouse's x and y positions such that they will be accurate to the apparent position of elements in the sandbox. These new positions are then returned.

```
19    //a function that is run each frame and updates the position of the comp
20    updatePos(){
21      if (this.dragging){
22        let {x,y} = transformMouse()
23        let s = 1/(1+zoom*zoomSF)
24
25        //copy the value of the position if we need to reset it
26        var oldPos = Object.assign({}, this.pos)
27
28        this.pos.x = this.offset.x + x
29        this.pos.y = this.offset.y + y
30
31        //check if component is overlapping any others
32        for (let compB of [...circuit.gates,...circuit.inputs,...circuit.outputs]){
33          if(overlap(this, compB) && compB!=this){this.pos = Object.assign({}, oldPos)}
34        }
35
36        //ensure the component doesnt go off-screen
37        if (this.pos.x < (1-s)*width*0.5){this.pos.x = (1-s)*width*0.5}
38        if (this.pos.y < (1-s)*height*0.5){this.pos.y = (1-s)*height*0.5}
39        if (this.pos.x + this.w > (1+s)*width*0.5){this.pos.x = (1+s)*width*0.5 - this.w}
40        if (this.pos.y + this.h > (1+s)*height*0.5){this.pos.y = (1+s)*height*0.5 - this.h}
41      }
42    }
43
```

At all locations in which the mouse position was used, it has been replaced by the new scaled x and y values attained by calling the transform mouse function. This is shown in the update position function. This function has also changed in checking whether the component is off screen. Prior to the change components would only be movable to the width of the original screen scaled down, even if more screen was visible due to zooming out. By changing the conditional statements to include the scale factor, this allows components to be dragged to the edge of the sandbox, no matter the level of zoom.



To calculate the new conditions, I worked out the maximum and minimum possible x and y values for a given scale factor 's'. this was done by using the difference between the sandbox's actual origin and its apparent origin. These values are label 'Δx' and 'Δy' respectively.



The values of Δx and Δy are calculated to be half the width or height multiplied by one less than the scale factor

$$\Delta x = \frac{(s-1)\cdot w}{2}$$

$$\Delta y = \frac{(s-1)\cdot h}{2}$$

The minimum values of x and y are then calculated by subtracting Δx and Δy from the origin (0,0)

$$x_{min} := 0 - \Delta x = \frac{(1-s)\cdot w}{2}$$

$$y_{min} := 0 - \Delta y = \frac{(1-s)\cdot h}{2}$$

The maximum values of x and y are calculated by adding Δx and Δy to width and height

$$x_{max} := w + \Delta x = \frac{(1+s)\cdot w}{2}$$

$$y_{max} := h + \Delta y = \frac{(1+s)\cdot h}{2}$$

With these minimum and maximum values, we can calculate a bound for x and y positions of components in the scaled sandbox

$$\Rightarrow \begin{cases} \frac{(1-s)\cdot w}{2} < x < \frac{(1+s)\cdot w}{2} \\ \frac{(1-s)\cdot h}{2} < y < \frac{(1+s)\cdot h}{2} \end{cases}$$

These are the conditions that were added to the update position function, with the width of the component added when checking for overlap with maximal values. This ensures that the user can't accidentally drag a component off screen as this would negatively impact useability.

```
671  function setCookie(){
672    if(document.cookie != ''){deleteCookies()}
673    //create a cookie for the users colour scheme and boolean algebra type
674    var c = 'hc='+highColor+';'
675    document.cookie = c
676    c = 'lc='+lowColor+';'
677    document.cookie = c
678    c = 'dg='+dragColor+';'
679    document.cookie = c
680    c = 'bg='+bgColor+';'
681    document.cookie = c
682    c = 'sb='+sbColor+';'
683    document.cookie = c
684    c = 'ba='+boolType+';'
685    document.cookie = c
686  }
```

To allow for a more personalised user experience, the user's alterations to the settings should be persistent, even after the leave the page. For this I will make use of browser cookies, which are small text files that are stored locally on the user's browser. This will allow for a consistent and unique experience for each individual user. The above function sets these cookies for the user. It first checks whether any cookies are currently set for the page, if there are then it deletes them. Following this it repeatedly sets the value of a variable 'c' to a string containing an identifier, such as 'hc' for high colour, and its current value. It then writes this value to the cookie. Although the syntax makes it appear that the cookie is overwritten and only the final value will be present, the cookie variable has a special usage in which each time it is re-defined, it simply appends the new value to the current ones.

```
> document.cookie
< ''
> setCookie()
< undefined
> document.cookie
< 'hc=#0000ff; lc=#ffffff; dg=#969696; bg=#3333
  33; sb=#aabbcc; ba=logic'
```

The effect of the set cookie function can be seen above. The cookies are initially blank, and after executing the function the cookies are set to the user's current settings. The unique functionality of the cookie variable can also be seen, as after the function is executed, its value is simple every item written to it appended together.

```
688  //a helper function to delete current cookies
689  function deleteCookies(){
690    const cookies = document.cookie.split(';') //set var to array of each part of the cookie
691    for(const cookie of cookies){ //Loop through array
692      const name = (cookie.indexOf("=")>-1) ? cookie.substring(0, cookie.indexOf("=")) : cookie
693      //set experation date to epoch so it is removed
694      document.cookie = name + "=;expires=Thu, 01 Jan 1970 00:00:00 GMT"
695    }
696  }
```

Given the unique functionality of the cookie variable, we cannot delete cookies by simply setting the cookie variable to an empty string; as all that this will do is append an empty string to the existing cookie leaving it unchanged. For this reason, I have written a helper function that can delete cookies on demand. It initially creates an array from the cookie by splitting the string up on each semicolon. This array then contains each individual part of the cookie. It then loops over this array and sets the variable 'name' to the identifier of the cookie. It does this by selecting a substring of the cookie from the start to before the equal sign. It then rewrites the cookie to the cookie variable with the expiration date set to the UTC epoch. This ensure that the browser removes the cookie as it is expired.

```
> document.cookie
< 'hc=#0000ff; lc=#ffffff; dg=#969696; bg=#3333
  33; sb=#aabbcc; ba=logic'
> deleteCookies()
< undefined
> document.cookie
< ''
```

The effect of calling the delete cookies function can be seen above. The cookie is initially set to the user's settings, however, after the function executes the cookie is empty.

```
<input type="button" value="Default Settings" onclick="defaultSettings()">
```

