Logic Gates Program

Contents

[Analysis 4](#_Toc101980585)

[Introduction 4](#_Toc101980586)

[Definition of Terms 5](#_Toc101980587)

[Current System 5](#_Toc101980588)

[Figure 1, Screenshot of Logisim 2.7.1 6](#_Toc101980589)

[Questionnaire 7](#_Toc101980590)

[Current Problems 8](#_Toc101980591)

[Prospective Users 8](#_Toc101980592)

[Data-Flow Diagram 10](#_Toc101980593)

[Figure 2, Data flow diagram 10](#_Toc101980594)

[Figure 3, An example logic circuit diagram 11](#_Toc101980595)

[Approximate Data Volumes 11](#_Toc101980596)

[Logic Gate 11](#_Toc101980597)

[Wire 11](#_Toc101980598)

[Pin 12](#_Toc101980599)

[Label 12](#_Toc101980600)

[Data Sources and Destinations 12](#_Toc101980601)

[Inputs 12](#_Toc101980602)

[Outputs 13](#_Toc101980603)

[Research 14](#_Toc101980604)

[Logic Gates 14](#_Toc101980605)

[Boolean Notation 14](#_Toc101980606)

[RPN 14](#_Toc101980607)

[Figure 4, Example use of the shunting yard algorithm 15](#_Toc101980608)

[Figure 5, Program demonstrating reverse polish notation 18](#_Toc101980609)

[Truth Tables 18](#_Toc101980610)

[Objectives 18](#_Toc101980611)

[Constraints 21](#_Toc101980612)

[Potential Solutions 21](#_Toc101980613)

[Brower/Web Server based application 21](#_Toc101980614)

[.NET framework 21](#_Toc101980615)

[WPF vs Windows Form 21](#_Toc101980616)

[Design 22](#_Toc101980617)

[User Interface Design 22](#_Toc101980618)

[Main GUI 22](#_Toc101980619)

[Figure 6, Design of main GUI 22](#_Toc101980620)

[Main GUI Inputs 23](#_Toc101980621)

[Boolean GUI 24](#_Toc101980622)

[Figure 7, Design of Boolean GUI 25](#_Toc101980623)

[Boolean GUI Inputs 26](#_Toc101980624)

[Truth Table GUI 27](#_Toc101980625)

[Figure 8, Design of Truth Table GUI 28](#_Toc101980626)

[Logic Gates Rules 28](#_Toc101980627)

[File Structure 29](#_Toc101980628)

[Classes 30](#_Toc101980629)

[Figure 9, UML Diagram 30](#_Toc101980630)

[Circuit 31](#_Toc101980631)

[Component 32](#_Toc101980632)

[Pin 33](#_Toc101980633)

[Gate 34](#_Toc101980634)

[Wire 35](#_Toc101980635)

[Expression 35](#_Toc101980636)

[TreeNode 36](#_Toc101980637)

[Test Strategy 37](#_Toc101980638)

[Algorithms 38](#_Toc101980639)

[Evaluate 38](#_Toc101980640)

[Create Expression 39](#_Toc101980641)

[Generate Table 42](#_Toc101980642)

[Create Circuit 44](#_Toc101980643)

[Generate Tree 46](#_Toc101980644)

[Exporting 47](#_Toc101980645)

[Saving 48](#_Toc101980646)

[Loading 48](#_Toc101980647)

[Testing 50](#_Toc101980648)

[Testing Outline 50](#_Toc101980649)

[Program Tests 51](#_Toc101980650)

[Figure 10, Test 1.11 61](#_Toc101980651)

[Figure 11, Test 1.14 62](#_Toc101980652)

[Figure 12, Test 2.4 62](#_Toc101980653)

[Figure 13, Test 3.1 63](#_Toc101980654)

[Figure 14, Test 3.2 63](#_Toc101980655)

[Figure 15, Test 4.3 64](#_Toc101980656)

[Figure 16, Test 4.4 64](#_Toc101980657)

[Figure 17, Test 4.5 64](#_Toc101980658)

[Figure 18, Test 4.8 65](#_Toc101980659)

[User Tests 65](#_Toc101980660)

[Questionnaire 65](#_Toc101980661)

[Reponses 67](#_Toc101980662)

[Code Listing 68](#_Toc101980663)

[XAML 68](#_Toc101980664)

[MainWindow 71](#_Toc101980665)

[Canvas Interactions 72](#_Toc101980666)

[Component Drawing 83](#_Toc101980667)

[ListBox Selection 90](#_Toc101980668)

[Truth Tables 91](#_Toc101980669)

[Boolean 94](#_Toc101980670)

[Canvas Movement 114](#_Toc101980671)

[File Interaction 117](#_Toc101980672)

[Program Logic Code 126](#_Toc101980673)

[Circuit Class 126](#_Toc101980674)

[Expression Class 133](#_Toc101980675)

[TreeNode Class 137](#_Toc101980676)

[Component Classes 138](#_Toc101980677)

[Evaluation 142](#_Toc101980678)

[Objectives 142](#_Toc101980679)

[Extensions 145](#_Toc101980680)

[Overall Comments 145](#_Toc101980681)

# Analysis

## Introduction

A common way of representing Boolean logic is through logic circuit diagrams. These diagrams show individual logic gates that compare on/off inputs to provide an on/off output connected by wires that can be in an on or off state. The circuit then takes input from specific wires and provides an output on a specific wire. This is a useful representation for teaching students the fundamentals of computer logic but can also be used at a more advanced level to represent complex circuits like the full adder which can be used to add together binary numbers. Truth tables are diagrams used to show how different input states create different output states. Every logic circuit has an associated truth table that can be generated algorithmically.

The Bishop’s Stortford High School (TBSHS) is a state school teaching computer science to students from ages 11 to 18. All students study CS in years 7 and 8. Some students drop the subject in year 9. By year 10, students have decided whether to take the subject at GCSE level. Sixth form (year 12 and 13) CS classes studying A-Levels tend to be small, sometimes with fewer than 10 students. The year in which students are first taught about Boolean logic varies but is generally before year 9. The Computer Science department at TBSHS has two teachers, Mr Mullen and Mr Atkinson. Both teach logic gates and Boolean algebra to younger and older students.

When teaching logic circuits to pupils, the school often makes use of a program called Logisim. Logisim allows users to create complex logic circuits consisting of a wide selection of logic gates, user-made wires and custom inputs and outputs. This gives students the ability to create their own logic circuits to aid in their learning of the visual system. The circuits are evaluated by the program in real time which helps to teach students the functions of different logic gates. Logisim also has the capability to generate truth tables of user-created circuits but it is not a particularly user-friendly system. Logisim has worked as a learning aid for some time now but has several key issues regarding user interface and long-term sustainability.

Another way logic is taught at TBSHS is through Boolean algebra. This is a formalised description of logical operations that can be used to represent the same logic circuit in a diagram. At TBSHS, this is taught in years 10 and 11 as well as a more advanced version in years 12 and 13. The exam board followed at GCSE is OCR whereas at A Level, AQA is used. This means that different symbols and standards are used for years 10 and 11 compared to years 12 and 13. It is possible to convert directly between logic circuit diagrams and Boolean algebra (and vice versa).

It would be useful to the school if a new program that did not have the same issues found in Logisim and had greater functionality with regards to Boolean and other course-relevant area could be created. Students would maintain the ability to create and learn from their own logic circuits but would also be able to create Boolean algebra expressions with the same purpose. Creating a synergy of Circuits, Boolean and Truth tables in a single program created bespoke for the school would go a long way to assist the students’ learning. Mr Atkinson is particularly enthusiastic about this program.

## Definition of Terms

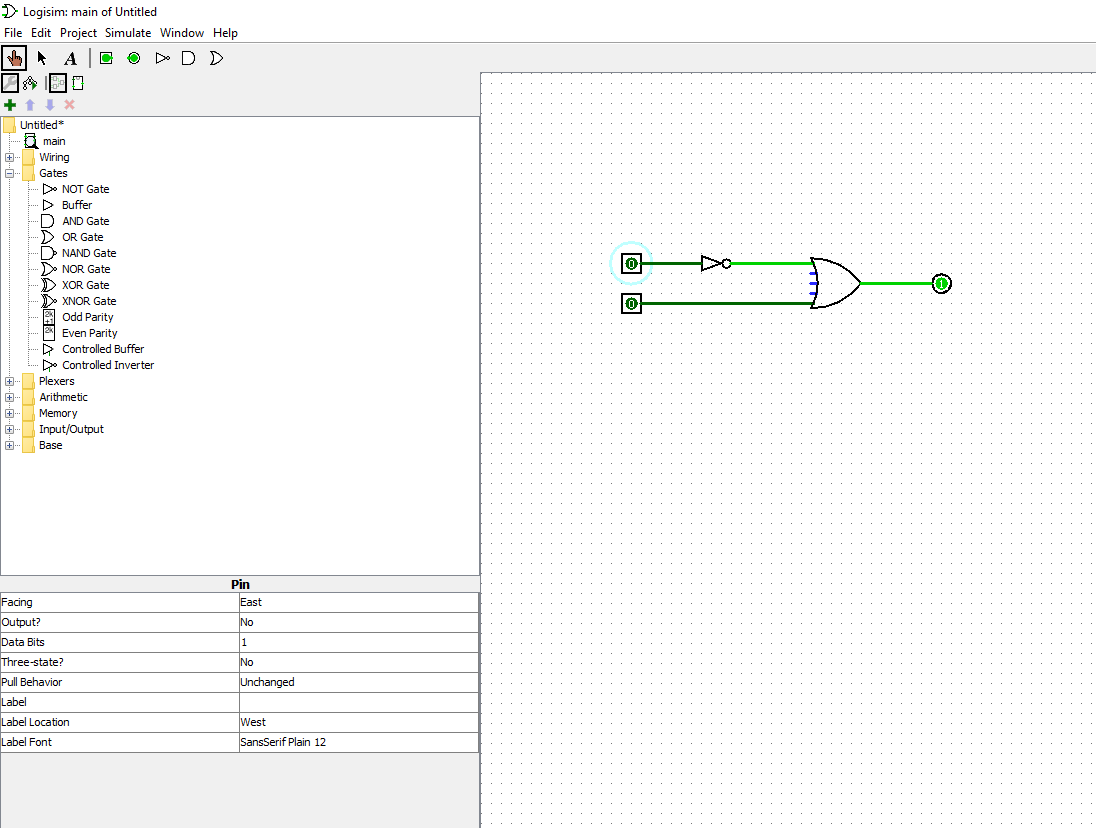
|  |  |
| --- | --- |
| Logic Circuit Diagram | A graphical representation of a Boolean logic circuit. |
| Boolean Algebra | An algebraic representation of a Boolean logic circuit. |
| Binary | A store with two potential states, generally on or off. This is the form taken by inputs and outputs of logic gates as well as the general state of a wire or pin. |
| GCSE | The exam system studied from age 14 to 16 at TBSHS. This includes a Computer Science course. |
| A Level | Properly, GCE Advanced Level. The exam system studied at TBSHS after GCSE from age 16 to 18. This includes a Computer Science course. |
| Computer Science | The school subject based on the study of computers. At GCSE, this can be called Computer Programming. At A Level it is called computer science. |
| Logisim | The program previously used at TBSHS for logic gate teaching. |
| Karnaugh map | A more abstract representation of a Boolean logic circuit involving highlighted areas of a grid. |
| Truth Table | A table showing every possible input to a logic circuit and the output resulting from each of them. |
| Logic Gate | A model for a physical electronic component used in logical circuitry. Generally compares on/off inputs to provide and on/off output. |
| Pin | The inputs and outputs users can add to each end of their circuit in order to change the states of the gates’ inputs and read their outputs. |
| Windows Forms | An older framework for .NET UI creation. |
| Reverse Polish Notation | A method of representing expressions using operators before operands rather than either side as in more common infix notation. |
| Student | Computer Science learners ranging from ages 11 to 18. |
| Teacher | Computer Science instructors at TBSHS, namely Mr Atkinson and Mr Mullen. |
| WPF | Windows Presentation Foundation. A newer framework for .NET UI creation. |
| Wire | An abstract connection between two logic gates. Can either be in an on or off state. |
| Boolean logic | The set of functions that compare binary on/off state inputs to provide binary on/off state outputs. |
| TBSHS | The Bishop’s Stortford High School |

## Current System

For years, TBSHS has used the program Logisim for teaching and representing logic circuits. Logisim’s website describes the program as:

‘***an educational tool for designing and simulating digital logic circuits. With its simple toolbar interface and simulation of circuits as you build them, it is simple enough to facilitate learning the most basic concepts related to logic circuits. With the capacity to build larger circuits from smaller subcircuits, and to draw bundles of wires with a single mouse drag, Logisim can be used (and is used) to design and simulate entire CPUs for educational purposes***’ (<http://www.cburch.com/logisim/>)

Figure 1, Screenshot of Logisim 2.7.1



The user selects logic gates from the panel on the left and places them in the grid on the right. They can then connect them with wires by dragging lines between gates and then add inputs and outputs. Then, using the hand tool from the top ribbon, they can set the state of an input to on or off. The circuit will react instantly, and the output can be observed. The program can also analyse the circuit that has been created, producing a Boolean algebra statement representing it and a truth table showing the outputs from every possible input.

These are the basic functions Logisim is used for at TBSHS, but the program is capable of far more. The program can hold RAM and ROM and maintain thousands of clock cycles. It can give output in an LED matrix or a 7-segment display. It can take input from a joystick or keyboard. The program has functionality to store user-created subcircuits and recreate their behaviour as a component in a larger circuit. These extra features are excellent for an advanced user who requires them, but TBSHS does not make use of any of them. Also, while Logisim can convert user-created circuit diagrams into Boolean algebra expressions and this feature is used for demonstrating the connection between the mediums, the program is not helpful for turning user-created Boolean algebra expressions into logic circuit diagrams limiting the usage of Logisim for teaching Boolean algebra.

Younger students are certainly not helped by the complicated nature of Logisim’s design. Of the 7 folders of gates and other items the program comes with, the school makes use of 1 and not even all the items in it. Students unfamiliar with computer science (years 7 and 8 mostly) can become confused between exporting and saving circuits they have created. None of these issues are likely to be fixed as development of Logisim was suspended in 2014, the program will therefore age and become less compatible as time goes on. This also raises the issue of compatibility as Logisim will may become unsupported as time goes on and the program’s dependencies advance without it. A simpler, sleeker project that is easier for younger students to use would be helpful at TBSHS for teaching logic and Boolean algebra.

## Questionnaire

A questionnaire was sent to TBSHS Sixth form students and teachers with the following questions:

1. How do you feel about your understanding of logic circuits?
   1. I don’t understand logic circuits at all
   2. I understand them somewhat
   3. I have no issue understanding logic circuits
2. What would you say are the benefits of using Logisim?
3. What would you say are the drawbacks of using Logisim?
4. If you wanted to help users to understand more advanced logic circuits (e.g. the full adder) what features do you think could make this easier?
5. Do you think circuits will be made that are larger than a monitor’s size?
6. What kind of simplification of Boolean algebra expressions would be most useful to you?
   1. Fully simplified
   2. Exactly representational of the logic circuit that has been created
7. In what school year were you first introduced to logic circuits?
8. Would you want logic circuits you create to be outputted in a usable format and if so what format would be most useful?
9. There are other ways to represent logical expression than Boolean and circuit diagrams. Would there be another way the program could represent them that would help you?

The responses to the questions were as follows:

1. Every respondent chose option C, that they had no issue in understanding logic circuits. This shows that students in years 12 and 13 do not have difficulty understanding the concepts behind logic circuits and Logisim’s interface issues are not a major problem for older students.
2. The responses were varied. Multiple students reported that Logisim’s features allowing analysis of circuits were helpful, specifically truth tables. Some students said the program helped with checking the validity of a logic circuit as Logisim highlights incompatible or incomplete sections of a circuit in red.
3. Most respondents reported the unintuitive layout of menus and general over-complication of the program as their main issues.
4. Some useful design ideas given. Pre-built snippets of circuits were a common theme. The ability to add a separate circuit as a single element with its own inputs and outputs was also mentioned.
5. Most respondents said yes.
6. Almost 50/50 responses.
7. The majority of respondents said years 7 and 8. A few respondents reported years 10 and 11.
8. Almost all respondents said an image format would be helpful. PNG was most common file format mentioned.
9. Truth tables were definitely the most wanted feature, Karnaugh maps were also mentioned though these are less important as they are not on either the GCSE or A Level courses.

The responses to these questions confirm several things and bring a few new ideas to the table. Question 1 shows that Logisim’s failings are not preventing students from understanding logic circuits thoroughly. This indicates that it is a good idea to stick close to Logisim’s implementation of the basic logic circuit creation (panel of possible gates, grid to place them on, mouse-created wires connecting gates, etc.). Question 2 adds to this, showing that the analysis features of Logisim are worth keeping in a new program. Truth tables and circuit validation will therefore be important parts of the design phase. The response to question 3 confirms the problems already understood about Logisim, namely that the program is too complex and designed for far more intensive use than is needed by TBSHS.

Question 4 provided some good ideas for the program’s design. Having the ability to create a separate sub-circuit and place it into a larger one as a single ‘gate’ is a feature from Logisim that can be handy when working with larger circuits, though it is not required for any of TBSHS’ students and so would be an extension of the program. The idea of having pre-built circuits also came up. This would be helpful for using circuits like the full adder where multiple have to be chained to add larger binary numbers. Most respondents thought it would be useful to have an infinitely sized canvas. This means the grid will have to store all objects not currently on the screen as well as having some method for traversing the extended canvas. The response to question 6 was almost 50/50. It would therefore make sense to implement both features, having the program display an exactly representative expression and a fully simplified expression. De Morgan’s laws could also be demonstrated here for older students.

PNG was the most requested image format and could be implemented easily. SVG could also be added as an extension. The response to question 9 was almost universally in favour of truth tables and so these will be a definite feature of the program. Karnaugh maps were brought up but are not in the scope of the prospective user groups and so will be an extension feature. Speaking to Mr Atkinson separately, I found that flip flops and buffers have been used in the past while using Logisim. These are not on any of the relevant courses but they are often helpful for creating logic circuits and so they could be included as an extension feature. As well as this, it was reported that often it is helpful to be able to label the diagram with text prior to it being exported. This is a good extension idea that will improve the program’s utility for teachers.

## Current Problems

Summary of the current issues with using Logisim:

* Logisim is far more powerful than TBSHS will ever need. A program more suited to the requirements of a school would be more applicable for teaching.
* Because Logisim is so powerful, it is highly complex and difficult for some users to use. A more streamlined approach would make a program far easier to use.
* Logisim cannot convert Boolean algebra statements into logic circuits easily, limiting the use of the program in teaching Boolean algebra. A program that can create circuits from Boolean algebra expressions and vice versa would be more useful to older students.
* Development of Logisim was suspended in 2014 and is unlikely to continue again. The program will continue to age and may eventually become incompatible with other releases. A newer program with a sleeker design will not only be easier to use but will last far longer before becoming obsolete.
* Younger students can often become confused between saving and exporting their circuit diagrams from Logisim.

## Prospective Users

The main group who will use the program will obviously be the students. However, this can range from 11-year-old year 7 students to 18-year-old year 13 students. A program will therefore need to be simple enough so as not to confuse younger students but not so simple that older students find the program incapable of providing the tool they need.

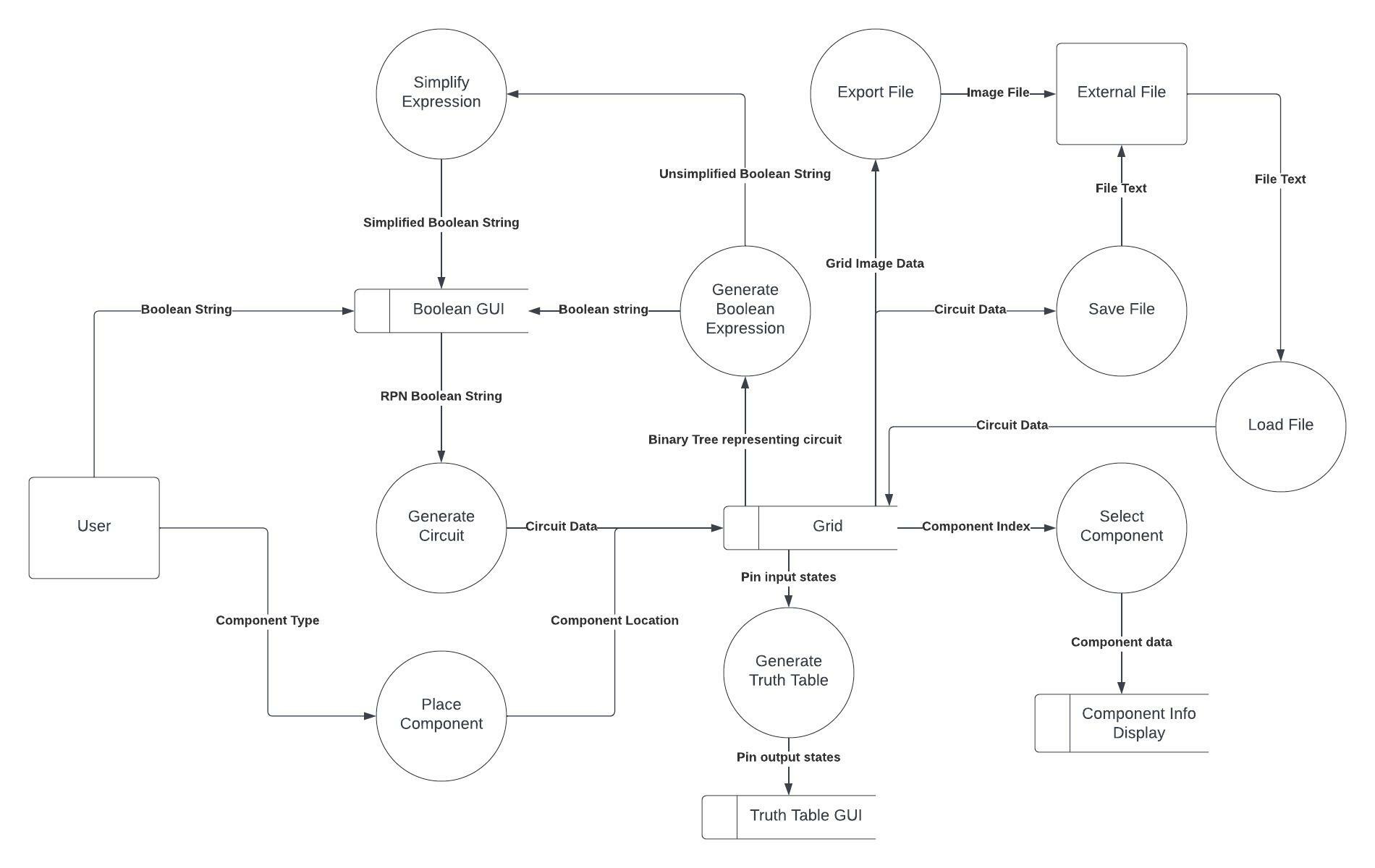
Younger students will not generally make use of Boolean algebra functions as this is usually taught higher up in the school. The symbols used in GCSE Boolean algebra are different to those used at A-Level.

The program will also be used by teachers for demonstration and potentially to create example circuit images.

* Young Students
  + Students in years 7-9
  + These students are assumed to be less technically able than older students. They should be able to interact with a well laid out interface but could become confused by over-complication and unnecessary features getting in the way of the functions they need.
  + These students will only need to create simple circuits that help them get to grips with logic circuits. They may use truth tables as these are fundamental aspects of logic circuits, but they should not need to use Boolean algebra features and so those features will not need to be designed with these users in mind.
  + Features that these users use should be readily available on loading up the program. If these users must look for something in a menu, the layout is flawed.
  + It may be prudent to include basic instructions with the program to assist these students in familiarising themselves with the program’s systems.
* GCSE Students
  + Students in years 10 and 11
  + These students are assumed to be technically able. They can look through menus to find the features they need.
  + These students need a program that will allow them to create simple circuits, the same as the younger students. Some respondents to the questionnaire said they had only been introduced to logic circuits at GCSE. Some of the circuits created by these students will therefore be their first and so will be quite simple.
  + GCSE students need basic Boolean algebra features using the GCSE Boolean symbols (Ʌ V ¬). These are different to those at A Level and so a setting to swap between the two will be needed.
  + Most GCSE students at TBSHS will have come through the younger years and so may already have familiarised themselves with the program before.
* A Level Students
  + Students in years 12 and 13
  + These students are assumed to be technically able and no specific effort should be made to make features used by these students more readily available than others.
  + The logic encountered at A Level is significantly more complex than that of GCSE, especially in its Boolean form. These students will need to create and be able to work with more advanced circuits than the other years.
  + The Boolean algebra encountered at A Level is also more advanced. It uses A Level symbols (. + ̅ ^). This second set of symbols will need to be available as well as the GCSE set. However, since the A Level students are generally more technically proficient, it would make sense for the GCSE symbols to be the default as older students can be trusted to be able to change the settings.
* Teachers
  + Teaching students of all ages
  + Teachers are assumed to have no issues with technical ability. They should be able to understand every feature of the program.
  + The program would be used for presentation and demonstration purposes.

## Data-Flow Diagram

Figure 2, Data flow diagram

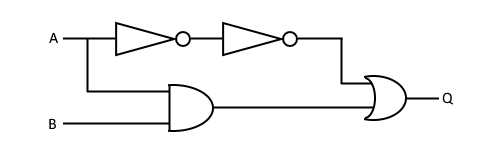


The two external data sources for the program will be the user and any external files. The user has two main inputs to the program: creating Boolean expressions and placing components on the grid. Placing components on the grid is the main way in which circuits will be created. The user will select the type of component and then place it. The location they enter will be passed to the grid which should then display the component at that location. The other input the user has it through creating Boolean strings. The user will use the Boolean GUI to create a string which can then be used to generate a circuit. The GUI will analyse the string in RPN form (explained in the research section) and then pass circuit data to the grid that can be used to display the intended circuit. While creating the circuit, the user may wish to examine a particular component more closely or change the state of a pin. To do this they will select the component on the grid. This will prompt the program to retrieve the relevant data from the grid and display it to the user on the component info display on the main GUI.

The other way input will be taken by the program is through file interaction. Users will be able to choose to save circuits they create to a file. Doing so will take the circuit’s data and create a file text that can be used to reload the circuit. This text will then be placed in a file. If the user wishes instead to export the file, the image data from the grid will be taken and an image file will be created containing that data. Once a circuit has been saved, the user will be able to reload it. This will reverse the saving process, taking the file text and returning the circuit data to the grid so the circuit can be displayed.

Once a circuit exists in the grid, several things could happen. If the user requests the program to generate a truth table, the grid will trial a series of input states and record the relevant output state. Then this data will be used to fill in the rows and columns of the truth table which will be displayed in the truth table GUI. Alternatively, the user may request a Boolean expression be created to describe the circuit. In this case, a binary tree will be created the represents the circuit, this tree can then be traversed to get a string representing the circuit. Initially the expression may not be simplified. The program will take the string and create a simplification, displaying both versions to the user in the Boolean GUI. The first version will be the direct output of the analysis, unaltered by the program.

Figure 3, An example logic circuit diagram



This circuit would first be displayed as but it can be simplified further. The next level of simplification would cancel out redundant NOTs. In this example, The two NOT gates at the top of the circuit cancel each other out. This circuit could then be simplified to just be . Further simplifications are possible though. In this circuit, the output cannot be true unless A is true and B has no effect. Therefore the simplest form of this circuit is just .

## Approximate Data Volumes

Logic Gate

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Explanation** | **Type** | **Approximate Size** |
| Input States | Every gate will have 2 inputs that can be in an on or off state. This will be 2 variables, one for each input. | Boolean | 2 Bits |
| Output State | The gate will have an output value that can be in an on or off state. The state of this will be determined by the program. | Boolean | 1 Bit |
| Position | The gate will have a position on the grid that will take 2 coordinate values. This can be stored in an array of length 2. | Array(Byte) | 2 Bytes |

Total: 27 Bits

Wire

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Explanation** | **Type** | **Approximate Size** |
| Vertexes | The set of coordinates on the grid that the wire will pass through. The start and end will be determined by the user, the path will be determined by the program. This will be a 2D array of coordinates. | 2DArray(Byte) | 2 Bytes \* Number of points in path |
| State | The wire can be in an on or off state. | Boolean | 1 Bit |

Total: 1 + (16 \* Number of points in path) Bits

Pin

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Explanation** | **Type** | **Approximate Size** |
| Position | The pin will have a position on the grid stored as a coordinate in an array of length 2. | Array(Byte) | 2 Bytes |
| State | The pin will have a state that can be either on or off. | Boolean | 1 Bit |
| Input/output | The pin will have to be either an input or an output pin. This will determine whether it takes an input from the left or give an output to the right and whether the user can set the state of the pin. | Boolean | 1 Bit |
| Name | For purposes of Boolean algebra, the pin will have a letter assigned to it. This will probably just be a char of ‘A’ for example. | Char | 2 Bytes |

Total: 34 Bits

Label

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Explanation** | **Type** | **Approximate Size** |
| Position | A label will have a position on the grid where the text will be shown. This will be a coordinate in an array. | Array(Byte) | 2 Bytes |
| Text | The label will have text that it will display on the screen. | String | 2\*(Number of characters) Bytes |

Total: 32 + (2 \* Number of characters) Bits

## Data Sources and Destinations

Inputs

|  |  |  |  |
| --- | --- | --- | --- |
| **INPUT** | **SOURCE** | **FORMAT** | **EXPLANATION** |
| Logic Gate Type | User | Integer | When the user goes to place a new logic circuit, the type they pick will correspond to the class governing the behaviour of that kind of gate. |
| Logic Gate Location | User | Integer Coordinate | Every gate/pin that is placed on the grid will have a location stored as a co-ordinate. From the user’s perspective, they will simply drop the gate somewhere on the grid. |
| Pin State | User | Boolean | Each input pin will automatically be in the off state. Users can change the state of this pin to on so that they can give input to their circuit. |
| Gate output | Program | Boolean | Every logic gate will have an output state associated with its current inputs. This will be output to later gates or pins for their use. |
| Saved Circuit | File | Unique File Format | Circuits can be saved from the program to a unique file format that the program will be able to load. |
| Wire Locations | User | Integer Coordinate | Each wire the user places will have a starting co-ordinate and an ending co-ordinate. These could exist on the main grid or in a sub-circuit. The path the wire takes will be determined by the program. |
| Boolean Input String | User | String | The user will use the ‘keyboard’ in the Boolean GUI to enter a Boolean expression that the program will store as a string. |

Outputs

|  |  |  |  |
| --- | --- | --- | --- |
| **OUTPUT** | **DESTINATION** | **FORMAT** | **EXPLANATION** |
| Circuit Image | User-Specified file location | Image Format (PNG) | User-created circuits will be exportable to an image format in a user-specified file location. |
| Truth table image | User-Specified file location | Image Format (PNG) | User-created truth tables will be exportable to an image format in a user-specified file location. |
| Saved Circuit | User-Specified file location | Unique File Format | When a user is finished with their circuit, they will be able to save it to a chosen file location. The file format will be unique to the program and will store the circuit in a way that can be read by the program later. |
| Boolean Output String | Boolean Menu | String/Image | The program will be able to convert the circuit to a Boolean expression representation. Several symbols used in Boolean algebra are not common to strings (e.g. bar for not) and so the output of the program may have to use some sort of image as output rather than a string. |
| Truth Table | Truth Table GUI | Image Format | The program will not exactly store any truth tables. They will be generated as images and displayed in the truth table GUI. Each box in the table will contain a 1 or 0 generated by the program analysing the user’s circuit. |

## Research

Logic Gates

There are 3 main logic gates the program will need to use. These are the AND, OR and NOT gates. AND takes 2 inputs and provides an on output if both inputs are on. OR takes 2 inputs and provides and on output if either of the inputs are on. NOT takes 1 input and reverses it. If the input is on, the output is off and vice versa.

There are other gates that the program could also use. These would be NOR, XOR, NAND and XNOR. NOR is like NOT but with multiple inputs. NOR will output on unless any of its inputs are on in which case it outputs off. XOR is exclusive OR. If either of its outputs are on, it outputs on. However, if both outputs are on it outputs off. NAND will output on with any input except if both of its inputs are on, in which case it outputs off. XNOR is like NOR except it will output on if both its inputs are on.

All logic gates can be decomposed into a circuit using only NAND gates or NOR gates. This fact isn’t relevant to the GCSE or A Level course, but a nice extension feature would be to have the program decompile any circuit into NANDs or NORs.

Boolean Notation

There are two systems of notation for Boolean algebra that this program needs to support. There are the symbols used at GCSE and the symbols used at A Level.

At GCSE the symbols are as follows: the AND gate is represented by **ꓥ**, the OR gate is represented by **ꓦ**, negation is represented by **¬**. At A level, the symbols used are as follows: the AND gate is represented by a dot (**.**) between two variables (e.g. A.B), the OR gate is represented by a **+**, the NOT gate is represented by a bar over the negated values (e.g. ). The specific notations specified by each course do not include a symbol for the XOR, but there are symbols used in practise. A common one is ^ and this is what the program should use.

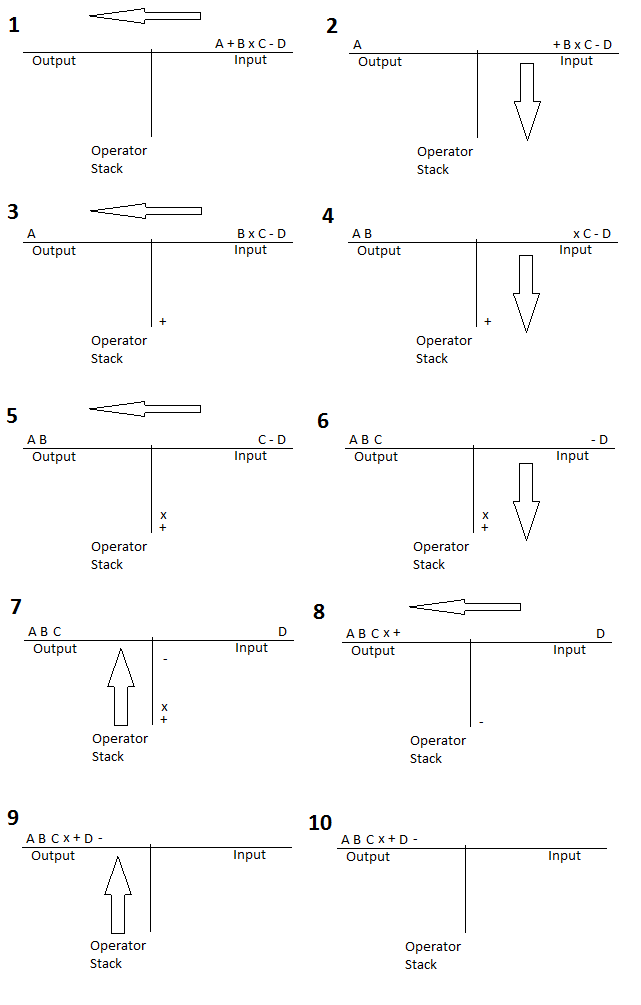
Since there is no way to represent a bar over a character in a string, the program will need a way to handle storing NOT gates.

RPN

Boolean uses an order of operations with NOT at the top followed by AND then OR. This can make parsing a Boolean algebra expression difficult as often brackets are required for the true logic of the circuit to be expressed. It would be helpful to work with Boolean expressions in reverse polish notation (RPN). In this form, operators are placed after operands and no brackets are required.

Expressions can be converted to RPN using the shunting yard algorithm. In this algorithm, input is read left to right and operators are placed on an operator stack while operands are placed into the output. The operators are then placed into the output and the expression is outputted in RPN.

Figure 4, Example use of the shunting yard algorithm



To test the algorithm, I made a program that converted arithmetic expressions to RPN:

Queue<string> input = new Queue<string>();

Stack<string> operatorStack = new Stack<string>();

for (int i = 0; i < labInfix.Text.Length; i++)

{

input.Enqueue(Convert.ToString(labInfix.Text[i]));

}

string nums = "1234567890";

string ops = "+-\*/^";

string output = "";

while(input.Count != 0)

{

if (nums.Contains(input.Peek()))

{

output += input.Dequeue();

}

else if (ops.Contains(input.Peek()))

{

if (input.Peek() == "+" || input.Peek() == "-")

{

while (operatorStack.Count != 0 && operatorStack.Peek() == "\*" || operatorStack.Count != 0 && operatorStack.Peek() == "/" || operatorStack.Count != 0 && operatorStack.Peek() == "^")

{

output += " ";

output += operatorStack.Pop();

}

}

else if (input.Peek() == "\*" || input.Peek() == "/")

{

while (operatorStack.Count != 0 && operatorStack.Peek() == "^")

{

output += " ";

output += operatorStack.Pop();

}

}

output += " ";

operatorStack.Push(input.Dequeue());

}

else if (input.Peek() == "(")

{

operatorStack.Push(input.Dequeue());

}

else if (input.Peek() == ")")

{

while(operatorStack.Count != 0 && operatorStack.Peek() != "(")

{

output += " ";

output += operatorStack.Pop();

}

if (operatorStack.Count > 0)

{

operatorStack.Pop();

}

else if(operatorStack.Count == 0)

{

output = "ERROR";

break;

}

input.Dequeue();

}

}

while(operatorStack.Count != 0)

{

output += " ";

output += operatorStack.Pop();

}

labRPN.Text = output;

Though not commented, this code performs the shunting yard algorithm on an expression entered by the user which is stored in the input queue. This next algorithm evaluates the RPN expression:

string ops = "+-/\*^";

string[] tokenList;

Stack<string> operandStack = new Stack<string>();

double operand1;

double operand2;

tokenList = labRPN.Text.Split(' ');

for (int i = 0; i < tokenList.Length; i++)

{

if (ops.Contains(tokenList[i]))

{

operand1 = Convert.ToDouble(operandStack.Pop());

operand2 = Convert.ToDouble(operandStack.Pop());

if (tokenList[i] == "+")

{

operandStack.Push(Convert.ToString(operand1 + operand2));

}

else if (tokenList[i] == "-")

{

operandStack.Push(Convert.ToString(operand2 - operand1));

}

else if (tokenList[i] == "\*")

{

operandStack.Push(Convert.ToString(operand1 \* operand2));

}

else if (tokenList[i] == "/")

{

operandStack.Push(Convert.ToString(operand2 / operand1));

}

else if (tokenList[i] == "^")

{

int total = 1;

for(int j = 0; j<(int)operand1; j++)

{

total = total \* (int)operand2;

}

operandStack.Push(Convert.ToString(total));

}

}

else

{

operandStack.Push(tokenList[i]);

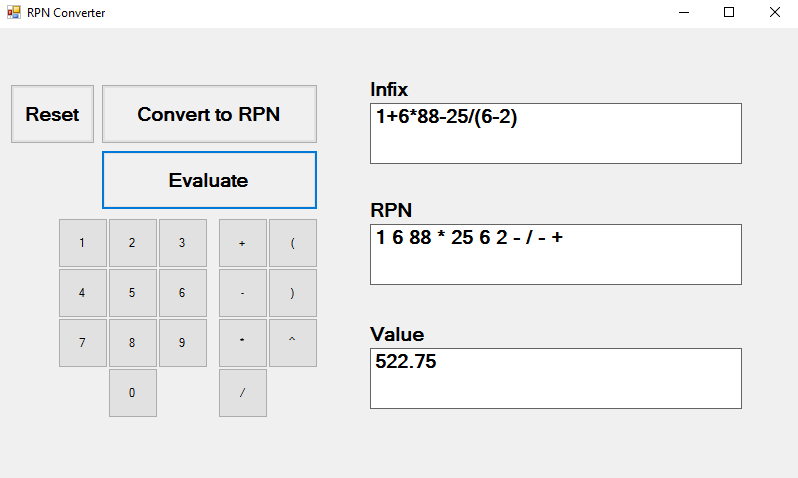
}

}

labValue.Text = operandStack.Pop();

Evaluating the RPN is very easy as each operator can be applied to the two operands immediately before it. This algorithm is perfect for converting arithmetic expressions to and from RPN but it will be inadequate for the Boolean expressions that will be generated by the program. Arithmetic expressions can be evaluated to produce a ‘composite’ result (i.e. 1+1 = 2 and 2 can then be used in another expression). Boolean values cannot be composed in this way which will mean evaluating the expressions will be harder and require more thought.

Figure 5, Program demonstrating reverse polish notation



Truth Tables

There are no particularly efficient algorithms for creating truth tables. Generally, the most effective method is to just evaluate the output for every possible input and create a table from those results. One issue truth tables could pose is that they become unreasonably large as the number of inputs increases. The number of combinations in a truth table is equivalent to 2n where n is the number of inputs. A circuit with 12 inputs will have 212 combinations of inputs or 4096 combinations. To represent this, the program would need to generate a table with 13 columns and 4097 rows. This is obviously not practical and so the program will need to impose a limit on how large a truth table it can produce.

Truth table for the 3 main gates:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **B** | **A AND B** | **A OR B** | **NOT A** |
| **F** | **F** | **F** | **F** | **T** |
| **T** | **F** | **F** | **T** | **F** |
| **F** | **T** | **F** | **T** | **T** |
| **T** | **T** | **T** | **T** | **F** |

## Objectives

The final program should meet the following criteria:

1. **Users must be able to easily create logic circuit diagrams that satisfy the requirements of GCSE and A Level course.**

This is the primary purpose of the program. These circuits should make use of AND, OR, NOT, XOR, XNOR, NOR and NAND gates with one or two inputs and one output per gate. Users should be able to add as many inputs and outputs as they want to the circuit. Users should be able to place the wires that will connect the circuit together. The logic circuit creation has the potential to be clunky and unintuitive which must be avoided to satisfy objective 6. To make the circuit creation easier, quality of life features like ghost images under the mouse and copy-pasting of sections of a circuit could be implemented.