Having added persistence in the form of cookies, you can no longer reset to the default settings by simply reloading the page. Given it is often useful to be on the default settings, the colour scheme being most appealing for instance, I have given the user the ability to return to the default settings. This is done through a button in the settings menu. The button calls the default settings function.

```
309    }
310    setCookie()
311  }
312
313  //set the boolean type
314  function setBoolType(){
315    boolType = document.getElementById("boolGrams").value
316    setCookie()
317  }
```

The set cookie function is called whenever the user changes a colour in the settings, as shown in line 310, or when they change the Boolean type, as shown in line 316.

```
40  //check if a cookie is present, if so get the colour scheme from it
41  if(document.cookie != ''){
42    try{
43      //split cookie into colour vairables and remove whitespace
44      var result = document.cookie.replace(/\s/g, '').split(';').slice(0,6)
45      //set colours from cookie
46      highColor = result[0].slice(3)
47      lowColor = result[1].slice(3)
48      dragColor = result[2].slice(3)
49      bgColor = result[3].slice(3)
50      sbColor = result[4].slice(3)
51      boolType = result[5].slice(3)
52      setHTML()
53    }catch (error){
54      console.log(error)
55      //set colours to default if not in cookies
56      highColor = "#0000ff"
57      lowColor = "#ffffff"
58      dragColor = "#969696"
59      bgColor = "#333333"
60      sbColor = "#aabbcc"
61      boolType = "logic"
62      setHTML()
63    }
64  }
```

To allow for the cookies to affect the site, during the preload function a check is made to see whether any cookies are set. If they are then the cookies firstly have any whitespace removed, then are split into an array on each semicolon. The colour scheme along with the Boolean type are then set to the cookie's values, using slice to remove the identifier and equals sign leaving just the data. Set HTML is then called to update any DOM elements. If any errors occurs, such as from an invalid or malformed cookie, the error is caught and the default values are set.

```
698  function defaultSettings(){
699    //get temp value of high/low colours
700    const _low = lowColor
701    const _high = highColor
702    //set the default colours
703    highColor = "#0000ff"
704    lowColor = "#ffffff"
705    dragColor = "#969696"
706    bgColor = "#333333"
707    sbColor = "#aabbcc"
708    swapCol(clockImgOff, color(_low), color(lowColor))
709    swapCol(clockImgOn, color(_high), color(highColor))
710    //set the boolean type
711    boolType = "logic"
712    //set the HTML values
713    setHTML()
714    setCookie()
715  }
```

The default settings function initially sets two temporary variables containing the current high and low colours. Following this it sets all the colours to the default colour scheme, along with calling the swap color function on the clock images, passing the temporary old colour values as the detect colour, and the new high and low colours as the replacements. It then sets the Boolean type to logical. It then calls the set HTML function and the set cookie function to ensure that the changes appear on the page and are persistent.

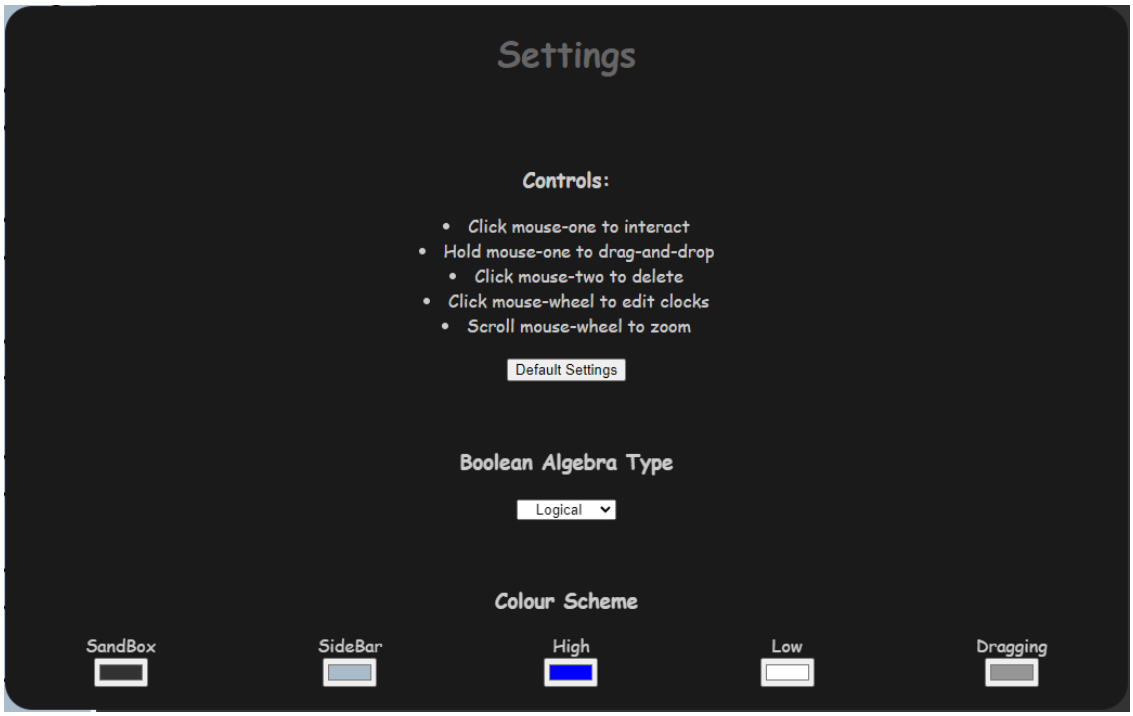
```
716  //set the html elements to the current expected values
717  function setHTML(){
718    document.getElementById('boolGrams').value = boolType
719    document.getElementById('sandBoxCol').value = bgColor
720    document.getElementById('highCol').value = highColor
721    document.getElementById('lowCol').value = lowColor
722    document.getElementById('sideBarCol').value = sbColor
723    document.getElementById('sideBar').style.backgroundColor = sbColor
724    document.getElementById('dragCol').value = dragColor
725  }
```

The set HTML function updates the DOM elements to reflect the current values of their respective variables. It first sets the Boolean-type drop-down menu to the current Boolean type. Following this it sets each of the colour selectors to the current colours. It also sets the background colour of the sidebar to the current side bar colour variable. This is the only style that needs to be updated in this function as all the other colours are set in the draw loop and therefore are automatically updated.