1. **Users must be able to receive useful analysis of any logic circuit they create in the program.**

Users will create their circuit, then they will have a choice of analysis. The primary form of analysis will simply be the user turning pins on and off to test the output of their circuit. For more advanced users, the main forms of deeper analysis will be Boolean algebra expressions and truth tables. It is also possible that Karnaugh maps could be implemented as these are also a helpful way to visualise logic. The Boolean expressions need to be shown in both GCSE notation and A Level notation. Many questionnaire respondents also expressed a desire for seeing different forms of simplification from the Boolean analysis which could be quite easily implemented to the program. Truth tables can become very large very quickly as every new input requires its own column and creates many new possible combinations that need to be displayed. A way to make these more manageable is needed.

1. **The program must have multiple ways to create logic circuits.**

Most users will create their circuits using the basic grid system. However, sixth form students would benefit from the ability to enter Boolean expressions. Users should therefore have the ability to enter any valid Boolean expression in either GCSE or A Level notation and have the program create a logic circuit diagram based on their expression. This will require an input menu or some kind of keyboard system as symbols in both notations are not common to keyboards. Intertwined with objective 4, users should be able to load other people’s circuits into their own instance of the program.

1. **The program must enable users to view and save their circuits in a useful formats.**

Users will certainly want to save their circuits to work on them later. This is easily achievable through a simple saving and loading system. It was also seen in the questionnaire that students want to be able to export their circuit in an image format. PNG would be the most useful format for this but others like SVG are also possible implementations. Younger users reportedly got confused in Logisim between saving and exporting their circuits. This ties in to objective 5.

1. **The program must be simple to use.**

One of the main issues with Logisim that this program is trying to overcome is the over complication of menus and the difficulty younger users had with UIs. Features like the list of logic gates need to be presented up front to the user. Younger users shouldn’t have to open a single menu to use the program. Functions need to be labelled clearly and descriptively. The different functions could also be organised into separate sections. Separating analysis menus will help to clear up confusion between different forms of analysis however, the menus should still be accessed from the same area of the program. The program should feel familiar and the locations of different functions needs to be obvious. A ribbon of labelled functions at the top of the window would be useful for this as a tried and tested system of creating intuitive UI. There are other ways to do this though. Once the program is complete, giving it to students to test should demonstrate whether or not they’re able to use the program efficiently.

1. **The program must feel enjoyable and intuitive to use.**

This program has the potential to be a real mess of UI. If the user experience is not properly implemented, users could become confused or misunderstand functionality. Quality of life needs to be a focus of the design, especially for younger users (objective 7). For example, when a logic gate is selected from the gate list, the program should highlight it somehow to demonstrate that it has been selected. Features like these are important as they make the program easier to use. The more feedback the program gives the user, the better the user’s understanding of the program’s functions will be. There are also features that can improve a user’s enjoyment of the program. It can be frustrating when users make mistakes in their circuit creation so making deletion of unwanted components quick and easy will be important. Questionnaires can be taken after students have tested the program asking about their experience using it. Whether or not the program is intuitive should be shown here.

1. **The program must be useful to both younger and older students as well as teachers.**

The program’s intended user base includes students from year 7 to year 13. Neither younger nor older students should be prioritised in the development. The program’s basic features need to be simple enough for younger users to use without becoming confused but not so simple that older users find them restrictive. Older students are going to require far greater functionality than younger students. For example, large parts of the analysis functionality are not going to be useful for younger students as they do not need to know anything about Boolean algebra for their curriculum. Similar to objective 6, a questionnaire will be taken to decide whether the students find the program helpful in their study. Teachers have a different set of functions required from a program like this. They would largely use it for demonstration to students and so the program needs to be clear in its display of circuits so that it can be understood in a presentation format. Adding features like the ability to place text on a diagram prior to export would improve teachers’ ability to use the program for demonstration.

1. **The program should complete its functions in a reasonable time.**

Functions should output results in negligible time after being called. Users should not be able to notice a pause while the program carries out an operation. For some functions, a slowdown may be unavoidable when larger circuits are created. For example, a truth table for a program with 8 pins will be very large and could take time to create but generally these functions should be very quick. Testing this should as simple as timing the program carrying out different functions.

The final program could be extended to meet the following criteria:

1. **The program could include features to improve the speed at which users can create accurate circuits.**

Logisim has a useful function to highlight invalid sections of a circuit in red. Users would be able to see easily which parts of their program are preventing validation rather than having to find them themselves. This would be a helpful form of active analysis but is not strictly necessary for the program to function. Another feature would be including a bank of commonly used circuits that could be loaded into the main circuit. This would help users creating larger circuits as they could place sections at a time rather than individual components. The main focus of the program is going to be on users creating their own circuits from scratch and this is therefore an extension feature.

1. **The program could include features to assist students in their understanding of the fundamentals of logic circuits.**

All logic gates can be decompiled into either NOR or NAND gates. If the program could do this automatically, it would allow the user to see the fundamental structure of their circuit and could help in teaching of the ideas behind logic circuits. Additionally, other components like latches or flip flops could be included to expand students understanding of computer circuity. Neither of these features are required for the specifications of GCSEs and A Levels so they are extension features.

1. **For the most advanced users, the program could remove limits on circuit sizes.**

This is not a major issue for students as circuits are rarely large enough to escape the screen, but Logisim’s canvas is somewhat limited in size. Circuit can extend off the canvas but it is difficult to navigate beyond them and so it is hard to make circuits larger than the canvas. This could cause a problem for an advanced user with a very large circuit. The new program could have a functionally infinite canvas so that users never run out of space for circuit creation. This would require some way of storing the components not currently on the screen as well as a way of navigating to these components, both of which would require pre-planning during design.

1. **The program could include sub-circuit functionality**.

Another feature of Logisim that is not relevant to students’ courses but is nonetheless useful is the ability to create sub-circuits and add them to another circuit. This would again need to be pre-planned in the design phase but the idea of saving a sub-circuit and reusing it multiple times would make creating large circuits much faster and easier.

## Constraints

There are several functions of logic circuits that this program does not intend to feature. For example, by default, Logisim uses gates with 5 inputs. Even at A-Level, logic gates will not need more than 2 inputs. This is not relevant to any of the courses used at TBSHS and so will not be featured in the new program. As well as this, Logisim has the ability to load external libraries to extend its features. This will not be featured in the new program as all relevant tools should be already included.

Programs with large numbers of inputs can create enormous truth tables. It would be prudent to limit the size of truth tables. Logisim refuses to analyse circuits with more than a certain number of inputs so the new program should draw a similar line after a particular number of input pins are added to a circuit. There is also the question of the Boolean input functions. It will be difficult to account for all the different ways a circuit can be generated. A limit may need to be imposed to prevent circuits of too large a size from being created.

## Potential Solutions

Brower/Web Server based application

One potential solution for the implementation of this program would be a browser-based application using a language like JavaScript. This would be easily accessible to students through either the Google Chrome or Internet Explorer web browsers. Making a system entirely cloud-based would be a poor idea though, as communication between the user and the program would be easier over a local network. This solution would easily support more operating systems than just Windows but since the school uses windows almost exclusively this would not be a massive benefit.

.NET framework

Originally developed in the 1990s, the .NET framework is a Microsoft platform integrated into Visual Studio. It received its final update in 2019 and is still properly supported by Windows. .NET is unlikely to lose compatibility any time soon and so will provide the futureproofing required by this program. The language I would write this in would be C#. This has several useful built-in libraries and integrations with Visual Studio. C# is specifically suited to object-oriented programs making it apt for this task.

I will use this solution for creating my program as it provides the best combination of language and integration for a rock-solid base to build the program on. This solution will be compatible for the foreseeable future, solving one of the key issues of Logisim.

WPF vs Windows Form

There are two main options in the .NET framework for creating graphical interfaces, Windows forms and WPF.

Microsoft’s Windows forms are the solution I’m most familiar with. Forms provide a framework for creating desktop Windows applications. The visual features of forms are provided by the Graphics Device Interface (GDI+). This component is good for creating static images, but animation can be clunky and certainly not as smooth as I’d like for this program.

One of the important parts of this program is that it needs to have compatibility for the foreseeable future. Windows forms should be compatible with Windows for a significant time though Microsoft have said they are no longer improving the framework.

Windows Presentation Foundation (WPF) is a newer system than forms. It’s also one I’m less familiar with. WPF has far greater support for animation and smooth UI design as well as support for styles to give a program a consistent feel. Documentation is a little scarce, but WPF makes use of XAML for directly describing elements in a UI which allows precise control of shape and form of the program. XAML is a well-documented format.

I have decided I will use WPF for my UI. Even if I’m less familiar with it, its functions allow a greater degree of control over the interface and will create a superior user experience, which is important for creating the desired feel of the program expressed in my objectives.

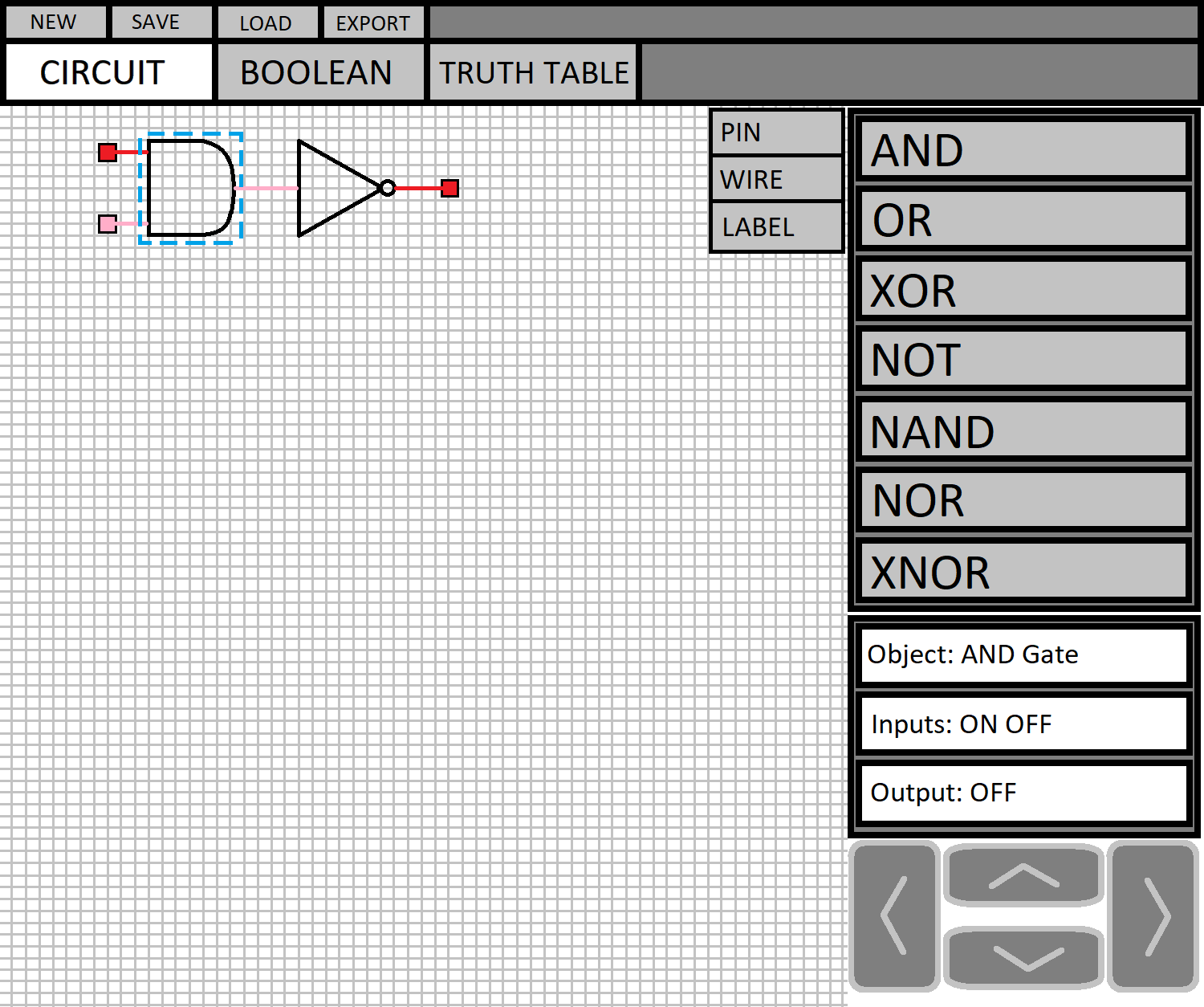
# Design

## User Interface Design

Main GUI

The screen the user will see on the program’s start and will spend most of their time interacting with will be the main GUI which will include the most fundamental features such as the list of logic gates that can be used, the grid they can be placed on and access to the other areas of the program. This is the most important interface as the user will spend so much of their time interfacing with it. The interface needs to be intuitive to use so that the user can quickly become proficient with it but not so simple that it prohibits more advanced users from using it in the way they wish.

Figure 6, Design of main GUI



This is a mock-up of the main GUI for the program. This is what users should see when they load up the program initially. Moving clockwise from the top left:

* The ‘new’ button will reset the canvas and allow the user to create a new circuit
* The program has buttons to save, load and export the circuit that the user has created. These should each open a separate dialogue box that will take relevant input from the user like a file location or format and then will complete the action associated with the button, i.e. save, load or export the circuit diagram.
* Below this are three buttons. The currently highlighted one is ‘Circuit’. This is the screen that is currently on show to the user, featuring the tools for creating circuits. Next to this button is the Boolean button. This will open the menu that allow the user to input Boolean which will be turned into a circuit or have the program output the Boolean expression that represents the circuit they have created. The third button labelled truth table will generate a truth table for the user’s created circuit. This should change the GUI to shows a truth table rather than the circuit.
* The grid itself is the largest part of the interface and rightly so as it will be the most important part of the user’s process when creating the circuit. The grid in this diagram is a little bold and I would prefer to make it a little softer in the actual program with a lighter shade for the grid lines. On the grid is a small example circuit with two input pins, an AND gate, a NOT gate and an output pin. The user has turned one input pin on, activating the top wire to the AND gate. Since the second pin is not on though, the gate itself still outputs off and the NOT gate inverts the signal to produce an on signal at the output pin. The user has selected the AND gate and it has been highlighted with a dashed blue ring. This is not a final decision on how to indicate which component is selected but it works for this diagram. Information regarding this selected AND gate is shown in a screen in the bottom right of the screen which will be explained in a moment.
* In the top right corner of the grid, there are three buttons for components of a circuit: pins, wires and labels. The user will click the button to select which component to place and then click on a location on the grid to place the component they have selected. In the case of the pin this will simply place a pin on the grid at the user’s selected point. Wires will require several locations to be selected after which the program will create a route through the diagram for the wire to follow. Labels are less components and more ornaments to the diagram. The user will select a position and then be able to specify the text they wish to place at the location.
* On the right of the program is the list of logic gates available to the user. This will contain all the logic gates the program can simulate in a list. They should be able to simply click a gate in the list and click on the grid to place it. When the user is mousing over the grid trying to place a gate, it would be helpful for a ghost image of the gate to appear so they can get a good idea of where the gate will be placed.
* In the bottom right of the screen is the box describing the characteristics of the currently selected object gate. It shows what the object is, what the state of its inputs are and what the state of its outputs are. In this instance an AND gate is selected with one input active and an inactive output. If the user wished to change the state of an input pin, they would be able to select it and then set its state using this box. A distinction will have to be made between input and output pins though, as output pins will not require their state to be manually changed by the user.
* In the bottom right corner of the screen are a set of buttons allowing traversal of the grid. The program will store objects that exist beyond the boundary of the current view and the user will be able to scroll to them by pressing these buttons.

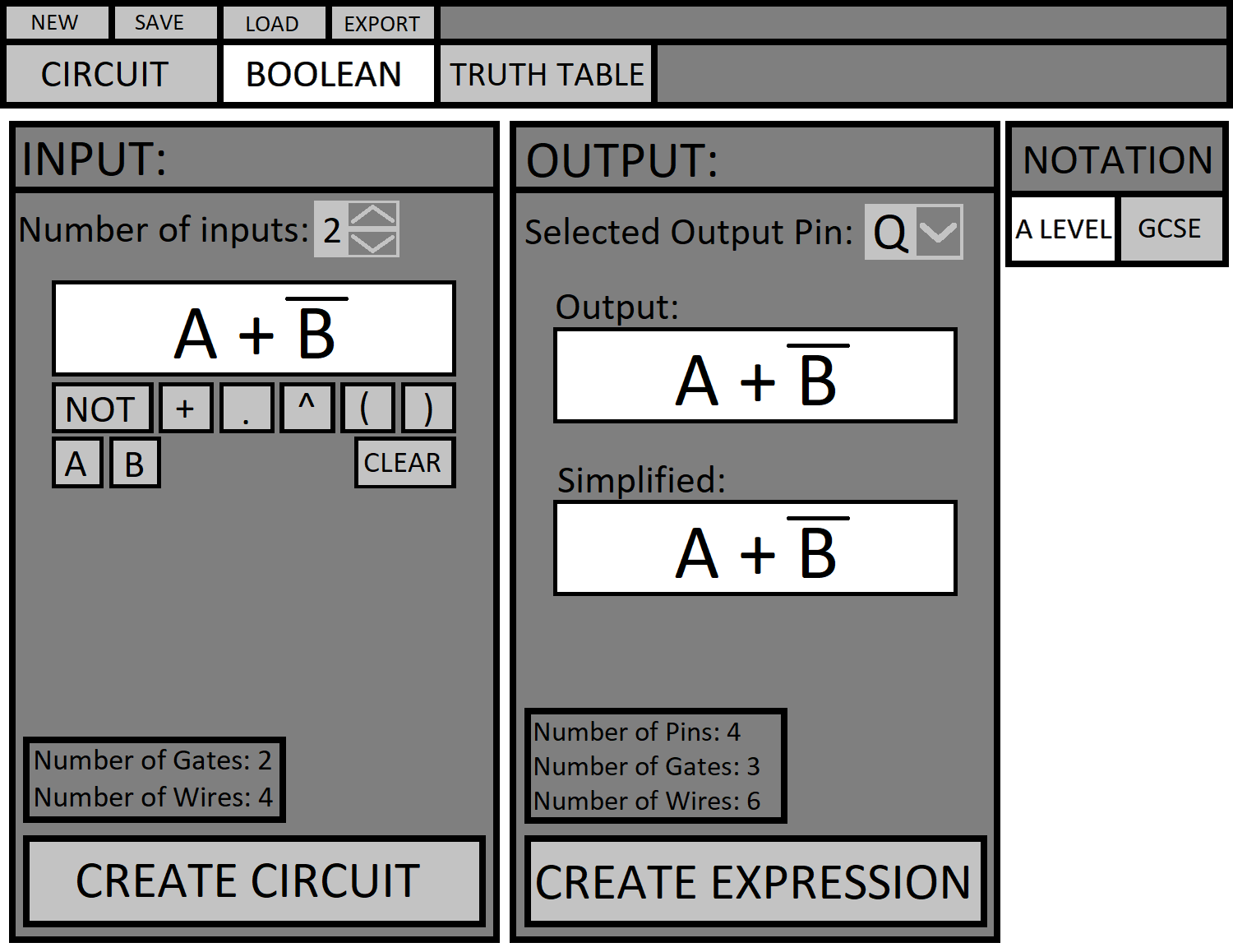
Main GUI Inputs

|  |  |
| --- | --- |
| **Input** | **Description** |
| Gate Placement | The user will select a logic gate and place it on the grid. This will be repeated many times throughout the program’s use. |
| Pin Placement | The user will select the pin object and place it on the grid. This will likely be the first thing they do to start their circuit. The user will also select whether this pin is an input or output pin. |
| Wire Placement | The user will select the wire object and give the start and end points on the grid that the wire will run between as well as any turn points. |
| Label Placement | The user will select the label object and place it on the grid. Then they can enter the text they wish it to display. |
| Pin state | The user will be able to select an input pin and set its state to on or off. |
| Secondary GUI | Pressing any of the other buttons at the top of the screen will take the user to a different GUI for the other functions of the program, Boolean and truth tables. |
| Grid movement | Using the arrows on the GUI, the user can move the grid to view a separate part of the circuit. This will require the redrawing of the grid multiple times as every time an arrow is pressed, the area of the grid being viewed will change. |
| Select Object | If an object is on the grid, the user can select it and be shown information regarding it. Which object is selected will be shown visually and information will be displayed in the bottom right box. |
| Save File | The user can press the save file button to create a file representing their circuit. This will require the program to convert the circuit as it exists in the program into a representation in a file. |
| Load File | The user can select a file to load into the program. This will require the program to loop through the file to extract the information and then turn that information into a circuit that can be displayed. |
| Export Circuit | The user can choose to export their circuit to an image format by pressing the ‘Export’ button. This will require the program to create an image of the circuit that has been created and then save it. |
| Preferences | The user can select the preferences button which will give them the option to swap between GCSE and A Level Boolean notation as well as changing other aspects of the program. |
| Analysis | The user can select a form of analysis by pressing the Boolean or Truth Table buttons. These will swap to a different GUI that allows the user to receive useful analysis of their program. |

Boolean GUI

The GUI used for Boolean algebra will be used almost exclusively by GCSE and A Level students as younger years do not need to understand the concept. This means that the interface can be made to be more complex as its predominant users should be competent.