The use of the default settings button can be seen above. As after the button is pressed, all the altered settings returned to normal. This includes both the colour scheme and Boolean type.

```
<h3>Controls:</h3>
<ul>
  <li>Click mouse-one to interact</li>
  <li>Hold mouse-one to drag-and-drop</li>
  <li>Click mouse-two to delete</li>
  <li>Click mouse-wheel to edit clocks</li>
  <li>Scroll mouse-wheel to zoom</li>
</ul>
```

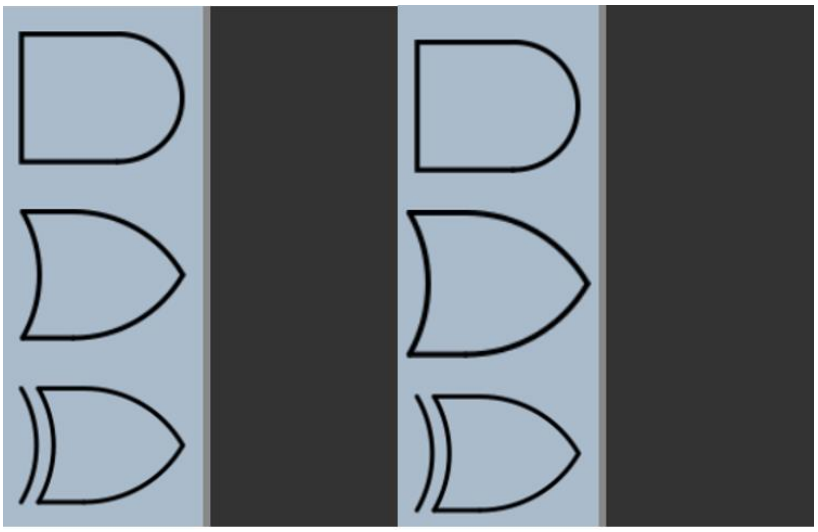
It is important that the software is easily usable, as this will make it available to a wide audience. Therefore, I have added a list of controls to the settings menu, this allows users to easily see how to control and interact with the simulation. The controls are displayed as an unordered list, so will be a set of bullet points displayed to the user.



The new settings menu can be seen above, with the new controls list and default settings button. The menu has usable functionalities while remaining uncluttered and readable, this ensures that it is easy to use even for new or unexperienced users.

```
160
161  @keyframes twist {
162    0%   {transform: rotate(0deg);}
163    25%  {transform: rotate(-30deg);}
164    50%  {transform: rotate(0deg);}
165    75%  {transform: rotate(30deg);}
166    100% {transform: rotate(0deg);}
167  }
168
169  .menuButton{
170    width:35px;
171    height: 35px;
172    transform: scale(0.9);
173  }
174
175  .menuButton:hover{
176    /*animation: twist .5s;*/
177    transform: scale(1);
178  }
179
180  .comp{
181    transform: scale(0.9);
182  }
183  .comp:hover{
184    transform: scale(1);
185  }
```

To make the use of the simulation more intuitive, I have added animations to the buttons. When hovering over a button it enlarges slightly making it obvious to the user that they can interact with it. This information is passed to the user in a simple way without having to bombard them with text or instruction, ensuring that they learn to use the simulation organically and in their own way. I initially had the buttons complete a twist animation when hovered over, and while this drew attention to the button, it didn't convey its usage as elegantly as the simple scaling did; as the effect of the animation should be to highlight the intractability of the button while not being distracting. To do this I added a default scale down to 90% of the buttons size, this scaling is removed when the mouse hovers over the component. This functionality is defined to both the menu button and component button classes.



The enlargement can be seen in the above image in the OR gate. The difference is not too large as to distract the user but enough to be noticeable. The effect is clearer in use when you can see the change happen fluidly with the mouse movement. The screenshot also removes the curser from the image making it less clear in the image than in real use.

```
601     //inputs
602     }else{
603       //set the type to clock or input depending on component
604       const type = (comp.type == 'clock') ? 'Clock' : 'Input'
605       return {type:type, id:circuit.inputs.indexOf(comp)}
606     }
```

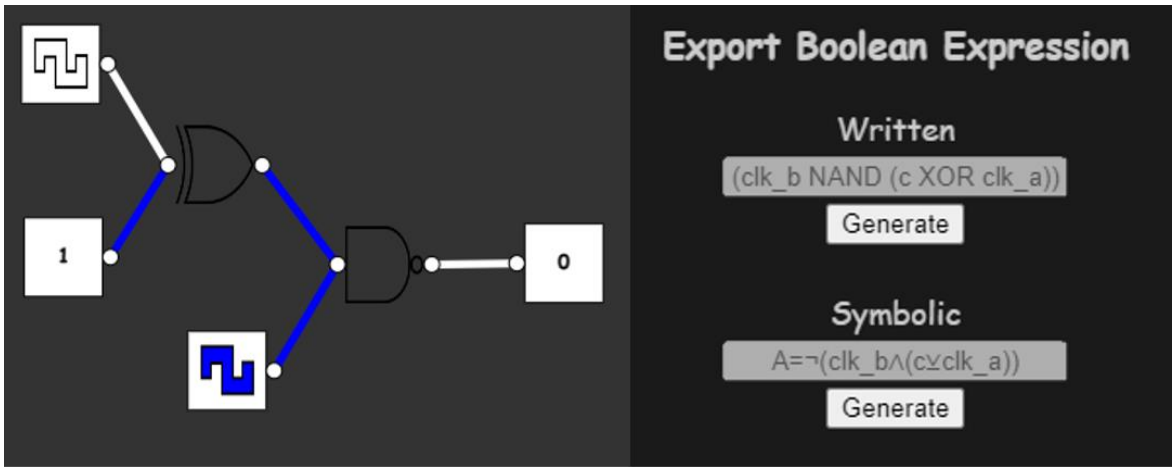
Having added clocks to the simulation, I wish to extend all previous functionalities to be able to make use of them. The first alteration is in the circuit to tree function, where I have added a condition at line 604 which either sets the type of the node to 'Clock' or 'Input' depending on the component. This allows the software to distinguish between the types of outputs in user-built circuits when converting them to trees.

```
609   function inOrder(node){
610     if(node.type=="Input"){
611       return inputNames[node.id]
612     }else if(node.type=="Clock"){
613       return 'clk_'+inputNames[node.id]
614     }else if(node.type=="Unary"){
615       return "("+node.comp+" "+inOrder(node.child)+")"
616     }else{
617       return "("+inOrder(node.left)+" "+node.comp+" "+inOrder(node.right)+")"
618     }
619   }
```

The in-order traversal is then altered to include differentiate between input types. This occurs by the second conditional on line 612 that checks whether the type is 'Clock'; and if it is, it appends 'clk_' to the input name to make it obvious in the output. The same change is made in the in-order symbolic function as well.

```
622     if(node.type=="Input" || node.type=="Clock"){return node}
```

The following change was also made in the pure Boolean function to ensure that it works with the new clock type.



With these changes made, the new output of this user-built circuit in both written and symbolic form contains the ‘clk’ prefix for all the clocks. This makes it easy for the user to see the functionality of even their complex circuits which make use of both normal inputs along with clocks.

```
347         //new input
348         if(exp.name.slice(0,3).toUpperCase()=='CLK'){
349           circuit.addClock(random(sideBar.width+50,width-sideBar.width-50), random(height-50), 50, 50)
350         }else{
351           circuit.addInput(random(sideBar.width+50,width-sideBar.width-50), random(height-50), 50, 50, "toggel")
352         }
```

To allow for users to generate circuits containing clocks from a Boolean expression, the following changes to the build tree function have been made. A new condition has been added for when a new input if found in the expression. If the name of the input is prefixed with ‘clk’, then a clock is added to the circuit, otherwise a toggle input is added. This gives the users more creative control in their circuits, and also allows them to share more complex circuits with each other easily.

With these changes made, users can generate circuits with a mixture of clocks and inputs from a simple Boolean expression.

An error is occurring with the circuit generation from Boolean expression. Any expression that contains either an XOR or XNOR gate simply creates a blank circuit, meaning an error is occurring during either the parsing of the expression or the generation of the circuit.

```
23  //define a regex of gate types
24  let gateRX = /AND|OR|XOR|NAND|NOR|XNOR|NOT/g
25  let symRX = /·|∧|\*|\.|\^|\/\\|x|×|&|\+|∨|\\\/|v|\||⊕|⊻|¬|-|!/gi
26  //deifne hash map of gate type to symbol
```

Upon inspection and debugging, it is apparent that the ‘X’ in the XOR or XNOR is being removed by the symbol regex and replaced with ‘AND‘. This causes an error as it leads to an AND gate and an OR gate being directly next to each other with inputs.

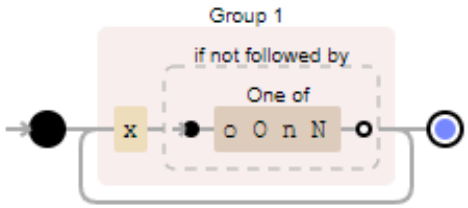
For instance:

a XOR b -> a AND OR b

This is due to the case-insensitive flag being set in the regex. This is the ‘i’ at the end of line 25 as it sees the X in XOR as a user-inputted multiplication sign and interprets this as an AND gate.

```
24  let gateRX = /AND|OR|XOR|NAND|NOR|XNOR|NOT/g
25  let symRX = /·|∧|\*|\.|\^|\/\\|x(?![oOnN])|x|&|\+|∨|\\\/|v|\||⊕|⊻|¬|-|!/gi
```

By changing the regex to the following the error shouldn’t occur. The regex now only catches an ‘x’ if it is not directly followed by an ‘o’ or an ‘n’. this means that the word XOR or XNOR won’t be caught but just using ‘x’ as the multiplicative symbol will be.



A representation of this part of the regex

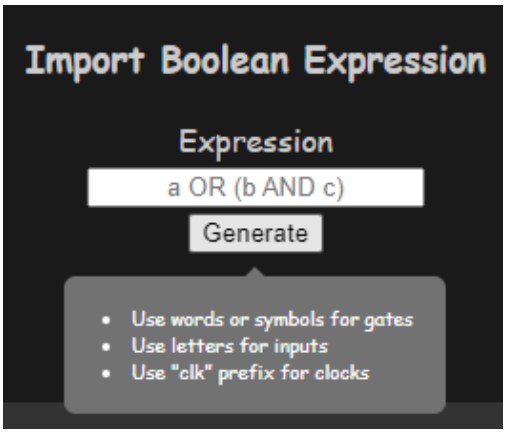
Having changed the regex, the generation occurs as expected, with the ability to add XOR and XNOR gates without any errors occuriong.

```html
<div class="ttip">
  <input type="button" value="Generate" onclick="genBool()">
  <span class="ttipText"><ul>
    <li>Use words or symbols for gates</li>
    <li>Use letters for inputs</li>
    <li>Use "clk" prefix for clocks</li>
  </ul></span>
</div>
```

With the updates made to the generation of circuits from expression it contains a large volume of functionality. Due to this it may be daunting or confusing to new users. To avoid this, I will add a tooltip to the generation, as this won't be intrusive or distracting, but will inform the user of how to use the feature. The tooltip itself will be made up of an unordered list, so will be displayed as a set of bullet points to the user. To allow for the tooltip, I have added the button along with the tooltip to a div, and the tooltip itself into a span element.

```css
.ttip {
  position: relative;
  display: inline-block;
}
.ttip .ttipText {
  visibility: hidden;
  width: 200px;
  top: 150%;
  left: 50%;
  margin-left: -100px;
  background-color: rgb(114, 114, 114);
  color: #fff;
  text-align: center;
  padding: 5px 0;
  border-radius: 6px;
  position: absolute;
  z-index: 1;
}
.ttip .ttipText::after {
  content: " ";
  position: absolute;
  bottom: 100%;
  left: 50%;
  margin-left: -5px;
  border-width: 5px;
  border-style: solid;
  border-color: transparent transparent rgb(114, 114, 114) transparent;
}
.ttip:hover .ttipText {
  visibility: visible;
}

.ttipText>ul{
  padding-left: 10%;
  font-size: 10px;
  text-align: left;
}
```

The following CSS allows for the tooltip to act as expected, hiding the text until the user hovers over the button. The CSS is based of the W3School tooltip guide. The block from line 193 to 195 sets the position of the tooltip container to be to its parent and for it to be displayed inline. Lines 197 to 209 hides the tooltip and sets its colour along with its shape and boarders. It also places it above the background so it can be seen when its reveled. Lines 211 to 220 add an arrow to the tooltip such that it points to the button, this makes it obvious what the tooltip is referring to. Lines 221 to 223 cause the tooltip to show when the mouse hovers over the button, allowing for the tooltip's functionality. Lines 225 to 229 affect the list inside the tooltip, they set the text size along with aligning the text to the left as opposed to the centre.