Figure 7, Design of Boolean GUI



This mock-up shows the two parts of the Boolean GUI, the input functions and the output functions. The Boolean button at the top of the screen has been highlighted white after the user clicked it to show that the program is in the Boolean menu. Clicking the buttons either side of it would remove the highlight and set the program to show a different GUI.

The input menu has several features the user can interact with:

* At the top left of the menu is a counter that allows the user to select the number of inputs their desired expression will have. In this case it has been set to 2. The number here will determine the number of variables in the ‘keyboard’ of Boolean function lower on the menu. Since 2 inputs are desired, A and B are present in the keyboard. If the user increased the number of inputs to 4, the keyboard would contain A, B, C and D.
* Moving down the menu, the large white box contains the current version of the expression the user has created. In this case it represents but it could be expanded using the buttons below the box. The expression is shown here in a form of infix notation with the AND operator affecting the two variables either side of it. The program will store and interact with the expression in a form of reverse polish notation as discussed in the research section.
* The ‘keyboard’ in the middle of the menu contains all the functions needed for creating the Boolean expression. In GCSE notation, NOT will be represented by the ¬ character. This is also how NOT will be represented in code. In A Level notation, pressing the NOT button will place the string ‘¬(‘ into the box. The user is then free to enter their expression before pressing the ‘)’ key which will close the previously created NOT. The program should then convert the expression that will look something like ‘¬(expression)’ to the more accurate version . To create the expression currently in the box, the user will have pressed the following keys: A, +, NOT, B, ). At the end of the bottom row of keys is the clear button. This will remove the expression in the box to allow the user to enter a new one.
* Below the keyboard is a box containing information about the expression that has been created. As the program will have functions for converting the expression into a circuit, it will not be difficult to give an accurate description of the pieces the circuit will end up containing. The output menu has a similar box but also contains the number of pins the circuit will have. This is not necessary in the input box as the number of pins will always be one more than the number at the top of the menu.
* At the bottom of the input menu is the create circuit button. Pressing this will have the circuit they entered into the input menu created on the grid. If they created a circuit with the standard drag and drop functions and then went to create a new one with the Boolean menu, nothing would happen to their previous circuit until they press the create circuit button. This will clear the grid and create the new circuit.

The output menu is similar to the input menu but fulfils the opposite function:

* At the top of the menu is a drop down box that will allow the user to select which of their output pins they wish to create an expression from. In this example the user has selected the Q output. The expression that is created could vary between pins if the circuit that has been created has more than one output. In this situation there is only one output pin.
* The output menu is dominated by two white boxes. Unlike the input menu, the user has no method of entering input into these box. Instead, they will display the expressions that represents the circuit that the user has created on the main GUI. The top box displays the circuit exactly with no simplification. In this case, the top box displays. The menu then has a second box showing a simplified version of the expression. In this case, the expression is already in its simplest form. However, if a redundant NOT gate existed in the expression (i.e. ), the program would simplify the expression and remove the unnecessary NOTs.
* Below the white boxes, there is a box containing information containing information about the circuit that has been outputted. Here there are 4 pins (A, B, C, Q), 3 gates (AND, OR, NOT) and 6 wires connecting them all.
* At the bottom of the menu is the create expression button. The white boxes will be empty upon the user first opening the Boolean menu. Pressing this will initiate the process of analysing the created circuit and will place the results into the boxes.

On the right side of the screen is a box showing that the selected notation is A Level. These buttons act the same as the ones at the top of the screen. If GCSE were selected, it would be highlighted the way that A Level is now. These buttons can be pressed at any time to swap between the notations.

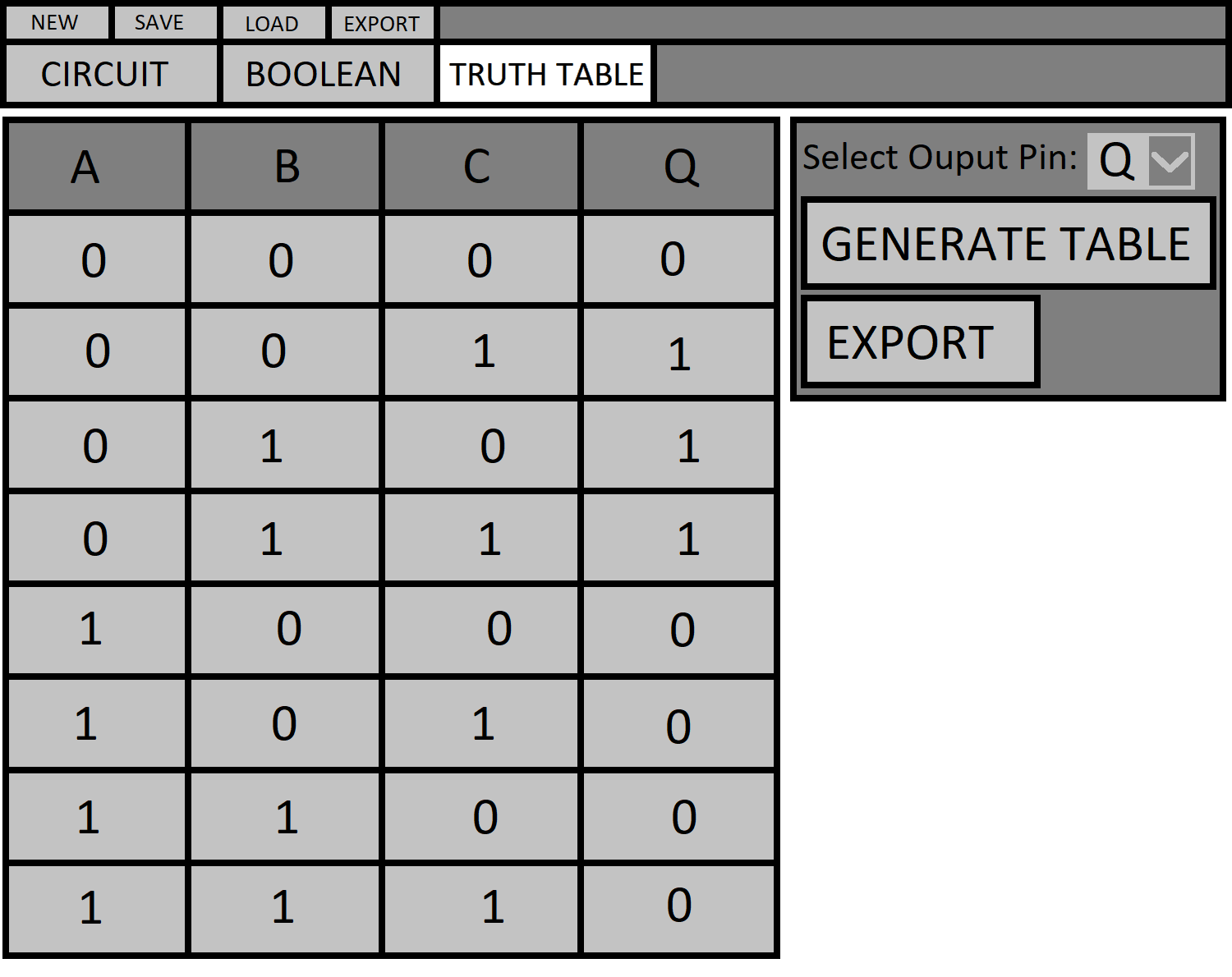
Boolean GUI Inputs

|  |  |
| --- | --- |
| **Input** | **Description** |
| Number of Inputs | Pressing the arrow buttons on the counter will increase or decrease the number of inputs in the Boolean expression. This will change the available variable buttons in the keyboard. |
| Press buttons | Pressing the buttons in the keyboard will add symbols to the expression in the box. The different buttons will act in different ways. The NOT gate will add the NOT( characters to the box which can be closed by the ) button. The ← button will remove the last symbol entered. The +, ., ^, ( and ) buttons will enter their displayed symbol. The variable buttons will enter their corresponding variable. When the user changes to GCSE notation, these buttons will shift to their GCSE alternatives. The + symbol will become ꓦ and the . symbol will become ꓥ. The NOT button will remain the same as its functionality will not change. When a NOT is closed in the expression however, it will display the ¬ character instead of placing a bar over the expression. The XOR operator will remain as ^ as there is no GCSE equivalent for this symbol. |
| Create Circuit | When the user presses the big ‘Create Circuit’ button, the program will run its algorithm for converting the Boolean expression into an actual circuit. This will involve a bunch of RPN conversion and some tree traversals. When the program is finished, the Boolean menu should close and the circuit should be shown on the grid. |
| Select Output Pin | The specific output pin that has been selected will determine how the program reads the circuit. Pressing the arrow on the drop down button will display all the output pins in the circuit. The user will then be able to select which pin they wish to use. |
| Create Expression | When the ‘Create Expression’ button is pressed, the program will undertake its analysis of the created circuit and, using tree traversal and some RPN conversion, create an expression representing it that will be displayed in the big white box. |
| Change Notation | The user will have the option in the preferences tab to change their preferred notation. By default it will be GCSE notation but A Level will also be available. When the user changes the selection in their preferences, the box in the top right will change to display the selected notation. The buttons will then shift to GCSE notation. |

Truth Table GUI

This is a very simple GUI that will provide the truth table functionality of the program

Figure 8, Design of Truth Table GUI



The only inputs on this menu appear on the right. The user is able to select the output pin the wish for the table to include and then press the generate table button to get the table. Once the table is generated they can press the export button to get an image file containing the table. In this mock-up, the circuit has only 3 inputs and yet the table takes up most of the available space. It will make sense to limit the size of truth table so as not to go off the edge of the screen when generating them. The processes related to this menu are similar to those in previous menus and so will not be explained here in depth.

## Logic Gates Rules

Every logic gate has its own rules that will be specified here. A and B will be the input variables, Q will be the output variable. All gates take two input variables except the NOT gate that only takes one. This mean NOT gates are going to need to be treated differently to every other gate whenever they are used.

**AND Gate:**

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **Q** |
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

**OR Gate:**

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **Q** |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

**XOR Gate:**

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **Q** |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

**NOR Gate:**

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **Q** |
| 0 | 0 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |

**NAND Gate:**

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **Q** |
| 0 | 0 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

**XNOR Gate:**

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **Q** |
| 0 | 0 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

**NOT Gate:**

|  |  |
| --- | --- |
| **A** | **Q** |
| 0 | 1 |
| 1 | 0 |

## File Structure

The program will use simple text files for storing circuit information. The first line in the file will store metadata about the file itself including the number of each type of component present in the file and the amount the file has been offset from the canvas origin. Each line after that will represent a different component and will store all the relevant information about that component. An example file is shown below:

2,5,4,2,0,0

220,80,0,0

400,100,1,0

160;100,220;100

160;140,220;140

160;200,360;160,400;160

300;120,400;120

480;140,560;140

140,80,0

140,120,0

140,180,0

560,120,1

40,75,Inputs

600,75,Outputs

The first line records that there are 2 gates, 5 wires, 4 pins and 2 labels in this file. The remaining two zeroes indicate that the canvas has not been offset to create this circuit. Storing this metadata means that the program will know what each line should be before it reads it. This will help to prevent the program crashing when it reads corrupted files as it will immediately know if a line is missing any data.

The next 2 lines describe the gates in the circuit. The first 2 parts of each line hold the coordinates of the gate, the next part stores the type of gate that it is and the final part records whether this gate is a NOT version of itself.

The next 5 lines describe wires. All a wire consists of is coordinates that it passes through. These coordinates are stored as integers separated by semicolons. The coordinates are then separated by commas.

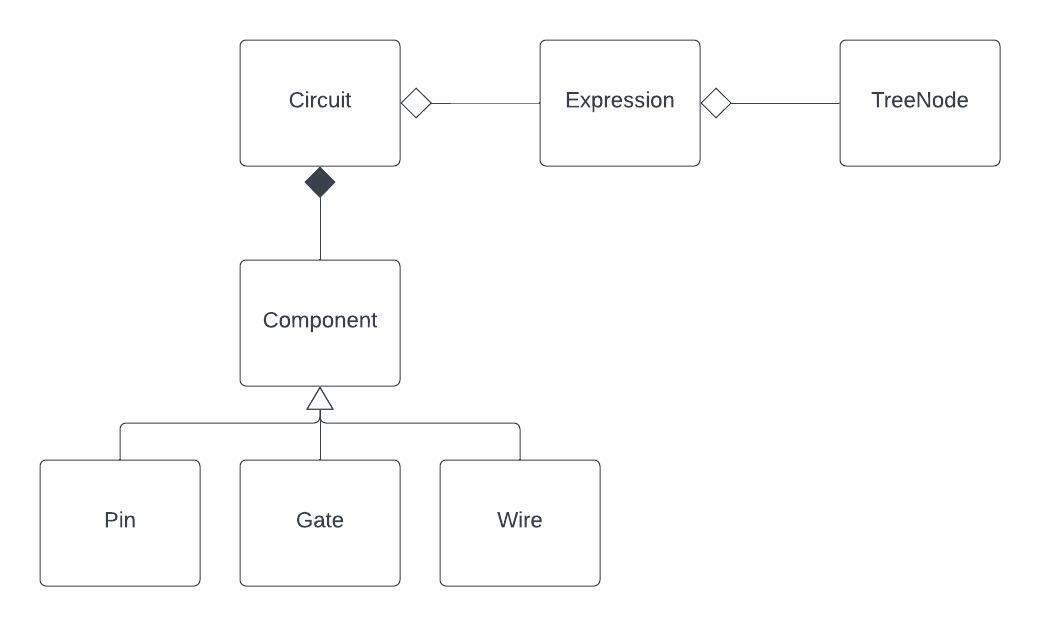
The next 4 lines represent pins. Similar to the gates, the first 2 parts of each line stores a coordinate location of the pin. The other part of the line records whether the pin is an input or output pin.

The remaining 2 lines describe labels. As with the other components, the first 2 parts store the location of the label. The text contained by the label is then kept in the remaining part of the line.

This file format should be robust in that the program will be able to predict what each line should contain and therefore should not be able to be confused by strings containing labels or new line characters. An infinite number of components can be stored in this way and the recording of the offset of the canvas allows the user to place the components wherever they wish and still have them loaded accurately.

## Classes

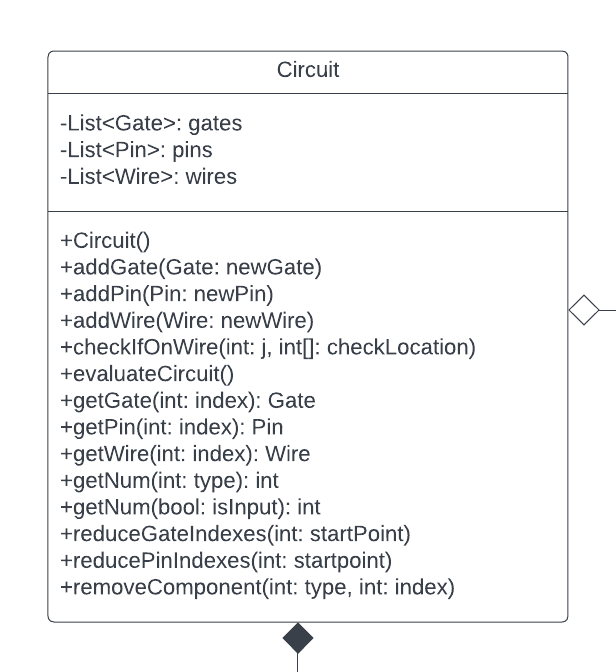
Figure 9, UML Diagram



This diagram shows the relationships between the main logical classes of the program. The centre of the diagram is the circuit class which contains all data relating to a user-created circuit that exists in the grid. The circuit will consist of lists of components. The component class will be an abstract class with three children. The pin class will handle all the data relating to individual pins, the gate class will handle all the data relating to individual gates and the wire class will handle all the data relating to wires. The circuit class will handle all the interactions between the components while the user is working with the circuit on the grid. When the output of one component changes, the circuit class will be responsible for changing the states of whatever component is connected.

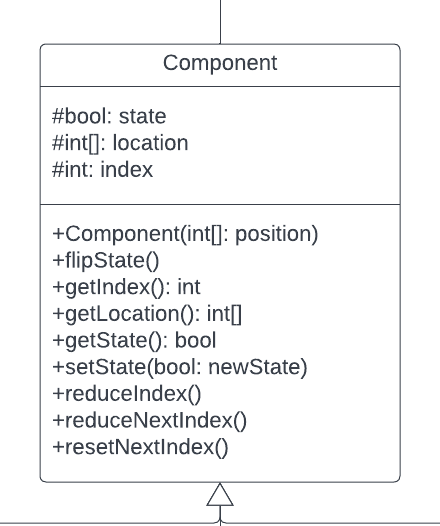
The expression class will handle all the strings and RPN interactions of the Boolean functionality. It will make use of the treenode class to generate binary trees of circuits that can be analysed to generate Boolean expressions. Expressions will be constantly generated and changed as the user creates uses the Boolean menu but they will not be used while the user is using truth tables or the main circuit grid. Treenodes have even more limited use as they will only be created when the user makes the transition between the Boolean functions and the circuit grid functions.

Circuit



The circuit class is the main hub of circuit logic. The three fields are lists of each type of component. These will hold instances of the children of the component class that will correspond with every component the user places on the grid. There are three addX methods that will be used to add components to the lists and the removeComponent method to remove them. The checkIfOnWire method will take the index of a wire and a pair of coordinates as input and will return whether that location exists on the wire indexed. This will be vital for connecting together wires as the program will have to check multiple time whether wires cross at any points. The evaluateCircuit method will parse through the lists several times and update the output of each component based on its inputs. The reduceGateIndexes and reducePinIndexes methods will be used when components are deleted so that the indexes of other components stay consistent. The remaining methods are getters to allow information about each component to be accessed while still maintaining private data types.

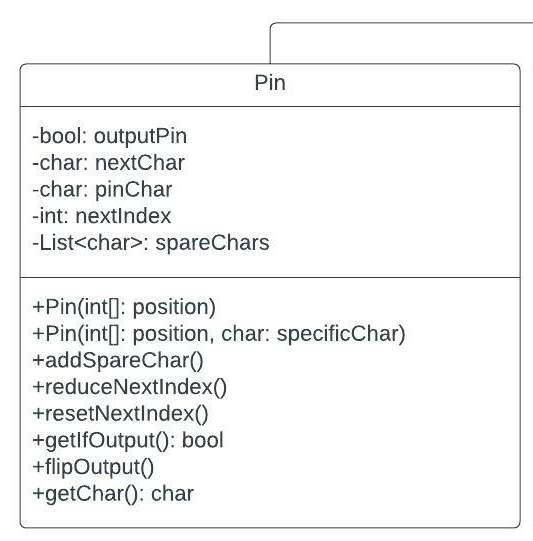
Component



The component class is an abstract class that will never be instantiated during the program. It contains fields and methods common to all components. The bool state will contain the output state of each component. The int array location will contain the coordinates of the gate. The int index will contain the index of the component that can be displayed to the user and used to distinguish between components.

The flipState method will invert the state variable and will be used when inputs are changed. setState can be used when a specific state needs to be loaded rather than just inversion. The reduceIndex method will be used to reduce the index of the component by one. The reduceNextIndex and resetNextIndex methods will be virtual and overwritten by the child classes. They will be used to change the nextIndex fields of those children, explained later. The other methods are getters that will be used by the circuit class to get data about the components.

Pin



The pin class is the child class of the component class that will be used to store pin data. The bool outputPin will store whether the pin is an input pin or an output pin. The nextChar will be a static field containing the next available character that can be assigned to a pin. This char that has been assigned will be stored in the pinChar field. nextIndex will be another static class used to hold the next available integer index for the pin which will be stored in the inherited index field. The spareChars list will be a third static field that will contain the chars of pins that have been deleted and can now be assigned to a new pin.

The class has two constructors, one for generating a pin with the next available char and one for generating a pin with a specific char. The addSpareChar method will be used when a pin is deleted to add its char to the list of spareChars so another pin can use it. The reduceNextIndex and resetNextIndex methods are overwritten from the parent class and will be used to reduce the next index field by one when a pin is deleted and to reset the index to when the circuit is reset. When the pin is swapped from an input to an output or vice versa the flipOutput method will flip the outputPin bool. The other methods are getters to get data about the pin.