The screenshot shows the tooltip in action, displaying usage information to the user in an unintrusive way. The tooltip is usually hidden and only displays when the mouse is over the button, this isn't obvious from the screenshot as the curse is removed, but the mouse is over the button.

REVIEW

The third and final iteration has been completed, this is inline with my time constraints. The iteration has consisted of the development of the final success criteria as well as some additional features and functionality from my own, and the stakeholders, input. The new and existing code has also been tested to ensure the correct functionality.

WHAT HAS BEEN DEVELOPED

The initial key focus of the third iteration was the development and implementation of clocks. This included their functionality in the sandbox; their creation, deletion, movement, updating, interaction, and editability. The basic control of the clock object was relatively simple to implement as I could just make a child class of input and copy across much of the attributes and methods using inheritance. Polymorphism also aided in the development of function that acted differently to the parent class, such as the show function or the update function, but could still be called with the same name. The show function required a reasonable amount of background work, in the form of the image handling and editing. However, time in the long run was saved by writing and making use of a helper function to swap colours. The update function was computationally easy to implement, as I could simply make use of the frame count and the modulus function. This gave me an easy way to check whether the state should change, as well as making it easy to allow for changing speeds. The changing of clock speeds was implemented different to many seemingly similar features. This was because the changing of clock speeds required user input in reference to only a single component as opposed to an entire circuit. Due to this it was handled in a unique way.

A large amount of the early development focused on completing functionality to display circuits as Boolean expressions. This included the development of the input and output name generator which created a list of logical and distinct names for the inputs and outputs of a circuit. The development of an in-order tree traversal algorithm was also completed, allowing for a correct and consistent written form of the circuit, which was held in a data structure that could be visualised as a tree. A recursive function was also developed that acts over the tree data structure, which located and edited areas of the tree which couldn't be represented using symbolic Boolean algebra. Hash-maps and regexes were also made use of to allow for the translation between written and symbolic Boolean algebra. With this, along with a symbol-specific tree traversal function, the ability to represent a user-built circuit with symbolic Boolean algebra was implemented. These also made it possible to increase the domain of inputs available for generating a circuit form a Boolean expression, as the logic was already in place for written Boolean and symbolic Boolean can now be directly translated to written Boolean.

Alongside the implementation of clocks, the third iteration also included many of what could be called 'quality of life' improvements. These made the simulation seem more personal, and less demanding to use. A large part of this was the implementation of cookies, this allowed for persistence of settings by using local browser storage. This ensures that the site seems more personalised and consistent throughout multiple uses. The cookies also required a large number of helper function, such as for setting and deleting them. With the new settings persistence, a new way of returning to default values was required, so a default settings function was developed, allowing for users to return to the initial settings at will. Along with this a function to update the DOM to its current expected state was developed, ensuring consistency throughout all parts of the site. Other feature for the user were also implemented, such as zoom functionality which allows users to work on even larger circuits than before without sacrificing space or readability. Users were also given the ability to use a wide range and mix of symbols and

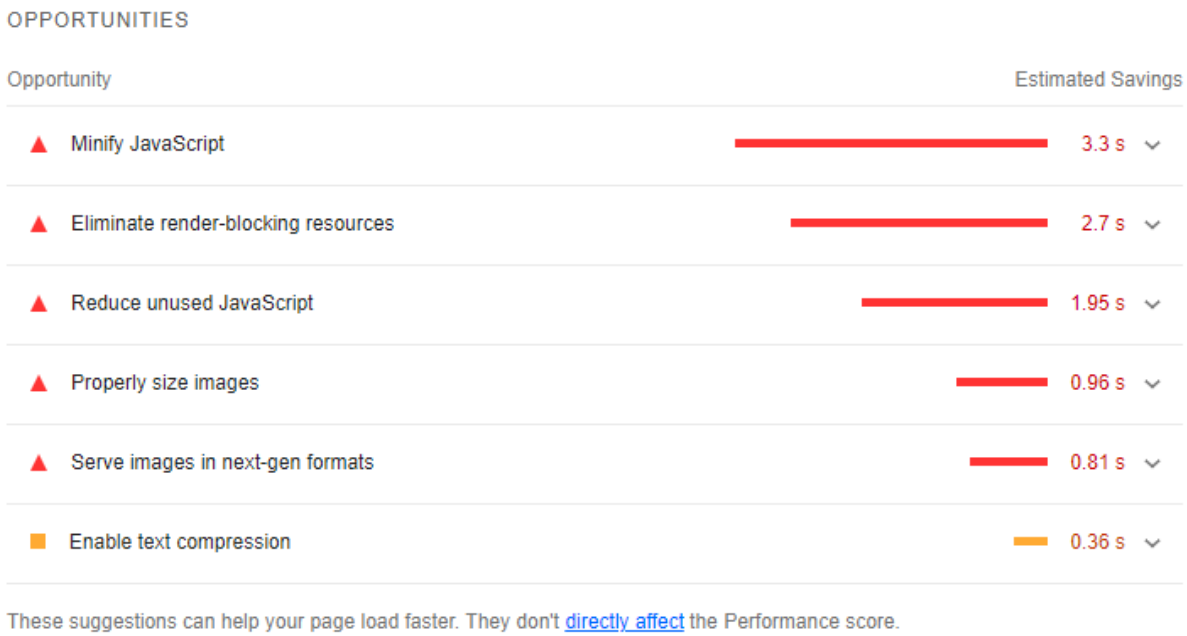
prefixes in the circuit generation. This gives them more power and control, as well as chance of mistake by ensuring that it is usable no matter prior knowledge of different symbol sets. A list of controls was given to the user so that they can easily and quickly learn how to make use of the simulation, along with a tooltip system making data input simpler to understand. Several graphical upgrades were also made, using CSS to show the intractability of buttons, as well as adding a clear all button to the sandbox, removing the timely process of removing each component one by one.

Criteria met
Allow the use of clocks as inputs to the circuit
Allow the user to edit the speed of clocks
Allow the user to generate a Boolean expression of their circuit

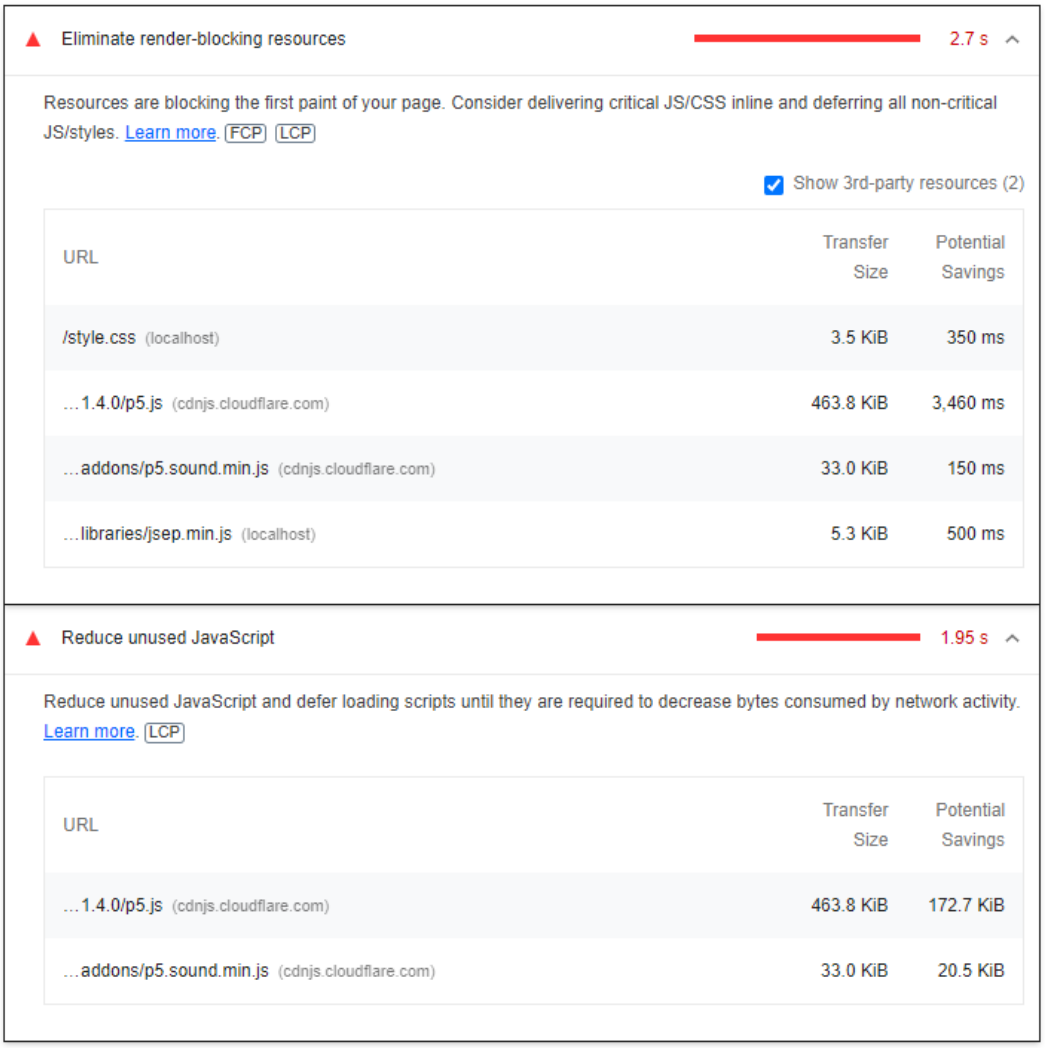
With this iteration, all of the success criteria posed in the design section have been met. There was only one major diversion from the design in this iteration, changing clock speeds. The initial design for this process would have involved the user going into the settings menu to alter the speed of specific clocks. However, this would push the burden of using the correct id for a clock to the user. Not only would this method be more complex for the user, but it would require a new id system for the components which wasn't necessary with the object-based approach. Due to this, me and the stakeholders decided that a much more intuitive way to change a clocks speed would be by interacting with it directly, using the mouse and having a specific pop-up menu.

TESTING AND FEEDBACK

With this being the final iteration, I intend to test the performance of the simulation. This will be done using the Lighthouse report generator, which creates a report pertaining to specific performance and usability optimisation. This should allow me to test the simulation in general and not simply in specific areas or functions.



By testing the performance of my code, areas for optimisation have been identified. The largest time save if from minifying the JavaScript, however this is not sensible to do during development as it would make the code hard to read and edit, however this would make sense to do after development is complete. The other non-negligible time saves can be made by removing resources that block initial rendering and JavaScript that is unused.



▲ Eliminate render-blocking resources — 2.7 s

Resources are blocking the first paint of your page. Consider delivering critical JS/CSS inline and deferring all non-critical JS/styles. Learn more. FCP LCP

Show 3rd-party resources (2)

URL	Transfer Size	Potential Savings
/style.css (localhost)	3.5 KiB	350 ms
…1.4.0/p5.js (cdnjs.cloudflare.com)	463.8 KiB	3,460 ms
…addons/p5.sound.min.js (cdnjs.cloudflare.com)	33.0 KiB	150 ms
…libraries/jsep.min.js (localhost)	5.3 KiB	500 ms

▲ Reduce unused JavaScript — 1.95 s

Reduce unused JavaScript and defer loading scripts until they are required to decrease bytes consumed by network activity. Learn more. LCP

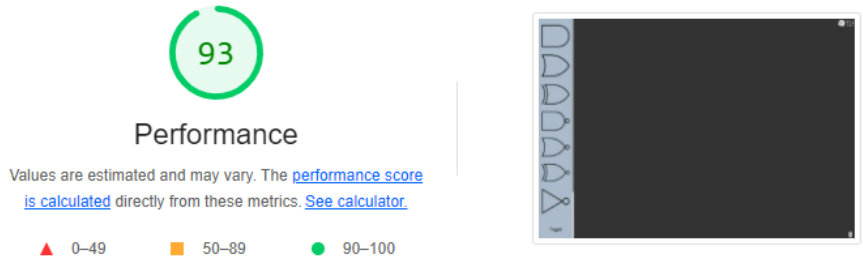
URL	Transfer Size	Potential Savings
…1.4.0/p5.js (cdnjs.cloudflare.com)	463.8 KiB	172.7 KiB
…addons/p5.sound.min.js (cdnjs.cloudflare.com)	33.0 KiB	20.5 KiB

Both of these are occurring due to including of all parts of the P5.JS library. However, certain parts of the library are not being used, such as sound handling. This means that these parts can be removed, and a lighter version of the library can be implemented. This should increase the performance of the simulation allowing it to be used effectively on weaker hardware, ensuring that the software is accessible to a wide range of audiences. Alterations to the use of images in the simulation could have a minor improvement to load times, however, due to the nature of editing images in real time and sizing them specifically to the user's machine, this alteration cannot easily be made while keeping functionality.