Gate

A picture containing table

Description automatically generated

The gate class is the second child of the component class. It will be used to store relevant data for all gates created in a circuit. The Boolean array inputs will be an array of length 2 storing both the gate’s inputs. In the case of a NOT gate, the second bool in the array will not be used, remaining false. Instead the gate’s single input will be the first bool in the array. The nextIndex int has the same purpose as in the pin class. It will be a static field holding the next available integer for a gate to be assigned to. The not bool will store whether this gate is a NOT version of a gate with true being a not version and false being a regular version. The type int will hold what kind of gate this is. 0 will represent AND, 1 will represent OR, 2 will represent XOR and 3 will represent NOT. This int will be used to determine what output to give when the inputs are changed.

The gate’s constructor takes the location, type and not state of the gate. The Output method will use the state of the inputs and the gate type to update the state field inherited from the constructor class to the correct output state. This can be as simple as an if statement. The reduceNextIndex and resetNextIndex methods are the same as in the pin class. They are overwritten from the component class to either reduce the nextIndex field by one or set it back to the default index. The setLocation method will be used when the gate is moved to update its location. The setState method will be used to set the state of one of the gates inputs. The num int in the method will indicate which input to change and the newState bool will be the state that input needs to swap to. The remaining methods are getters for fields in the class.

Wire

A picture containing graphical user interface

Description automatically generated

Wire is the smallest class in the program but one of the most important. A wire consists of a startpoint, an endpoint and a series of turnpoints between them. The startpoint will be stored as the location field inherited from the component class. The endpoint will be stored in the int array field named as such. Then all turnpoints will be stored in the list of int arrays. All these integer arrays will be length 2 and will contain coordinates relating to points on the grid. The wire constructor will take the startpoint of the wire and assign this as the location of the wire. The addTurnPoint method will take a coordinate and add it to the list of turnpoints. The finishWire method will take the last turnpoint added to the list, copy it to the endpoint field and remove it from the list of turnpoints. The other two methods are getters for other classes to get data on the wire.

Expression

A picture containing table

Description automatically generated

The expression class will be used every time the user interacts with the Boolean GUI. The only field is the literalString which holds the exact text of the expression as entered by the user on the Boolean menu. There are a series of addX methods that will be called when the user presses buttons in the Boolean keyboard to add characters to the expression string. The checkValidity method will analyse the literalstring and then return whether or not the expression is valid. The clear method will empty the string so the user can restart their expression. The closeBracket and openBracket methods add brackets to the expression. The convertRPN method will use the shunting yard algorithm to take the literalString and convert it into reverse polish notation to make it easier to generate a binary tree. The startNOT method will add the ‘¬’ character to the expression. This will be read by the Boolean GUI as the start of a negated area of the expression. If the NOT is eventually ended later in the expression, the ‘¬’ character will be replaced with a bar over the part of the expression that is negated. The remaining methods are getters for numbers of gates in the expression which will require the program to loop through the literalstring and count the number of gates within.

TreeNode

A picture containing table

Description automatically generated

The treenode class will be the program’s implementation of a binary tree. Each treenode will represent a node in the tree and will store the nodes to the left and right of it until it reaches the end of a branch where the left and right nodes will be stored as null values. Each node will record an ID string that could be a character corresponding to a pin or an index of a gate. The isNOT bool field will be used to record if this node represents a NOT gate. If it does, there will be no right node as NOT gates only have one input.

There are three constructors for the treenode class. The first one generates a new treenode with content for an ID but no nodes to the left or to the right. This will be used for generating end nodes that represent pins. The second constructor takes an ID, a right node and a left node. This can be used to generate a treenode representing a gate with nodes to the left and to the right of it. For a NOT gate, the third constructor can be used to generate a node with a value for the isNOT field. The other important methods are the inOrderTraverse method that will be used to generate a list of tokens from the treenode and its branches and the getDepth method that will return the maximum depth of the tree starting from this node. This will be a recursive function that should return an infix expression representing the tree and the circuit the tree was generated from.

## Test Strategy

Upon completion of the programming, the program will be handed over to other students for initial testing. They will attempt to cause errors and crashes to test the validation of all inputs. This will expose problems in code that can be rectified. Once the program is fully finished, it can be handed over to the target users for testing.

The program will be tested according to each objective set out in analysis. The first four objectives should be fairly easy to test as they are basic feature requirements. The other ones are less specific and therefore will have to be tested more creatively, likely through user feedback.

1. The first objective will be simple to test. The circuits able to be created by the program can be compared to the requirements of both the GCSE and A Level specifications for logic circuits. Specifically, the objective mentions the need for these circuits to be easily creatable. This is harder to measure, though users should be able to report whether they find the circuit creation easy. A questionnaire can be used asking users whether they had difficulty with the program’s UI and how long they take to use the program for its intended purpose. This is a very important measure as the program could be used by younger students who may be put off by an unintuitive system.
2. Analysis will take the form of the Boolean algebra and Truth Table functions. Again, these will need to match the requirements of the GCSE and A Level specifications. This should be easily tested by checking whether the program can display the analysis in the correct notation as specified by exam boards. The objective states that this analysis must be useful. This can be measured again in a questionnaire to the users, especially regarding the simplification of the algebra expressions which exists solely to help the user understand the analysis.
3. This is another simple yes/no objective. Either the program does support multiple forms of circuit creation, or it does not. The aim will be for all three possible systems to work. The three forms would be:
   1. Main circuit creation menu (drag and drop, grid, etc) which will be tested more in objective 1.
   2. Loading circuits from a file.
   3. Entering a Boolean expression.
4. This objective seems simple but could become quite expansive. The program should allow circuits to be saved to a unique format that can be loaded and this will be easy to measure. The harder part comes when discussing what formats are useful to users. A PNG format will be an expectation. Testing must also show that the program can export circuits that are larger than the canvas without losing any components in the final image.
5. This is a less specific objective that will be harder to test. The obvious strategy would be to simply ask users whether they find the program simple to use, similar to the strategy of objective 1. The program needs to be intuitive for users so it may be prudent to check with users whether they find any parts hard to understand or if they are confused by any of the input methods.
6. This is a two-pronged objective. Measuring how long users take to get to grips with the system will show how intuitive it is. Regarding enjoyability, most users will not necessarily ‘enjoy’ using a program to create logic circuits. The essence of this objective is that users feel the program is responsive and fulfils their requirements so that they have a good experience while using it. This will be evident in their reactions after using it for a significant period. The issue with testing this is that most users will not be using this program for too long a period unless during lesson time. Giving the program to students learning about logic circuits would seem to be the best course of action then.
7. This is similar to previous objectives in that it will vary between age groups. Older students need to be interrogated as to whether they find the program too simple for their needs. This would arise due to too much focus being placed on younger students. Conversely, younger students need to be able to use the program without having to work with complicated features. The most important thing is balancing these two groups’ needs with each other.
8. The success of this objective depends on the definition of ‘a reasonable time frame’. It would be acceptable for the analysis of a large circuit to take a moment but users could become frustrated with a long wait. For the purposes of this testing the aim will be for all functions to complete in at most a minute though the target will be for them to be instantaneous. This can be tested by creating large circuit and Boolean expressions and testing how long the program takes to analyse them and return an output.

For the extension objectives measurement is binary. These are extra features that could be added to improve the functionality of the program. All testing will be documented before the full code listing.

## Algorithms

Evaluate

Evaluate()

For(i = 0 to numWires)

For(j = 0 to numPins)

If(pins[j] is connected to wires[i])

If(pins[j] is an input pin)

Wires[i].state ← pins[j].state

Else if(pins[j] is an output pin)

Pins[j].state ← wires[i].state

End if

End for

For(j = 0 to numGates)

If(gates[j] output is connected to wires[i])

Wires[i].state ← gates[j].state

Else if(gates[j] top input is connected to wires[i])

Gates[j].state[0] ← wires[i].state

Else if(gates[j] bottom input is connected to wires[i])

Gate[j].state[1] ← wires[i].state

End if

End for

For(j = 0 to numWires)

If(wires[i].startPoint is on wires[j] OR wires[i].endpoint is on wires[j])

Wires[i].state ← wires[j].state

End if

End for

End for

This algorithm cycles through all the wires in the circuit and then loops through all the other components to find the components at either end of the wire. Once it finds those components it set the state of the wire to the state of that component or the state of that component to the state of the wire. This algorithm will be called every time any change is made to the circuit to ensure the outputs of every component is up to date.

Create Expression

This algorithm will generate a Boolean expression to match a circuit. The created expression will be in an RPN-like form with operators coming after operands. Under A Level notation, NOT operators will take the form ¬ but will be drawn as the ̅ in the program. Traversal will be done with the separate inOrderTraverse() method that makes an in-order traversal of the circuit.

The traversal of the circuit is going to be tricky since gates and pins do not record their connections, only wires do. This means the traversal algorithm will have to check through the circuit’s wires every time it moves to a new connection. Multiple forms of the traverser will also be required as the input it is given will vary between the different types of components. Connection[0] and connection[1] here represent the input component to the wire and the output component of the wire respectively.

createExpression(int outputPinIndex, Circuit input)

Wire currentWire

string boolean

for i = 0 to input.getNum(0)

currentWire ← input.getWire(i)

if(currentWire.connection[1] = input.getPin(outputPinIndex))

break

end if

end for

While currentWire.connection[0] = Wire

for i = 0 to input.getNum(0)

currentWire2 ← input.getWire(i)

if(currentWire2.connection[1] = currentWire

currentWire = currentWire2

break

end if

end for

end while

if(currentWire.connection[0] = Gate)

boolean ← GateTraverse(currentWire.connection[0].index, input, boolean)

elseif(currentWire.connection[0] = Pin)

Expression ← currentWire.connection[0].character

endif

return boolean

GateTraverse(int gateIndex, Circuit input, string boolean)

Wire currentWire

for i = 0 to input.getNum(0)

currentWire ← input.getWire(i)

if(currentWire.connection[1] = input.getGate(gateIndex).inputs[0])

break

end if

end for

While currentWire.connection[0] = Wire

for i = 0 to input.getNum(0)

currentWire2 ← input.getWire(i)

if(currentWire2.connection[1] = currentWire

currentWire = currentWire2

break

end if

end for

end while

if(currentWire.connection[0] = Gate)

boolean ← GateTraverse(currentwire.connection[0].index, input, boolean)

elseif(currentWire.connection[0] = Pin)

boolean += currentWire.connection[0].character

end if

if(input.getGate(gateIndex).getType() = NOTGate)

return boolean + “¬”

end if

for i = 0 to input.getNum(0)

currentWire ← input.getWire(i)

if(currentWire.connection[1] = input.getGate(gateIndex).inputs[1])

break

end if

end for

While currentWire.connection[0] = Wire

for i = 0 to input.getNum(0)

currentWire2 ← input.getWire(i)

if(currentWire2.connection[1] = currentWire

currentWire = currentWire2

break

end if

end for

end while

if(currentWire.connection[0] = Gate)

boolean ← GateTraverse(currentwire.connection[0].index, input, boolean)

elseif(currentWire.connection[0] = Pin)

boolean += currentWire.connection[0].character

end if

if(input.getGate(gateIndex).getNot() = false)

if(input.getGate(gateIndex).getType() = AND)

return boolean + “.”

elseif(input.getGate(gateIndex).getType() = OR)

return Boolean + “+”

elseif(input.getGate(gateIndex).getType() = XOR)

return Boolean + “?”

end if

else

if(input.getGate(gateIndex).getType() = AND)

return boolean + “.” + “¬”

elseif(input.getGate(gateIndex).getType() = OR)

return Boolean + “+” + “¬”

elseif(input.getGate(gateIndex).getType() = XOR)

return Boolean + “?” + “¬”

end if

end if

This algorithm can be demonstrated using an example circuit. For instance, using the circuit in figure 5:

Diagram

Description automatically generated

In code, createExpression(2, exampleCircuit) will have been called. The 2 will be the index of the corresponding Q pin.

The algorithm starts from the end of the circuit, at the Q output pin. The first thing to do is find the wire that connects Q to the final logic gate. To do this, the program cycles through the list of wires in the circuit until it finds one whose output connection is Q. This is the wire between Q and the OR gate at the end of the circuit. The algorithm then checks what’s at the other end of the wire. It’s a NOT gate so the algorithm calls the gateTraversal() method. The first part of this is the same as the main method, finding the wire that connects to this gate. Once it has done that, the method checks what is at the other end of that wire, finding another gate. The method then calls itself to repeat the process. This time though, it finds a pin at the other end of the wire. Therefore, the program adds the pin’s character (A) to the string that will become the expression. It then checks that the current gate is a NOT gate, which it is. “¬” is therefore added to the string and the expression is returned to the first instance of the gateTraversal() method. This method then checks that it is on a NOT gate and adds another “¬” to the expression.

We then return to the main createExpression() method and find the wire that connects to the second input of the OR gate. Following this, we reach an AND gate. The program repeats the same process, finding the first wire and checking its input. In this case, the input is from another wire. The program now enters a while loop that will keep recurring until it finds something other than a wire, which will be the A pin. With the A pin found, the program checks the other wire connected to the AND gate and finds the B pin. It adds “B” to the expression. Now we reach the final part of the traversal method which was not reached on the NOT gates. First the program checks if the gate is the inverted version of itself by checking the not Boolean variable. Then it checks the type of the gate (AND, OR or XOR) and adds the corresponding symbol to the expression. Once AND gate has been traversed, we return to the OR gate where the program repeats the process to add the “+” symbol.

This leaves us with the RPN expression “A ¬ ¬ A B . +” which can be converted into ¬¬A + A.B, the correct form for this circuit.

**N.B: This algorithm is quite different to the one eventually used in the program but follows a similar approach**

Generate Table

For Truth Table analysis to take place, the circuit must be analysed to measure the states of output pins with different configurations of input pins. The table will be stored in a 2d array of 1s and 0s that will be created every time the truth analysis class is instantiated. The constructor of the class will get the number of input pins of the circuit and therefore know how large the table needs to be. The number of columns in the array will be equal to the number of input pins plus one for the output column. The number of rows will be determined by the number of possible inputs.

In all truth tables, it is possible to fill the input columns by simply counting up in binary. This is similar to what this algorithm will do. Most truth tables will have few inputs, often no more than 2. The program will refuse to create tables larger than 4 inputs as these can become unwieldy and difficult to display.

generateTable(Circuit input, char outputPin)

list<char> inputs

for(i = 0 to numPins)

if(pin is an input pin)

inputs.add(pins[i].char)

end if

end for

for(i = 0 to inputs.count)

rectangle thisColumn ← new rectangle(height ← 2^inputs.count, width ← 100)

canvas.add(thisColumn)

end for

for(i = 0 to 2^inputs.count)

rectangle thisRow ← new rectangle(height ← 50, width ← 100 \* inputs.count)

canvas.add(thisRow)

end for

for(i = 0 to inputs.count)

string text ← inputs[i]

canvas.add(text, left ← i\*100, top ← 0)

end for

for(i = 0 to inputs.count)

for(j = 1 to 2^inputs.count)

string thisDigit ← (j.convertToBinary()[i])

canvas.add(thisDigit, left ← i\*100, top ← j \* 50)

end for

end for

for(i = 0 to 2^input.count)

string thisOuput ← evaluateForTable(i.convertToBinary, outputPin)

canvas.add(thisOutput, left ← inputs.count \* 100, top ← i \* 50)

end for

evalutateForTable(string inputBinary, char outputPin)

bool[inputBinary.length] inputs

for(i = 0 to inputBinary.length)

if(inputBinary[i] == 1)

inputs[i] ← true

else

inputs[j] ← false

end if

end for

for(i = 0 to numPins)

if(pins[i] is input pin)

pins[i].setState(inputs[i])

end if

end for

circuit.Evaluate()

for(i = 0 to numPins)

if(pins[i].char == outputPin)

bool returnState ← pins[i].state

end if

end for

for(i = 0 to numPins)

pins[i].setState(false)

end for

circuit.Evaluate()

return returnState

The first subroutine in this algorithm generates the truth table itself, with rectangles for the columns and rows and the headings of each column being filled with a string of the char from each input pin. Then the table is filled with 1s and 0s corresponding to the digits of binary numbers. When taken as a whole binary number, the rows of a truth table should always increment until every digit is 1.

The second subroutine is called at the end of the first to generate the contents of the output column. It takes the binary string of that row as an input as well as the output pin it is testing for. It then set the state of each input pin to the corresponding 1 or 0 in the binary number. The circuit is then evaluated, and the output recorded. Then, to reset the circuit, the pins are all set to false, and the circuit is evaluated again. The state of the output pin is then returned and added to the truth table.

Create Circuit

In this algorithm, numInputs is the number of inputs the user has specified for the circuit they have created.

createCircuit(Expression mainExpression)

if(mainExpression.checkValidity())

canvas.clear()

rootNode ← generateTree(mainExpression.convertRPN)

For(i = 0 to numInputs)

Canvas.addPin(left ← 40, top ← i \* 60)

End for

Canvas.addPin(rootNode.getDepth + mainExpression.getNumNOTGates(), numInputs - 1)

drawWires(rootNode)

drawGates(rootNode)

circuit.Evaluate()

end if

drawWires(TreeNode rootNode)

Wire newWire ← new Wire(rootNode. topInputlocation)

If(rootNode.Left != null && rootNode.Left.ID.type != char)

newWire.addTurnPoint(midpoint(rootNode.topInputlocation, rootNode.Left.outputLocation))

newWire.addTurnPoint(rootNode.Left.outputLocation)

newWire.finishWire()

drawWires(rootNode.Left)

else if(rootNode.Left != null && rootNode.Left.ID.type == char)

newWire.addTurnpoint(midpoint(rootNode.topInputlocation, pin[char].location)

newWire.addTurnpoint(pin[char].location)

newWire.finishWire()

end if

Wire newWire ← new Wire(rootNode.bottomInputLocation)

If(rootNode.Right != null && rootNode.Right.ID.type != char)

newWire.addTurnPoint(midpoint(rootNode.bottomInputLocation, rootNode.Right.outputLocation))

newWire.addTurnPoint(rootNode.Right.outputLocation)

newWire.finishWire()

drawWires(rootNode.Right)

else if(rootNode.Right != null && rootNode.Right.ID.type == char)

newWire.addTurnpoint(midpoint(rootNode.bottomInputLocation, pin[char].location)

newWire.addTurnpoint(pin[char].location)

newWire.finishWire()

end if

drawGates(TreeNode rootNode)

createNewGate(rootNode.location, rootNode.ID)

if(rootNode.Left != null && rootNode.Left.ID.type != char)

drawGates(rootNode.Left)

end if

if(rootNode.Right != null && rootNode.Right.ID.type != char)

drawGates(rootNode.Right)

end if

This algorithm takes an expression and uses the generateTree algorithm (which will be described after this algorithm) to generate a binary tree. It then creates all the components in the circuit by recursively calling the two subroutines drawGates and drawWires. These subroutines are separate as the number of wires a gate needs can vary (NOT gates only have one input). The drawWires algorithm creates a wire starting at the current node’s top connector and then connects it to the output of the component to the left of the current one in the tree. The subroutine is then called recursively for the left component and the process will continue until a pin is reached. The process is then repeated for the right branch. When a pin is reached, the program connects the gate directly to the pin as these are already created in the main createCircuit algorithm. The drawGates subroutine follows a similar process but uses gates instead of wires.

Generate Tree

This algorithm generates a binary tree recursively from a list of tokens.

generateTree(list<char> booleanRPN)

string operators ← “+.^”

string characters ← “ABCD”

if(operators.contains(booleanRPN[0]))

string thisID ← booleanRPN[0]

booleanRPN.Remove(0)

return new TreeNode(thisID, generateTreeFromString(booleanRPN), generateTreeFromString(booleanRPN))

else if(booleanRPN[0] == “¬”)

string thisID ← booleanRPN[0]

booleanRPN.Remove(0)

return new TreeNode(thisID, null, generateTreeFromString(booleanRPN))

else if(characters.contains(booleanRPN[0]))

string thisID ← booleanRPN[0]

booleanRPN.Remove(0)

return new TreeNode(thisID)

end if

The algorithm starts with the first token in the Boolean list and checks what it is. Once it knows what the token is, the algorithm saves it to thisID and removes it from the list. It then generates a new TreeNode with its ID as thisID. If the token was some kind of gate, the algorithm is then called recursively to generate the left and right branches of that new TreeNode. If the token is a NOT gate specifically, the algorithm does not generate one of the branches as NOT gates only have one input. If the token is a character then it represents a pin. Pins are end nodes with no branches and so a new TreeNode is returned with thisID but no further recursive calls are made.

Exporting

This algorithm takes the components on the grid, assigns them to a new canvas and then saves that canvas as a PNG. The canvas to be saved is the newCanvas.

ExportCircuit(Canvas newCanvas)

For(i = 0 to numPins)

newCanvas.addPin(pins[i].location)

end for

for(i = 0 to numGates)

newCanvas.addGate(gate[i].location)

end for

for(i = 0 to numWires)

newCanvas.addWire

for(j = 0 to wires[i].numTurnPoints)

newCanvas.wires[i].addTurnPoint(wires[i].getTurnPoints()[j])

end for

newCanvas.wires[i].finishWire()

end for

int canvasWidth

int canvasHeight

for(i = 0 to newCanvas.children.count)

if(canvasWidth < newCanvas.children[i].left)

canvasWidth = newCanvas.children[i].left

end if

if(canvasHeight < newCanvas.children[i].top)

canvasHeight = newCanvas.children[i].top

end if

end for

newCanvas.width = canvasWidth

newCanvas.height = canvasHeight

ExportToPNG(newCanvas)

This algorithm loops through all the components on the grid and creates duplicates on the newCanvas. This creates a canvas with all the correct children, however the canvas doesn’t exist in XAML so its width and height are not defined meaning it cannot be exported. To rectify this, the algorithm traces through all the canvas’ children and finds the greatest left value and top value of each. It then assigns these as the width and height of the canvas respectively. This means that the canvas dimensions encompass the components and nothing else. The canvas is then exported to a PNG file.

Saving

saveFile()

SaveFileDialog thisDialog ← SaveFileDialog.show()

String fileText = numGates + “,” + numWires + “,” + numPins + “,” + totalOffset + “\n”

if(thisDialog.Result == true)

for(i = 0 to numGates)

fileText += gates[i].Location + “,” + gates[i].gateType + “,” + gates[i].isNOT + “\n”

end for

for(i = 0 to numWires)

fileText += wires[i].Location[0] + “;” + wires[i].Location[1]

for(j = 0 to wires[i].numTurnPoints)

fileText += “,” + wires[i].getTurnPoint(j)[0] + “;” + wires[i].getTurnPoint(j)[1]

end for

fileText += “,” + wires[i].End[0] + “;” + wires[i].End[1] + “\n”

end for

for(i = 0 to numPins)

fileText += pins[i].Location + “,” + pins[i].isOutput + “\n”

end for

for(i = 0 to numLabels)

fileText += labels[i].Location + “,” + Labels[i].Text + “\n”

end for

File.Write(fileText)

end if

This algorithm carries out the saving function in accordance with the file format mentioned earlier in the design. If the user chooses to save, the file text first takes the metadata of the circuit including the number of each component and the total amount the canvas has been offset during the creation of the circuit. Then the algorithm loops through each set of components and adds the relevant data to the file text. Finally, when all the components and labels have been added, the algorithm writes the text to the file.

Loading

loadFile()

LoadFileDialog thisDialog ← LoadFileDialog.show()

Int[] metadata ← int[6]

If(thisDialog.Result == true)

fileText ← file.Read()

for(i = 0 to 5)

metadata[i] ← fileText[0].split(‘,’)[i]

end for

resetMainCircuit()

totalOffset ← metadata[5]

int[,] gateInfo ← new int[metadata[0], 4]

for(i = 0 to metadata[0])

for(j = 0 to 4)

gateInfo[i, j] ← fileText[i + 1].split(‘,’)[j]

end for

createNewGate(gateInfo[i, 0], gateInfo[i, 2], gateInfo[i, 3])

end for

list<int[]> wireInfo ← new list<int[2]>

for(i = 0 to metadata[1])

for(j = 0 to fileText[i + metadata[0] + 1].split(‘,’).Length)

wireInfo.Add(fileText[i + metadata[0] + 1].split(‘,’)[j]

end for

Wire thisWire = new Wire(wireInfo[0])

If(wireInfo.Length == 2)

thisWire.addTurnPoint(wireInfo[1])

else

for(j = 0 to wireInfo.Length)

thisWire.addTurnPoint(wireInfo[j])

end for

end if

circuit.addWire(thisWire)

end for

int[,] pinInfo ← new int[metadata[2], 3]

for(i = 0 to metadata[2])

for(j = 0 to 3)

pinInfo[i, j] ← fileText[i + metadata[0] + metadata[1] + 1].split(‘,’)[j]

end for

Pin thisPin ← new Pin(pinInfo[i, 0])

If(pinInfo[i, 2] == true)

drawPin(pinInfo[i, 0], true)

thisPin.flipOutput()

else

drawPin(pinInfo[i, 0], false)

end if

circuit.addPin(thisPin)

end for

for(i = 0 to metadata[3])

string[] labelInfo ← fileText[i + metadata[0] + metadata[1] + metadata[2] + 1].split(‘,’)

string thisLabel ← labelInfo[2]

Canvas.add(thislabel, left ← labelInfo[0], top ← labelInfo[1])

End for

Circuit.evaluate()

This algorithm takes the output of the saveFile algorithm and reverses the process to recreate the circuit. First it gets the metadata for the circuit from the first line of the file. It then knows how many times to loops through each part of the file to get the correct number of components. It takes the info from each line by splitting the line around the commas and then creates components with the data from each part of the file text. At the end, the algorithm calls the evaluate process from the circuit to set the correct states of every component.

# Testing

## Testing Outline

There will be two parts to testing. First, a series of tests will be performed by me on the program itself. These will check for any issues in code or unhandled exceptions that can be encountered when using the program, as well as making sure the basic functions of the program work as intended. The first 4 objectives as well as the 8th objective can be tested in this way. Then the program will be given to prospective users (students and teachers) and they will be asked about their experience with the program. This will be used to test the other 3 objectives.

The specific purpose of each section of tests is defined in the test strategy in the design section. Each objective has its own set of tests that will check the suitability and accuracy of each main feature of the program.

The tests for the program will be categorised in a table:

Each test will be identified with a 2-part ID in the first column. The first part of the ID will correspond to the objective that is being tested and the second part will correspond to the test number for that objective.

The data used in the test (circuit used, Boolean expression used, etc.) will be kept in the second column. This could take the form of the contents of a file that could be loaded back into the program or a literal string of a Boolean expression.

The third column will contain the expected result of the test. This could be as simple as ‘Program continues without crashing’ or may be more complex.

The fourth column will contain the actual result of the test. This could be as simple as ‘Program crashes’ or may be identical to the expected result. It may be necessary to refer to an image result, all of which will be found at the end of the tests.

If the test is successful and the program performs as intended, the fifth column will state a pass for the test. Otherwise, if the program did not perform in the expected way, the fifth column will state a fail for the test.

If the fifth column states that the test failed, the final column will record what action was taken to rectify the error. For issues with error handling and input sanitation this will likely be a small addition to the program. For larger failures, greater action may be taken.

## Program Tests

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test ID** | **Test Data** | **Expected Result** | **Actual Result** | **Pass/Fail** | **Action Taken** |
| **1.1** | **File text:**  1,3,3,0,0,0  180,40,0,0  140;60,180;60  140;100,180;100  260;80,340;80  120,40,0  120,80,0  340,60,1 | The test data contains a simple AND gate circuit. The user should be able to change the states of both input pins ‘A’ and ‘B’ and an accurate output should be returned in the output pin ‘C’. | When neither or only one of the input pins are on, the output is off. When both inputs are on, the output is on. This is the correct behaviour of an AND gate. | **Pass** |  |
| **1.2** | **File text:**  1,3,3,0,0,0  180,40,1,0  140;60,180;60  140;100,180;100  260;80,340;80  120,40,0  120,80,0  340,60,1 | The test data contains a simple OR gate circuit. The user should be able to change the states of both input pins ‘A’ and ‘B’ and an accurate output should be returned in the output pin ‘C’. | When neither input pin is on, the output is off. When either or both input pins are on, the output is on. This is the correct behaviour of an OR gate. | **Pass** |  |
| **1.3** | **File text:**  1,3,3,0,0,0  180,40,2,0  140;60,180;60  140;100,180;100  260;80,340;80  120,40,0  120,80,0  340,60,1 | The test data contains a simple XOR gate circuit. The user should be able to change the states of both input pins ‘A’ and ‘B’ and an accurate output should be returned in the output pin ‘C’. | When neither input pin is on, the output is off. If either input pin is on, the output is on. If both input pins are on, the output is off. This is the correct behaviour of an XOR gate. | **Pass** |  |
| **1.4** | **File text:**  1,2,2,0,0,0  180,20,3,1  140;60,180;60  280;60,360;60  120,40,0  360,40,1 | The test data contains a simple NOT gate circuit. The user should be able to change the state of the input pin ‘A’ and the output pin ‘B’ should contain the opposite state. | When the input pin is on, the output pin is off. When the input pin is off, the output pin is on. This is the correct behaviour of a NOT gate. | **Pass** |  |
| **1.5** | **File text:**  1,3,3,0,0,0  180,40,0,1  140;60,180;60  140;100,180;100  280;80,360;80  120,40,0  120,80,0  360,60,1 | The test data contains a simple NAND gate circuit. The user should be able to change the states of both input pins ‘A’ and ‘B’ and an accurate output should be returned in the output pin ‘C’ | When neither or only one of the input pins are on, the output is on. When both inputs are on, the output is off. This is the correct behaviour of a NAND gate. | **Pass** |  |
| **1.6** | **File text:**  1,3,3,0,0,0  180,40,1,1  140;100,180;100  280;80,360;80  140;60,180;60  120,40,0  120,80,0  360,60,1 | The test data contains a simple NOR gate circuit. The user should be able to change the states of both input pins ‘A’ and ‘B’ and an accurate output should be returned in the output pin ‘C’ | When neither input pin is on, the output is on. When either or both of the input pins are on, the output is off. This is the correct behaviour of a NOR gate. | **Pass** |  |
| **1.7** | **File text:**  1,3,3,0,0,0  180,40,2,1  140;100,180;100  280;80,360;80  140;60,180;60  120,40,0  120,80,0  360,60,1 | The test data contains a simple XNOR gate circuit. The user should be able to change the states of both input pins ‘A’ and ‘B’ and an accurate output should be returned in the output pin ‘C’ | When neither input pin is on, the output is on. If either input pin is on, the output is off. If both input pins are on, the output is on. This is the correct behaviour of an XNOR gate. | **Pass** |  |
| **1.8** | **File text:**  4,9,6,0,0,0  220,80,0,0  400,100,1,0  560,120,2,0  220,240,1,1  180;200,360;160,400;160  180;260,220;260  180;300,220;300  180;100,220;100  180;140,220;140  300;120,400;120  480;140,560;140  320;280,520;180,560;180  640;160,700;160  160,80,0  160,120,0  160,180,0  160,240,0  160,280,0  700,140,1 | The test data contains a complex circuit with 4 gates, 5 input pins and 9 wires. When the states of the input pins are altered, the circuit should update visually and logically to produce accurate outputs from all the gates and at the single output pin. | After testing all the possible combinations of inputs, the output pin had an accurate output for every test. | **Pass** |  |
| **1.9** |  | When a user has selected a gate, if the mouse is over the canvas a ghost of that gate should be drawn. | The ghost is drawn of each gate accurately and it moves with the cursor around the canvas. | **Pass** |  |
| **1.10** |  | When the user has selected a pin, if the mouse is over the canvas a ghost of a pin should be drawn. | The ghost of the pin is drawn and it moves with the cursor around the canvas. | **Pass** |  |
| **1.11** |  | When the user is drawing a wire, the program should indicate where the next turnpoint of the wire will be if they click and should show the segments of the wire that have already been entered. | The ghosts are drawn as shown in figure 16, indicating the next turnpoint and the segments already entered accurately. | **Pass** |  |
| **1.12** | **File text:**  3,7,5,0,0,0  240,100,1,0  260,120,0,0  440,120,1,0  160;120,240;120  120;140,260;140  160;160,240;160  120;180,260;180  320;140,440;140  340;160,400;180,440;180  520;160,580;160  140,100,0  140,140,0  100,120,0  100,160,0  580,140,1 | The circuit in the file text contains an AND gate and an OR gate overlapping on the grid. However, since no wires overlap, when the user changes the input states, the circuit sould evaluate with no issues. | The circuit evaluates with no issues, as though the circuits are not overlapping at all. | **Pass** |  |
| **1.13** | **File text:**  1,3,4,0,0,0  240,140,0,0  160;160,240;160  160;200,240;200  320;180,420;180  140,140,0  140,140,0  420,160,1  140,180,0 | The circuit in the file text contains two pins in the same position. This is an invalid circuit so there is no expected functionality. The program must be able to handle this situation i.e. the user shouldn’t be able to select both pins at once and the program must not crash. | The user cannot select both pins at once and the program does not crash. Changing the state of the pin the user can select does not change the state of the wire connected to it. This behaviour is could be interpreted as a fault but since this circuit is invalid, any behaviour that allows the circuit to function correctly would be faulty. | **Pass** |  |
| **1.14** | **File text:**  0,5,1,0,0,0  140;120,380;300  380;220,500;220  320;120,320;220  340;120,340;60,460;180,600;140  460;220,460;340,560;380,560;400,660;380,640;260  120,100,0 | The file contains several wires branching off of each other, all connected to a single pin. When the pin is turned on, the wires should all take the state of the wire they connect to (on) and when the pin is off the wires should again take the state of the wire they connect to (off). | The wires in an on state can be seen in figure 17. When the pin is on, all wires are on. When the pin is off, all wires are off. This is the correct behavious. | **Pass** |  |
| **1.15** |  | When the user is in delete mode and they click on a component, it should be removed from the circuit. | When the user enters delete mode, any component they mouse over is highlighted in red. When they click on it, it is removed from the circuit and the grid. | **Pass** |  |
| **1.16** | **File text:**  0,0,2,0,0,0  620,80,0  620,200,0 | The circuit contains two pins, ‘A’ and ‘B’. When the user deletes a pin, the next pin they add should have the index of the first one they deleted. | If pin ‘A’ is deleted, when the user goes to place a new pin it is given the index ‘A’ rather than ‘C’. | **Pass** |  |
| **1.17** | **File text:**  0,0,2,0,0,0  620,80,0  620,200,0 | Similar to test 1.16 except instead of the pin being deleted, when the user presses the ‘new’ button, the pin indexes should be reset. | The pin indexes are not reset, the first pin to be placed after the ‘new’ button is pressed is pin ‘C’. | **Fail** | Added missing code to call the resetNextIndex() function when the ‘new’ button is pressed. |
| **1.18** |  | When the user presses the movement buttons in the bottom right of the UI, all components on the grid should move in opposite direction to the button specified (mimicking the movement of a camera). | The components move one square in the opposite direction to the button specified, every time the button is pressed. | **Pass** |  |
| **1.19** |  | When the user attempts to scroll past the logical edge of the canvas, the program should stop them and display a message saying they cannot move any further. | The program displays the message ‘Cannot move - Edge of canvas reached’ and does not move any further. | **Pass** |  |
| **2.1** | **File text:**  2,5,4,0,0,0  200,80,0,0  360,140,1,0  120;100,200;100  120;140,200;140  280;120,320;160,360;160  120;200,360;200  440;180,520;180  100,80,0  100,120,0  100,180,0  520,160,1 | When the user presses the ‘create expression’ button while the Boolean GUI is in A Level mode, an accurate expression for the circuit should be returned. | The program outputs C + B.A which correctly describes the circuit in the grid using A Level notation. | **Pass** |  |
| **2.2** | **File text:**  2,5,4,0,0,0  200,80,0,0  360,140,1,0  120;100,200;100  120;140,200;140  280;120,320;160,360;160  120;200,360;200  440;180,520;180  100,80,0  100,120,0  100,180,0  520,160,1 | When the user presses the ‘create expression’ button while the Boolean GUI is in GCSE mode, an accurate expression for the circuit should be returned. | The program outputs C **˄** B **˅** A which correctly describes the circuit in the grid using GCSE notation. | **Pass** |  |
| **2.3** | **File text:**  4,7,4,0,0,0  200,80,0,0  320,80,3,1  460,80,3,1  640,120,1,0  120;100,200;100  120;140,200;140  720;160,760;160  560;120,600;140,640;140  420;120,460;120  280;120,320;120  120;200,600;180,640;180  100,80,0  100,120,0  100,180,0  760,140,1 | This circuit contains two NOT gates that cancel each other’s effects out. When the user presses the ‘create expression’ button while the Boolean GUI is in A Level mode, an accurate expression for the circuit should be returned, followed by an accurate simplification. | The program outputs which is an accurate expression for the circuit. The program then outputs C + B.A as a simplified form which is also accurate. | **Pass** |  |
| **2.4** | **File text:**  2,5,4,0,0,0  200,80,0,0  360,140,1,0  120;100,200;100  120;140,200;140  280;120,320;160,360;160  120;200,360;200  440;180,520;180  100,80,0  100,120,0  100,180,0  520,160,1 | When the user presses the ‘generate table’ button while in the truth table GUI, an accurate truth table for the circuit should be generated. | The program outputs the table shown in figure 18. This correctly describes the circuit behaviour. | **Pass** |  |
| **2.5** | **Output pin with no connections** | The program should fail to output a Boolean expression or a truth table, showing a message box explaining such. | The program crashes when the generate truth table button or the generate circuit button are pressed. | **Fail** | Error handling added to Boolean circuit generation code. |
| **2.6** |  | When a user selects a gate, the program should display data about that gate such as the input states and its output state. This data should update as the circuit changes. | The program does display the gate data but the data does not update immediately when a change is made to the circuit. | **Fail** | Added an extra parseCircuit() call to the end of the drawing logic. Data now updates every time a component is drawn or redrawn which should be every time a change is made. |
| **2.7** | **File text:**  1,3,3,0,0,0  320,100,0,0  240;120,320;120  240;160,320;160  400;140,520;140  220,100,0  220,140,0  520,120,1 | The file is a simple AND gate with 1 gate, 3 pins and 3 wires. When the program generates a Boolean string to describe the circuit, it should also output the number of each component in the designated boxes. | The program returns B.A as the expression. It outputs the number of pins as 3, the number of gates as 1 and the number of wires as 3. All of which are accurate. | **Pass** |  |
| **2.8** | **Circuit with 4 or more input pins.** | When the user attempts to create a truth table describing a circuit with more than 3 input pins, the program should display an error saying it cannot generate the table. | The program displays the error message ‘Cannot generate truth table with more than 3 inputs’. | **Pass** |  |
| **2.9** | **Circuit with 4 or more input pins.** | If the user attempts to export the truth table without generating it first and the circuit has more than 3 input pins, the program should display an error saying it cannot generate the table. | The program displays the error message ‘Cannot generate truth table with more than 3 inputs’. | **Pass** |  |
| **3.1** | **Expression entered into Boolean menu.** | The program should accurately generate the circuit on the grid. | The program generates the circuit shown in figure 19 which is an accurate representation of the expression. | **Pass** |  |
| **3.2** | **Expression entered into Boolean menu.** | The program should draw a NOT hate across the B + C section when the bracket is closed. | The program creates the expression shown in figure 20 which is accurate. | **Pass** |  |
| **3.3** | **Expression ABC.)D( entered into Boolean menu.** | The program should show an error message that the expression is invalid when the ‘create circuit’ button is pressed. | The program displays an error message. | **Pass** |  |
| **3.4** | **Expression entered into Boolean menu.** | The program should give an accurate readout of the number of gates and wires in the circuit it has created. | The program returns 2 as the number if gates and 5 as the number of wires which is accurate. | **Pass** |  |
| **3.5** | **File text:**  2,5,4,2,0,0  140,40,0,0  300,100,1,0  100;60,140;60  100;100,140;100  220;80,280;120,300;120  100;160,300;160  380;140,480;140  80,40,0  80,80,0  80,140,0  480,120,1  20,195,Inputs  500,195,Output | When the user selects a file to load, the program should generate the circuit based on the data contained in the file. | The program loads the file accurately with every component in the correct location. |  |  |
| **3.6** | **Circuit with a label that contains a comma** | If a label contains a comma, the file system should either fail to load the file or load the label regardless. | The program loads the part of the label before the commas. | **Pass** |  |
| **3.7** | **Circuit with a label that contains a ‘\n’ new line code** | If a label contains a new line code, the file system should either fail to load the file or load the label regardless. | The program loads the label regardless. | **Pass** |  |
| **3.8** | **File with invalid formating** | If a file becomes corrupted or is in any way invalid, the program should display an error and fail to load it. | The program displays an error message: ‘Failed to load file’ and no file is loaded. | **Pass** |  |
| **3.9** |  | When the user presses the ‘new’ button in the top left corner of the GUI, the circuit grid should be cleared and all component indexes should be reset. | The canvas is wiped, gate indexes are reset to 1 and pin indexes are reset to A. Wire indexes are not reset but since these are never seen by the user it is not an issue. | **Pass** |  |
| **4.1** | **Empty circuit** | If an empty circuit is exported, the program should fail to execute. | The program crashes when the export button is pressed. | **Fail** | Error handling added to circuit export code. Now an error message is displayed. |
| **4.2** |  | If the program tries to overwrite an already saved file, it should overwrite the text in that file. | The text is appended to the end of the file. | **Fail** | Replaced ‘File.AppendAllText’ code with ‘File.WriteAllText’ |
| **4.3** | **File text:**  2,5,4,2,0,0  140,40,0,0  300,100,1,0  100;60,140;60  100;100,140;100  220;80,280;120,300;120  100;160,300;160  380;140,480;140  80,40,0  80,80,0  80,140,0  480,120,1  20,195,Inputs  500,195,Output | When instructed to export the circuit, the program should output a PNG containing an image of the circuit with no gridlines and all components in their correct positions. | The program outputs the image shown in figure 21. This is identical to the circuit created on the canvas except without gridlines. | **Pass** |  |
| **4.4** | **File text:**  2,5,4,2,0,0  140,40,0,0  300,100,1,0  100;60,140;60  100;100,140;100  220;80,280;120,300;120  100;160,300;160  380;140,480;140  80,40,0  80,80,0  80,140,0  480,120,1  20,195,Inputs  500,195,Output  **The A and B pins have been swapped to their True state.** | When instructed to export the circuit, the exported PNG should include the state of each wire in the circuit at the time of the exporting. | The program outputs the image shown in figure 22. This has the correct states of every wire. | **Pass** |  |
| **4.5** | **File text:**  1,3,3,0,0,0  820,60,1,0  900;100,1120;100  100;80,820;80  100;120,820;120  80,60,0  80,100,0  1120,80,1 | The circuit in the file text goes far off to the right side of the canvas. When exported, the program should output a PNG containing all the components with none of them cut off. | The program outputs the image shown in figure 23. This contains the correct circuit with no components lost. | **Pass** |  |
| **4.6** | **Circuit with a label that is located on the canvas but extends far off it.** | The program should export an image that is wide enough to contain the entire label. | The program cuts the label off. | **Fail** | Added label width to export canvas dimensions. Now labels are kept on the image. |
| **4.7** | **Circuit with the camera offset by 10 clicks to the right and 10 clicks downwards.** | When the circuit is exported, the empty space the camera is viewing should not be kept in the image. | The exported file contains only the circuit components, not empty space. | **Pass** |  |
| **4.8** | **File text:**  2,5,4,0,0,0  220,60,0,0  360,100,2,0  300;100,340;120,360;120  160;180,340;160,360;160  440;140,480;140  160;80,220;80  160;120,220;120  140,60,0  140,100,0  140,160,0  480,120,1 | When a truth table is generated, if the user presses the export table button, a PNG file containing an image of the truth table should be outputted. | The image in figure 24 is output to a PNG file. This truth table is an accurate representation of the circuit in the file. | **Pass** |  |
| **8.1** | **The expression A.(B+C)^D was entered into the Boolean menu.** | The expression in the Boolean menu is one of the largest possible with the program’s implementation of Boolean. When the ‘create circuit’ button is pressed, the circuit should be created almost instantaneously. | The circuit is produced practically immediately. There is no noticeable delay. | **Pass** |  |
| **8.2** | **File text:**  5,11,7,0,200,0  -60,40,0,0  100,100,0,0  260,160,0,0  420,220,0,0  580,280,0,0  -140;60,-60;60  -140;100,-60;100  -140;160,100;160  -140;220,260;220  -140;280,420;280  -140;340,580;340  20;80,80;120,100;120  180;140,240;180,260;180  340;200,400;240,420;240  500;260,560;300,580;300  660;320,720;320  -160,40,0  -160,80,0  -160,140,0  -160,200,0  -160,260,0  -160,320,0  720,300,1 | The circuit contained in the file text is quite large. When an expression is generated to represent it, the program should take little to no time to create the expression. | The expression is produced practically immediately. There is no noticeable delay. | **Pass** |  |