OPPORTUNITIES

Opportunity	Estimated Savings
Minify JavaScript	0.44 s
Eliminate render-blocking resources	0.35 s
Reduce unused JavaScript	0.24 s
Properly size images	0.2 s
Serve images in next-gen formats	0.2 s

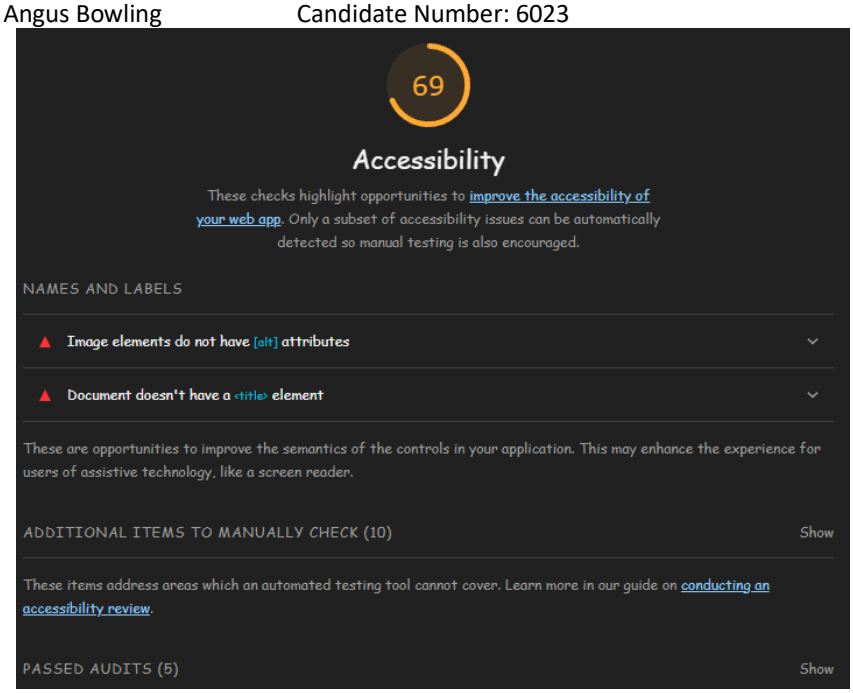
Having made these changes, the performance of the simulation is greatly increased with the load bottlenecks being significantly decreased. This allows the use of the site even on systems with older processors and weak internet connections (slow download speeds).



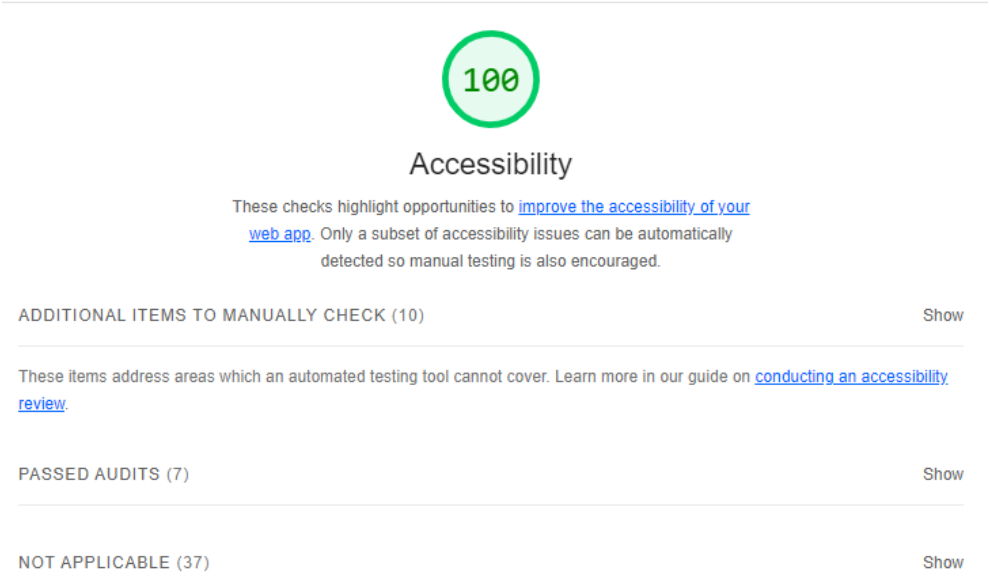
METRICS Expand view

First Contentful Paint 0.7 s	Time to Interactive 1.3 s
Speed Index 2.1 s	Total Blocking Time 60 ms
Largest Contentful Paint 1.2 s	Cumulative Layout Shift 0.061

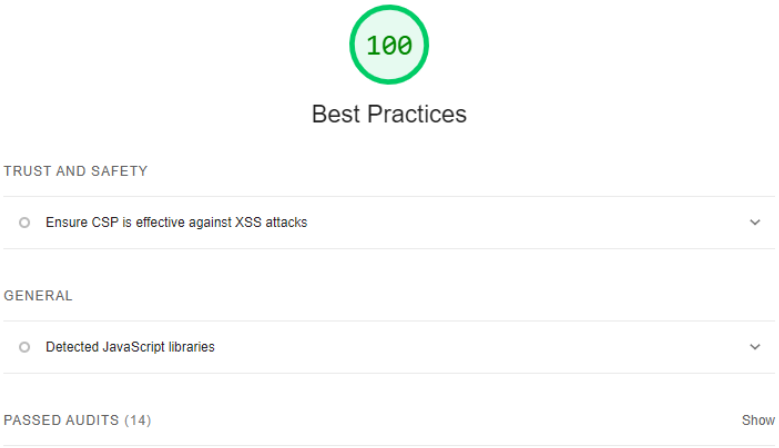
The overall performance of the page is increased, with room to increase the speed index upon minification of the JavaScript. For an interactive site having a low time to interactive and a low blocking time means that the user isn't sitting waiting to make use of the simulation



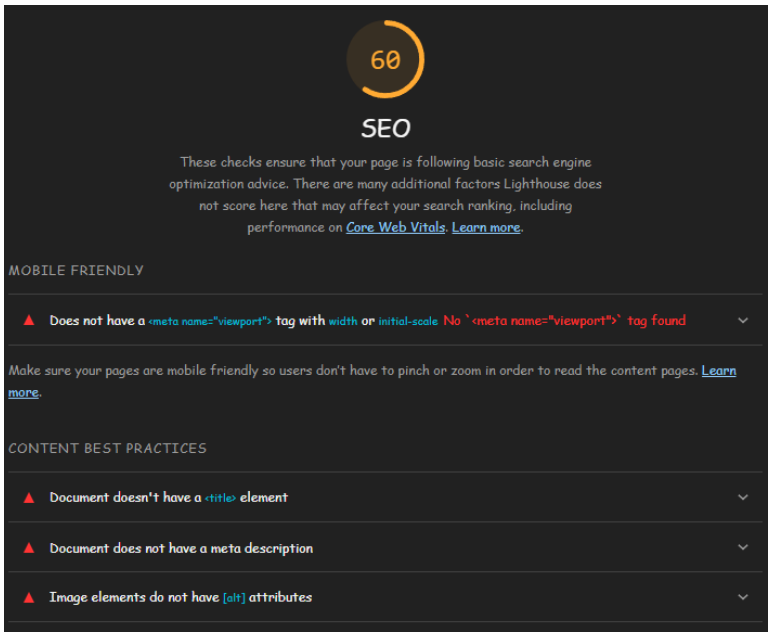
The accessibility of the site is currently lower than desired. The main cause for that is a lack of alternate texts for the images. This makes it harder for users of screen-readers to effectively use the site. Due to the nature of my simulation a large number of images are used, so it is important to fix the lack of alternate texts. The page also lacks a title, making it hard to quickly see what the purpose of the page is.



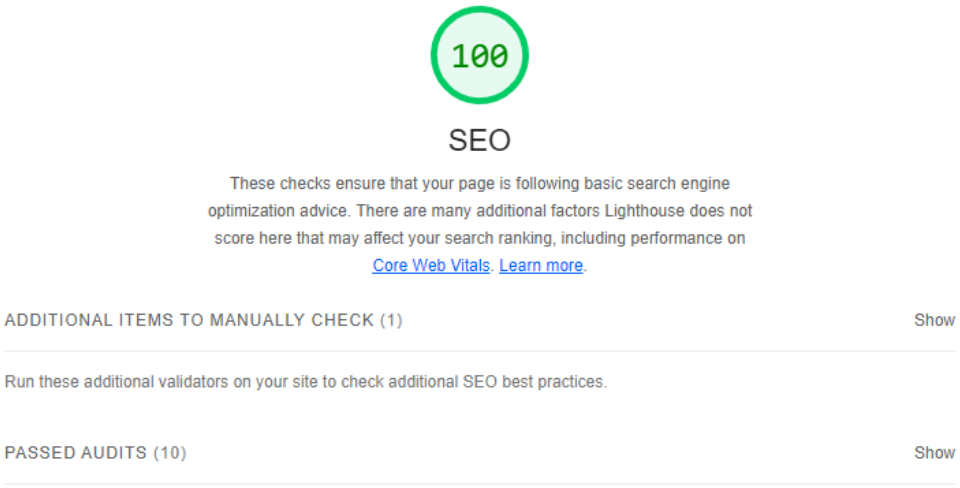
Having made the necessary alterations, adding titles and alternate texts, the site is extremely accessible, scoring a maximal rating of 100 with zero failed audits. This ensures that people with all levels of accessibility can make use of the site and the simulation. It is vital for a learning resource to be available to as many people as possible, so it was important to me to make sure that the sire was accessible.



When writing the simulation, I strived to use the best practices, this ensures that the code is readable and understandable, as well as being general and reusable. This has led to a maximal rating of 100 in best practices, with zero failed audits.



The site currently scores low on search engine optimisation (SEO). This is due to the lack of meta tags, which allow search engines to determine the type and purpose of the site. A key issue is the lack of a viewport definition, by adding a viewport meta tag and defining the width to the devices width large improvements can be made. A description meta tag is also added to allow search engines to know what the purpose of the page is; a logic circuit simulator.



With these changes made, site has a perfect 100 rating in the search engine optimisation, this ensures that the site is easy to find making sure that users who are looking for a logic circuit simulation can find the page. This opens up the site to a wide range of new users who may find the page through their own searching.

This testing is in tandem to a large amount of function specific testing that has occurred during the iteration. Each function that has been written has been test in isolation as well as its iterations with the rest of the code base. In any locations in which errors or unexpected functionality occur, this has been documented with an explanation of the error and what caused it, typically along with a screenshot of the erroneous result. This has then been fixed, with the fix being documented.

Many new features were developed during this iteration, many of which were unique and not related to any other areas of the simulation. Due to this, much feedback was gathered from the stakeholders. Interaction with them has been consistent throughout the iteration and their feedback has been implemented during development when it was received. This is slightly different to the previous iterations, where the majority of feedback was received at the end of development and added to the task list for the next iteration. This is because this will be the final iteration and so any unimplemented feedback will have to be seen as possible future development or part of perfective maintenance as time limitation prevent it from being implemented now. The only such feedback that I did receive for possible future development was the implementation of sound effects. This would be a possible sound connected to button presses which could give a tactile sense to the interface, as well as a possible sound for clock cycles, presenting the user an auditory way to sense the speed and synchronization of their clocks.

EVALUATION

<See H446-03 Project Advice Booklet for help and guidance of what must go here.>

PROJECT APPENDIXES

Insert as many project appendixes as you need for your project.

These might include, but are not limited to:

- Complete Code Listing (ESSENTIAL)
- Interview Transcripts
- Meeting notes
- Observation notes or questionnaires

BIBLIOGRAPHY NOT FORMATTED

- Page 9 image 1 https://logic.ly/demo/
- Page 9 image 2 https://circuitverse.org/simulator
- Page 10 https://academo.org/demos/logic-gate-simulator/
- Page 11 https://github.com/processing/p5.js/blob/main/contributor_docs/supported_browsers.md
- Page 35 https://www.allaboutcircuits.com/textbook/digital/chpt-7/converting-truth-tables-boolean-expressions/
- Page 89 https://www.techiedelight.com/inorder-tree-traversal-iterative-recursive/
- Page 113 https://www.debuggex.com/