Figure 10, Test 1.11

Chart

Description automatically generated

Figure 11, Test 1.14

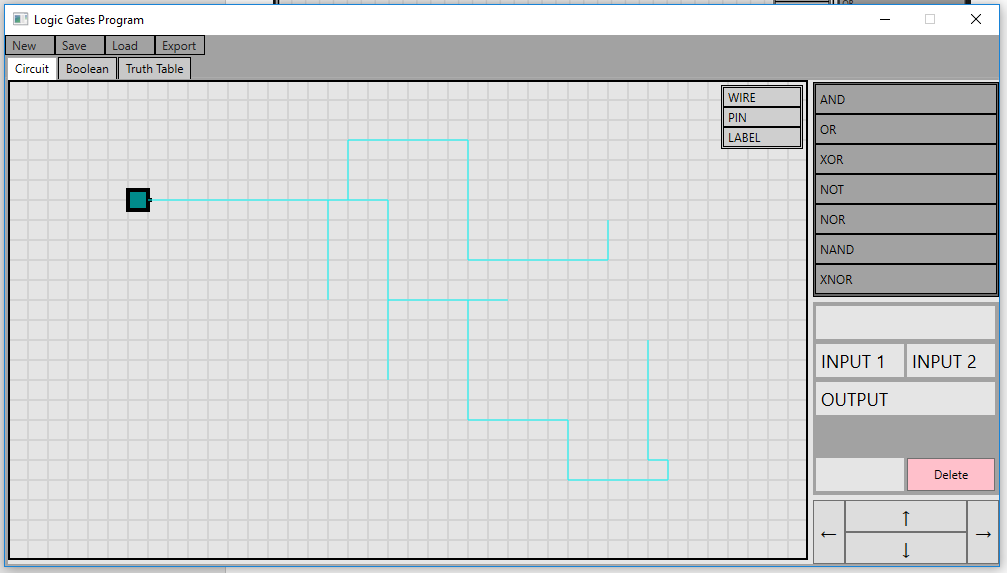


Figure 12, Test 2.4

Table

Description automatically generated

Figure 13, Test 3.1

Diagram

Description automatically generated

Figure 14, Test 3.2

Graphical user interface

Description automatically generated

Figure 15, Test 4.3

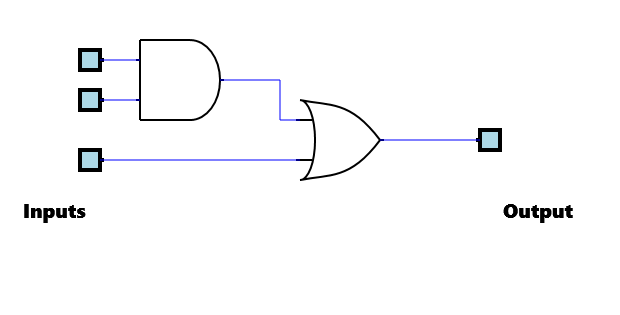


Figure 16, Test 4.4

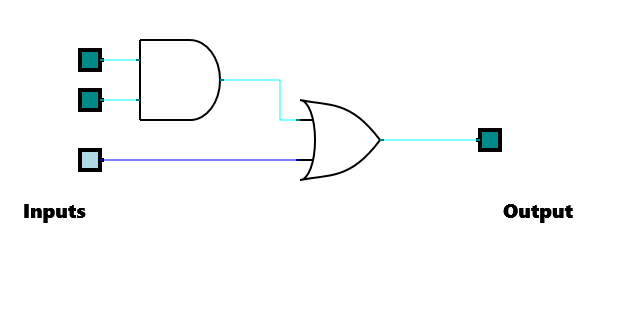


Figure 17, Test 4.5

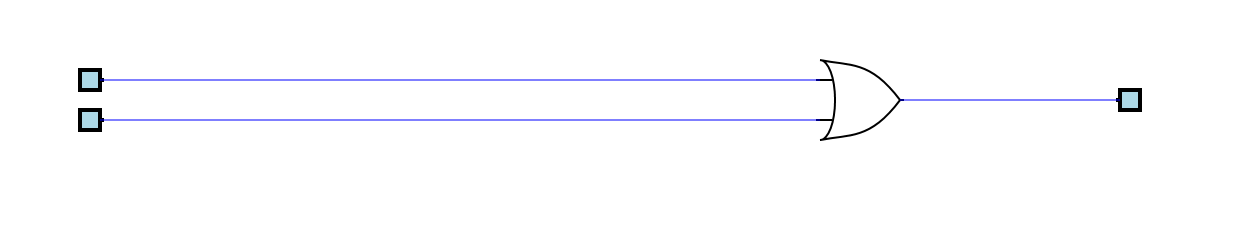
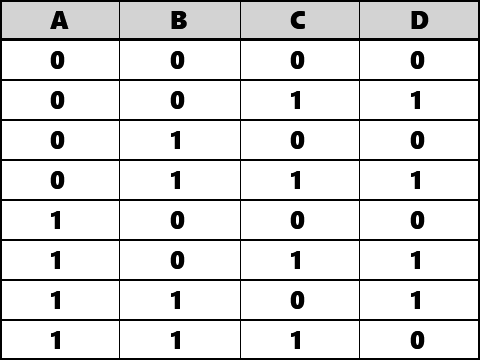


Figure 18, Test 4.8



## User Tests

Questionnaire

A questionnaire was given to several sixth form students with an explanation of the program’s functions and a series of tasks to carry out. The questions were as follows:

1. Please create the following circuit using the program and describe any issues you encounter:

A picture containing chart

Description automatically generated

1. How long did creating that circuit take?
   1. Less than 2 minutes
   2. 3-5 minutes
   3. More than 5 minutes
2. Please create the following circuit using the program and describe any issues you encounter:

Graphical user interface

Description automatically generated

1. How long did creating that circuit take?
   1. Less than 2 minutes
   2. 3-5 minutes
   3. More than 5 minutes
2. Please enter the following Boolean expression into the program, create the circuit and describe any issues you encounter:
3. How long did creating that expression take?
   1. Less than a minutes
   2. 2-3 minutes
   3. More than 3 minutes
4. How would you describe your understanding of logic circuits?
   1. Complete understanding
   2. Solid understanding
   3. Basic understanding
   4. No understanding
5. How would you describe your understanding of Boolean algebra?
   1. Complete understanding
   2. Solid understanding
   3. Basic understanding
   4. No understanding
6. Would you describe the program as responsive? If so, how? If not, what improvements could be made?
7. Would you say this program would be helpful for learning about logic circuits and Boolean algebra? If so, how? If not, why not?
8. Do you find the interface cluttered or hard to navigate?
   1. The interface is cluttered and this causes me difficulty
   2. The interface is cluttered but I don’t find it a problem
   3. The interface is not cluttered
9. Are there any other comments you have about the program from a user perspective?

Reponses

Question 1:

Most users reported no issues. One encountered an unexpected bug that required them to restart the program but after examination the issue was fixed.

Question 2:

One user said the circuit took them more than five minutes to make. Some respondents reported the circuit taking more than 3 minutes to make. However, the majority of students reported taking less than 2 minutes to make the circuit.

Question 3:

Most users again reported no issues. One respondent said they had difficulty distinguishing the wires in an on state from the white background however no other users reported this issue.

Question 4:

This circuit was larger, and the majority of users took 3-5 minutes to make it, in contrast to the majority in question 2 that took less than 2 minutes. Again though, only one user required more than 5 minutes to create the circuit. Some were also able to make this one in less than 2 minutes.

Question 5:

All users were able to create the expression. Two users reported an issue with needing extra brackets to make the NOT gate function. This was fixed in code afterwards. One user was confused when the program did not automatically swap to the circuit menu when they pressed the ‘create circuit’ button. They did figure out they had to switch manually though and the problem did not reoccur.

Question 6:

There was a mix of responses to this question. The plurality of respondents took 2-3 minutes though a sizeable portion of users took less than a minute. Two users took more than three minutes.

Question 7:

All respondents reported at least a basic understanding of logic circuits with the vast majority saying their have a solid understanding. A small number of respondents reported a complete understanding of logic circuits.

Question 8:

Again, all respondents reported at least a basic understanding. However, there were far more students who reported only a basic understanding rather than a solid understanding. A smaller number of students reported a complete understanding of logic circuits.

Question 9:

All respondents said they thought the program was responsive. Some requested further quality of life features like keyboard shortcuts or a ctrl+Z undo feature.

Question 10:

Many users were enthusiastic about the program being used to teach Boolean algebra and logic circuits. Others also said it would be useful to check answers to questions. Specific quotes include but are not limited to “This is great! No negatives!”, “Absolutely! The Boolean output and truth table generation is perfect for learning” and “Yes [it is] excellent at visualising and showing the links between algebra and logic circuits”

Question 11:

Almost all respondents said the interface was not cluttered. The few who said it was chose the option that it did not cause any issues.

Question 12:

The main response was that the program’s input UI could be fiddly initially but became quite easy to use once users understood the basic functions.

These responses indicate that the program meets the requirements of most if not all of the remaining objectives. The testing outcomes are talked about in the project evaluation.

# Code Listing

## XAML

<Window x:Class="MainProgram.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:MainProgram"

mc:Ignorable="d"

Title="Logic Gates Program" Height="560" Width="1000" Background="#FFA2A2A2" ResizeMode="CanMinimize" SizeToContent="WidthAndHeight" BorderBrush="Black" MouseLeftButtonDown="Window\_MouseLeftButtonDown" MouseRightButtonDown="Window\_MouseRightbuttonDown" Closing="Window\_Closing">

<Grid>

<!--Top menu for file interaction-->

<Menu HorizontalAlignment="Left" Height="20" VerticalAlignment="Top" Width="792" Background="#FFA2A2A2">

<MenuItem x:Name="butNew" Header="New" Height="20" Width="50" BorderBrush="Black" Click="ButNew\_Click"/>

<MenuItem x:Name="butSave" Header="Save" Height="20" Width="50" BorderBrush="Black" Click="butSave\_Click"/>

<MenuItem x:Name="butLoad" Header="Load" Height="20" Width="50" BorderBrush="Black" Click="butLoad\_Click"/>

<MenuItem x:Name="butExport" Header="Export" Height="20" Width="50" BorderBrush="Black" Click="butExport\_Click"/>

</Menu>

<!--Tab control for the three main parts of the program-->

<TabControl x:Name="tabScreens" HorizontalAlignment="Left" Height="511" Margin="0,20,0,0" VerticalAlignment="Top" Width="994" Background="{x:Null}">

<!--Main circuit grid tab item-->

<TabItem Header="Circuit" Background="#FFC8C8C8" BorderBrush="Black" Margin="0,0,-4,-2">

<Grid x:Name="grdMain" Background="#FFE5E5E5" Margin="0,0,-3,-1">

<!--List of gates-->

<ListBox x:Name="gateList" HorizontalAlignment="Left" Height="215" Margin="805,2,0,0" VerticalAlignment="Top" Width="186" Background="#FF686868" BorderBrush="Black" RenderTransformOrigin="0.508,0.505" SelectedIndex="-1">

<ListBoxItem Content="AND" Background="#FFA2A2A2" BorderBrush="Black" Selected="ListBoxItem\_Selected\_AND" Height="30"/>

<ListBoxItem Content="OR" Background="#FFA2A2A2" BorderBrush="Black" Selected="ListBoxItem\_Selected\_OR" Height="30"/>

<ListBoxItem Content="XOR" Background="#FFA2A2A2" BorderBrush="Black" Selected="ListBoxItem\_Selected\_XOR" Height="30"/>

<ListBoxItem Content="NOT" Background="#FFA2A2A2" BorderBrush="Black" Selected="ListBoxItem\_Selected\_NOT" Height="30"/>

<ListBoxItem Content="NOR" Background="#FFA2A2A2" BorderBrush="Black" Selected="ListBoxItem\_Selected\_NOR" Height="30"/>

<ListBoxItem Content="NAND" Background="#FFA2A2A2" BorderBrush="Black" Selected="ListBoxItem\_Selected\_NAND" Height="30"/>

<ListBoxItem Content="XNOR" Background="#FFA2A2A2" BorderBrush="Black" Selected="ListBoxItem\_Selected\_XNOR" Height="30"/>

</ListBox>

<!--Rectangle for information boxes-->

<Rectangle Fill="#FFA2A2A2" HorizontalAlignment="Left" Height="193" Margin="805,222,0,0" VerticalAlignment="Top" Width="186"/>

<!--Movement buttons-->

<Button x:Name="butUp" Content="↑" HorizontalAlignment="Left" Height="32" Margin="837,420,0,0" VerticalAlignment="Top" Width="122" Cursor="ScrollN" Click="butUp\_Click" FontSize="22"/>

<Button x:Name="butDown" Content="↓" HorizontalAlignment="Left" Height="32" Margin="837,452,0,0" VerticalAlignment="Top" Width="122" Cursor="ScrollS" Click="butDown\_Click" FontSize="22"/>

<Button x:Name="butLeft" Content="←" HorizontalAlignment="Left" Height="64" Margin="805,420,0,0" VerticalAlignment="Top" Width="32" Cursor="ScrollW" Click="butLeft\_Click" FontSize="22"/>

<Button x:Name="butRight" Content="→" HorizontalAlignment="Left" Height="64" Margin="959,420,0,0" VerticalAlignment="Top" Width="32" Cursor="ScrollE" Click="butRight\_Click" FontSize="22"/>

<!--Main circuit creation canvas-->

<Canvas x:Name="cnvMain" HorizontalAlignment="Left" Height="480" VerticalAlignment="Top" Width="800" ClipToBounds="True"/>

<!--List of components-->

<ListBox x:Name="compList" HorizontalAlignment="Left" Height="64" Margin="713,5,0,0" VerticalAlignment="Top" Width="82" Background="#FFCFCFCF" BorderBrush="Black">

<ListBoxItem Content="WIRE" BorderBrush="Black" Selected="ListBoxItem\_Selected\_WIRE"/>

<ListBoxItem Content="PIN" BorderBrush="Black" Selected="ListBoxItem\_Selected\_PIN"/>

<ListBoxItem Content="LABEL" BorderBrush="Black" Selected="ListBoxItem\_Selected\_LABEL"/>

</ListBox>

<!--Information boxes-->

<Label x:Name="labComponentName" Content="" HorizontalAlignment="Left" Height="33" Margin="808,226,0,0" VerticalAlignment="Top" Width="179" Background="#FFE5E5E5" BorderBrush="Black" FontSize="18" FontWeight="Bold"/>

<Label x:Name="labComponentState1" Content="INPUT 1" HorizontalAlignment="Left" Height="33" Margin="808,264,0,0" VerticalAlignment="Top" Width="88" Background="#FFE5E5E5" BorderBrush="Black" FontSize="18"/>

<Label x:Name="labComponentState2" Content="INPUT 2" HorizontalAlignment="Left" Height="33" Margin="899,264,0,0" VerticalAlignment="Top" Width="88" Background="#FFE5E5E5" BorderBrush="Black" FontSize="18"/>

<Label x:Name="labComponentIndex" Content="" HorizontalAlignment="Left" Height="33" Margin="808,378,0,0" VerticalAlignment="Top" Width="88" Background="#FFE5E5E5" BorderBrush="Black" FontSize="18"/>

<Label x:Name="labComponentOutput" Content="OUTPUT" HorizontalAlignment="Left" Height="33" Margin="808,302,0,0" VerticalAlignment="Top" Width="179" Background="#FFE5E5E5" BorderBrush="Black" FontSize="18"/>

<!--Delete button-->

<Button x:Name="butDelete" Content="Delete" HorizontalAlignment="Left" Height="33" Margin="899,378,0,0" VerticalAlignment="Top" Width="88" Click="ButDelete\_Click" Background="#FFFFC0CB"/>

</Grid>

</TabItem>

<!--Boolean interface tab item-->

<TabItem Header="Boolean" Background="#FFC8C8C8" BorderBrush="Black" Margin="5,0,-9,-2">

<Grid x:Name="grdBoolean" Background="#FFE5E5E5">

<!--Input, output and notation boxes-->

<Rectangle Fill="#FFA2A2A2" HorizontalAlignment="Left" Height="406" Margin="10,10,0,0" Stroke="Black" VerticalAlignment="Top" Width="306" Grid.RowSpan="2"/>

<Rectangle Fill="#FFA2A2A2" HorizontalAlignment="Left" Height="406" Margin="321,10,0,0" Stroke="Black" VerticalAlignment="Top" Width="306" Grid.RowSpan="2"/>

<Label Content="BOOLEAN INPUT" HorizontalAlignment="Left" Height="45" Margin="10,11,0,0" VerticalAlignment="Top" Width="272" FontSize="16" FontWeight="Bold" FontFamily="Myriad Pro"/>

<Label Content="BOOLEAN OUTPUT" HorizontalAlignment="Left" Height="45" Margin="321,11,0,0" VerticalAlignment="Top" Width="272" FontSize="16" FontWeight="Bold" FontFamily="Myriad Pro"/>

<Label Content="Inputs:" HorizontalAlignment="Left" Height="27" Margin="10,38,0,0" VerticalAlignment="Top" Width="46" FontFamily="Myriad Pro"/>

<Rectangle Fill="#FFA2A2A2" HorizontalAlignment="Left" Height="79" Margin="632,10,0,0" Stroke="Black" VerticalAlignment="Top" Width="150"/>

<!--Number of inputs slider-->

<Slider x:Name="sliNumInputs" HorizontalAlignment="Left" Height="25" Margin="56,40,0,0" VerticalAlignment="Top" Width="107" ValueChanged="SliNumInputs\_ValueChanged" Maximum="3" SmallChange="1"/>

<!--Number of inputs label-->

<Label x:Name="labNumInputs" Content="1" HorizontalAlignment="Left" Height="27" Margin="163,36,0,0" VerticalAlignment="Top" Width="17" Background="#FFC8C8C8"/>

<!--Input expression canvas-->

<Canvas x:Name="cnvBoolInput" HorizontalAlignment="Left" Height="39" Margin="16,70,0,0" VerticalAlignment="Top" Width="293" Background="White" OpacityMask="Black" ClipToBounds="True"/>

<Border BorderBrush="Black" BorderThickness="1" Height="39" Width="293" Margin="16,70,679,374"/>

<!--Simplified output canvas-->

<Canvas x:Name="cnvSimplified" HorizontalAlignment="Left" Height="39" Margin="327,163,0,0" VerticalAlignment="Top" Width="293" Background="White" OpacityMask="Black" ClipToBounds="True"/>

<Border BorderBrush="Black" BorderThickness="1" Height="39" Width="293" Margin="327,163,368,281"/>

<!--Unsimplified output canvas-->

<Canvas x:Name="cnvOutput" HorizontalAlignment="Left" Height="39" Margin="327,101,0,0" VerticalAlignment="Top" Width="293" Background="White" OpacityMask="Black" ClipToBounds="True"/>

<Border BorderBrush="Black" BorderThickness="1" Height="39" Width="293" Margin="327,101,368,343"/>

<!--Input keyboard buttons-->

<Button x:Name="butNOT" Content="NOT" HorizontalAlignment="Left" Height="30" Margin="16,114,0,0" VerticalAlignment="Top" Width="46" BorderBrush="Black" Click="butNOT\_Click"/>

<Button x:Name="butAND" Content="AND" HorizontalAlignment="Left" Height="30" Margin="67,114,0,0" VerticalAlignment="Top" Width="46" BorderBrush="Black" Click="ButAND\_Click"/>

<Button x:Name="butOR" Content="OR" HorizontalAlignment="Left" Height="30" Margin="118,114,0,0" VerticalAlignment="Top" Width="46" BorderBrush="Black" Click="ButOR\_Click"/>

<Button x:Name="butXOR" Content="XOR" HorizontalAlignment="Left" Height="30" Margin="169,114,0,0" VerticalAlignment="Top" Width="46" BorderBrush="Black" Click="ButXOR\_Click"/>

<Button x:Name="butOBracket" Content="(" HorizontalAlignment="Left" Height="30" Margin="220,114,0,0" VerticalAlignment="Top" Width="42" BorderBrush="Black" Click="ButOBracket\_Click"/>

<Button x:Name="butCBracket" Content=")" HorizontalAlignment="Left" Height="30" Margin="267,114,0,0" VerticalAlignment="Top" Width="42" BorderBrush="Black" Click="ButCBracket\_Click"/>

<Button x:Name="butClear" Content="CLEAR" HorizontalAlignment="Left" Height="37" Margin="267,149,0,0" VerticalAlignment="Top" Width="42" BorderBrush="Black" Click="ButtonClear\_Click"/>

<!--Number of input components labels-->

<Label x:Name="labNumGatesNew" Content="Number of Gates: " HorizontalAlignment="Left" Height="27" Margin="17,304,0,0" VerticalAlignment="Top" Width="147" Background="#FFC8C8C8"/>

<Label x:Name="labNumWiresNew" Content="Number of Wires: " HorizontalAlignment="Left" Height="27" Margin="17,336,0,0" VerticalAlignment="Top" Width="147" Background="#FFC8C8C8"/>

<!--Create circuit button-->

<Button x:Name="butCircuit" Content="Create Circuit" HorizontalAlignment="Left" Height="38" Margin="16,368,0,0" VerticalAlignment="Top" Width="293" BorderBrush="Black" Click="ButCircuit\_Click"/>

<!--Canvas labels-->

<Label Content="Output:" HorizontalAlignment="Left" Height="27" Margin="327,79,0,0" VerticalAlignment="Top" Width="59" FontFamily="Myriad Pro"/>

<Label Content="Simplified Output:" HorizontalAlignment="Left" Height="27" Margin="327,141,0,0" VerticalAlignment="Top" Width="110" FontFamily="Myriad Pro"/>

<!--Number of output components labels-->

<Label x:Name="labNumGatesCurrent" Content="Number of Gates: " HorizontalAlignment="Left" Height="27" Margin="327,304,0,0" VerticalAlignment="Top" Width="147" Background="#FFC8C8C8"/>

<Label x:Name="labNumWiresCurrent" Content="Number of Wires: " HorizontalAlignment="Left" Height="27" Margin="327,336,0,0" VerticalAlignment="Top" Width="147" Background="#FFC8C8C8"/>

<Label x:Name="labNumPinsCurrent" Content="Number of Pins: " HorizontalAlignment="Left" Height="27" Margin="327,272,0,0" VerticalAlignment="Top" Width="147" Background="#FFC8C8C8"/>

<!--Create expression button-->

<Button x:Name="butExpression" Content="Create Expression" HorizontalAlignment="Left" Height="38" Margin="327,368,0,0" VerticalAlignment="Top" Width="293" BorderBrush="Black" Click="butExpression\_Click"/>

<!--Notation buttons-->

<Button x:Name="butALEVEL" Content="A LEVEL" HorizontalAlignment="Left" Height="38" Margin="634,49,0,0" VerticalAlignment="Top" Width="72" Background="LightGray" Click="ButtonALEVEL\_Click"/>

<Button x:Name="butGCSE" Content="GCSE" HorizontalAlignment="Left" Height="38" Margin="708,49,0,0" VerticalAlignment="Top" Width="72" Background="DarkGray" Click="ButGCSE\_Click"/>

<!--NOtation label-->

<Label Content="NOTATION" HorizontalAlignment="Left" Height="34" Margin="659,10,0,0" VerticalAlignment="Top" Width="150" FontSize="16" FontWeight="Bold" FontFamily="Myriad Pro"/>

<!--Output pin combo box-->

<ComboBox x:Name="listOutputPinsBoolean" HorizontalAlignment="Left" Height="30" Margin="414,38,0,0" VerticalAlignment="Top" Width="44" BorderBrush="White" Background="White" Foreground="Black"/>

<Label Content="Output Pin:" HorizontalAlignment="Left" Height="30" Margin="327,38,0,0" VerticalAlignment="Top" Width="87" FontWeight="Bold" FontSize="14"/>

</Grid>

</TabItem>

<!--Truth table interface tab item-->

<TabItem Header="Truth Table" Background="#FFC8C8C8" BorderBrush="Black" Margin="10,0,-14,-2" >

<Grid Background="#FFE5E5E5">

<!--Buttons box-->

<Rectangle Fill="#FFA2A2A2" HorizontalAlignment="Left" Height="140" Margin="793,4,0,0" VerticalAlignment="Top" Width="192" Stroke="Black"/>

<!--Output pin combo box-->

<ComboBox x:Name="listOutputPinsTruth" HorizontalAlignment="Left" Height="30" Margin="934,10,0,0" VerticalAlignment="Top" Width="44" BorderBrush="White" Background="White" Foreground="Black"/>

<!--Truth table canvas-->

<Canvas x:Name="cnvTruth" HorizontalAlignment="Left" Height="470" Margin="7,10,0,0" VerticalAlignment="Top" Width="880"/>

<!--Generate table button-->

<Button x:Name="butTruth" Content="Generate Table" HorizontalAlignment="Left" Height="44" Margin="892,52,0,0" VerticalAlignment="Top" Width="89" Click="ButTruth\_Click"/>

<!--Output pin label-->

<Label Content="Select Output Pin:" HorizontalAlignment="Left" Height="30" Margin="803,10,0,0" VerticalAlignment="Top" Width="131" FontWeight="Bold" FontSize="14"/>

<!--Export button-->

<Button x:Name="butTableExport" Content="Export" HorizontalAlignment="Left" Height="36" Margin="892,101,0,0" VerticalAlignment="Top" Width="89" Click="butTableExport\_Click"/>

</Grid>

</TabItem>

</TabControl>

</Grid>

</Window>

## MainWindow

The graphical aspects of the program are handled in the MainWindow partial class that is associated directly with the program’s XAML.

public partial class MainWindow : Window

{

private int selection;

private Path[] currentComponent;

private bool currentNot;

private Circuit mainCircuit;

private bool drawingWire;

private PathGeometry entireWire;

private int[] savedCoords;

private Wire currentWire;

private Path[] wireHead;

private List<Path> gatePaths;

private List<Path> wirePaths;

private List<Path> pinPaths;

private List<TextBox> labels;

private int selectedIndex;

private Button[] infoButtons;

private Button[] inputButtons;

private int selectedComponentType;

private int numInputs;

private Expression mainExpression;

private bool GCSESelection;

private bool deleteMode;

private int highlightedIndex;

private int[] totalOffset;

private bool failedConstruction;

public MainWindow()

{

InitializeComponent();

drawCanvas();

gatePaths = new List<Path>();

wirePaths = new List<Path>();

pinPaths = new List<Path>();

labels = new List<TextBox>();

mainCircuit = new Circuit();

inputButtons = new Button[4];

infoButtons = new Button[2];

savedCoords = new int[2];

totalOffset = new int[2];

selectedComponentType = -1; // 0 Gate, 1 Wire, 2 Pin

DispatcherTimer timer = new DispatcherTimer();

timer.Tick += timerTick;

timer.Interval = new TimeSpan(0, 0, 0, 0, 1);

timer.Start();

selectedIndex = -1;

wireHead = new Path[2];

entireWire = new PathGeometry();

selection = -1;

currentComponent = new Path[2];

deleteMode = false;

highlightedIndex = -1;

failedConstruction = false;

mainExpression = new Expression();

addNewInputButton(1); // Add the 'A' input button for the boolean

}

Canvas Interactions

These methods handle the interactions with the main circuit canvas. This includes drawing ghosts under the cursor, handling users clicking on the canvas, selection and deletion.

#region Canvas Interactions

private void timerTick(object sender, EventArgs e) // The timer handles visual updates that need to happen as the user moves their mouse over the canvas

{

if(Mouse.GetPosition(cnvMain).X < cnvMain.Width && Mouse.GetPosition(cnvMain).Y < cnvMain.Height) // If the mouse is over the canvas, draw a ghost

{

int[] mouseCoords = new int[] { (((int)Mouse.GetPosition(cnvMain).X / 20) \* 20), (((int)Mouse.GetPosition(cnvMain).Y / 20) \* 20) }; // The coordinates of the mouse snapped to the grid

cnvMain.Children.Remove(currentComponent[0]); // Remove the previous ghost

currentComponent[0] = currentComponent[1]; // Delete the previous ghost from the array

for (int i = 0; i < gatePaths.Count; i++) // Components sometimes get stuck red when moused over during deletion, these loops correct this if it arises

{

if (gatePaths[i].Stroke == Brushes.Red) // If a gate is coloured red still, colour it black

{

gatePaths[i].Stroke = Brushes.Black;

}

}

for(int i = 0; i < pinPaths.Count; i++)

{

if(pinPaths[i].Stroke == Brushes.Red) // If a pin is coloured red still, colour it black

{

pinPaths[i].Stroke = Brushes.Black;

}

}

for(int i = 0; i < wirePaths.Count; i++)

{

if(wirePaths[i].Stroke == Brushes.Red) // If a wire is coloured red still, parse the circuit to reset its colour to blue or aqua

{

parseCircuit();

}

}

if (deleteMode)

{

for (int i = 0; i < mainCircuit.getNum(0); i++) // For loop runs for the number of gates in the circuit

{

if (mainCircuit.getGate(i).getLocation()[0] <= mouseCoords[0] && mainCircuit.getGate(i).getLocation()[0] + 80 >= mouseCoords[0] && mainCircuit.getGate(i).getLocation()[1] + 60 >= mouseCoords[1] && mainCircuit.getGate(i).getLocation()[1] <= mouseCoords[1])// If this gate is moused over

{

gatePaths[i].Stroke = Brushes.Red; // Colour the gate red and record which gate is highlighted

highlightedIndex = i;

selectedComponentType = 0;

break;

}

else if(highlightedIndex > -1 && selectedComponentType == 0) // If a different gate is highlighted

{

gatePaths[highlightedIndex].Stroke = Brushes.Black; // Unhighlight the gate and remove the highlighted gate's index

selectedComponentType = -1;

highlightedIndex = -1;

}

}

for (int i = 0; i < mainCircuit.getNum(2); i++) // Loop through the pins

{

if (mouseCoords[0] == mainCircuit.getPin(i).getLocation()[0] && (mouseCoords[1] == mainCircuit.getPin(i).getLocation()[1] || mouseCoords[1] == mainCircuit.getPin(i).getLocation()[1] + 20)) // If this pin is moused over

{

pinPaths[i].Stroke = Brushes.Red; // Colour the pin red and record which pin is highlighted

highlightedIndex = i;

selectedComponentType = 2;

break;

}

else if(highlightedIndex > -1 && selectedComponentType == 2) // If a different pin is highlighted

{

pinPaths[highlightedIndex].Stroke = Brushes.Black; // Unhighlight the pin and remove the pin's index

selectedComponentType = -1;

highlightedIndex = -1;

}

}

for(int i = 0; i < mainCircuit.getNum(1); i++) // Loop through the wires

{

if(mainCircuit.checkIfOnWire(i, mouseCoords)) // If this wire is moused over

{

wirePaths[i].Stroke = Brushes.Red; // Colour the wire red and record the wire's index

highlightedIndex = i;

selectedComponentType = 1;

break;

}

else if(highlightedIndex > -1 && selectedComponentType == 1) // If a different wire is highlighted

{

parseCircuit(); // Parse the circuit to set the wire colour back to blue or aqua and remove the wire's index

selectedComponentType = -1;

highlightedIndex = -1;

}

}

for(int i = 0; i < labels.Count; i++) // Loop through the labels

{

if (Canvas.GetLeft(labels[i]) <= mouseCoords[0] && Canvas.GetLeft(labels[i]) + labels[i].ActualWidth >= mouseCoords[0] && Canvas.GetTop(labels[i]) + 5 == mouseCoords[1]) // If a label is moused over

{

labels[i].Background = Brushes.Red; // Set the label's background to red and record the label's index

highlightedIndex = i;

selectedComponentType = 3;

break;

}

else if (highlightedIndex > -1 && selectedComponentType == 3) // If a different label is highlighted

{

labels[highlightedIndex].Background = Brushes.Transparent; // Clear the background of the label and remove the label's index

selectedComponentType = -1;

highlightedIndex = -1;

}

}

}

else if (drawingWire) // Create a wire head ghost and a wire ghost

{

cnvMain.Children.Remove(wireHead[0]); // Remove the previous wire head ghost

wireHead[0] = wireHead[1]; // Delete the previous wire head ghost from the array

wireHead[1] = new Path(); // Create a ghost of the head of the wire

wireHead[1].Data = new RectangleGeometry(new Rect(new Point(mouseCoords[0] - 1, mouseCoords[1] - 1), new Size(2, 2)));

wireHead[1].Stroke = Brushes.Gray;

wireHead[1].StrokeThickness = 3;

cnvMain.Children.Add(wireHead[1]); // Add the ghost to the canvas

currentComponent[1] = new Path(); // Create a wire ghost

currentComponent[1].Data = entireWire;

currentComponent[1].Stroke = Brushes.Gray;

cnvMain.Children.Add(currentComponent[1]); // Add the ghost to the array

}

else if (gateList.SelectedIndex > -1) // Create gate ghost

{

currentComponent[1] = drawGate(mouseCoords, selection, currentNot); // Draw a new ghost

currentComponent[1].Stroke = Brushes.Gray;

currentComponent[1].StrokeThickness = 3;

cnvMain.Children.Add(currentComponent[1]);

}

else if (selection == 0) // Create wire start point ghost

{

currentComponent[1] = new Path();

currentComponent[1].Data = new RectangleGeometry(new Rect(new Point(mouseCoords[0] - 1, mouseCoords[1] - 1), new Size(2, 2)));

currentComponent[1].Stroke = Brushes.Gray;

currentComponent[1].StrokeThickness = 3;

cnvMain.Children.Add(currentComponent[1]);

}

else if (selection == 1) // Create pin ghost

{

currentComponent[1] = new Path();

currentComponent[1].Data = new RectangleGeometry(new Rect(new Point(mouseCoords[0], mouseCoords[1] + 10), new Size(20, 20)));

currentComponent[1].Stroke = Brushes.Gray;

currentComponent[1].StrokeThickness = 3;

cnvMain.Children.Add(currentComponent[1]);

}

}

}

private void Window\_MouseRightbuttonDown(object sender, MouseButtonEventArgs e)

{

if (drawingWire) // If the user is drawing a wire

{

Path wirePath = new Path(); // Create a new path to hold the finished wire

wirePath.Data = entireWire; // Set the path data

wirePath.Stroke = Brushes.Blue;

cnvMain.Children.Add(wirePath); // Add the whole wire to the canvas

if (!(currentWire.getTurnPoints().Count == 0 && currentWire.getEnd()[0] == 0 && currentWire.getEnd()[1] == 0))

{

currentWire.finishWire(); // Set the endpoint of the wire

mainCircuit.addWire(currentWire); // Add the wire to the circuit

wirePaths.Add(wirePath);

}

cnvMain.Children.Remove(wireHead[0]); // Remove the previous ghost of the wire head

cnvMain.Children.Remove(wireHead[1]); // Remove the current ghost of the wire head

entireWire = new PathGeometry(); // Reset entire wire

}

drawingWire = false; // Stop drawing the wire

compList.SelectedIndex = -1; // Reset the selection of each list

gateList.SelectedIndex = -1;

selection = -1; // Reset selection

parseCircuit(); // Parse any changes to the circuit

}

private void Window\_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)

{

if (Mouse.GetPosition(cnvMain).X < cnvMain.Width && Mouse.GetPosition(cnvMain).Y < cnvMain.Height) // If the mouse is over the canvas when the user clicks

{

int[] mouseCoords = new int[] { (((int)Mouse.GetPosition(cnvMain).X / 20) \* 20), (((int)Mouse.GetPosition(cnvMain).Y / 20) \* 20) }; // Snap the mouse coords to the grid

if (deleteMode) // If currently in delete mode

{

if (highlightedIndex > -1 && selectedComponentType == 0)// If a gate is clicked on, delete it

{

deselectComponent(); // Deselect any component

cnvMain.Children.Remove(gatePaths[highlightedIndex]); // Remove the gate from the canvas

gatePaths.RemoveAt(highlightedIndex); // Remove the gate's path from the list

mainCircuit.getGate(highlightedIndex).reduceNextIndex(); // Reduce the next index of the gates

mainCircuit.removeComponent(0, highlightedIndex); // Remove the gate from the circuit

mainCircuit.reduceGateIndexes(highlightedIndex); // Reduce the gate indexes above the removed gate

parseCircuit(); // Parse any changes to the circuit

highlightedIndex = -1; // Reset the highlighted index

}

else if (highlightedIndex > -1 && selectedComponentType == 2)// If a pin is clicked on, delete it

{

if (mainCircuit.getPin(highlightedIndex).getIfOutput())// If the pin is an ouput pin, remove it from the lists of output pins

{

listOutputPinsTruth.Items.Remove(mainCircuit.getPin(highlightedIndex).getChar());

listOutputPinsBoolean.Items.Remove(mainCircuit.getPin(highlightedIndex).getChar());

}

deselectComponent(); // Deselect any component

cnvMain.Children.Remove(pinPaths[highlightedIndex]); // Remove the pin from the canvas

pinPaths.RemoveAt(highlightedIndex); // Remove the pin's path from the list

mainCircuit.getPin(highlightedIndex).addSpareChar(); // Add the pin's char to the list of spare chars

mainCircuit.getPin(highlightedIndex).reduceNextIndex(); // Reduce the next index of the pins

mainCircuit.removeComponent(2, highlightedIndex); // Remove the pin from the circuit

mainCircuit.reducePinIndexes(highlightedIndex); // Reduce the pin indexes above the removed pin

parseCircuit(); // Parse any changes to the circuit

highlightedIndex = -1; // Reset the highlighted index

}

else if (highlightedIndex > -1 && selectedComponentType == 1)// If a wire is clicked on, delete it

{

deselectComponent(); // Deselect any component

cnvMain.Children.Remove(wirePaths[highlightedIndex]); // Remove the wire from the canvas

wirePaths.RemoveAt(highlightedIndex); // Remove the wire's path from the list

mainCircuit.getWire(highlightedIndex).reduceNextIndex(); // Reduce the next index of the wires

mainCircuit.removeComponent(1, highlightedIndex); // Remove the component from the list

parseCircuit(); // Parse any changes to the circuit

highlightedIndex = -1; // Reset the highlighted index

}

else if (highlightedIndex > -1 && selectedComponentType == 3)// If a label is clicked on, delete it

{

cnvMain.Children.Remove(labels[highlightedIndex]); // the label from the canvas

labels.RemoveAt(highlightedIndex); // Remove the label from the list

highlightedIndex = -1; // Reset the highlighted index

}

}

else if (drawingWire) // If a wire is currently being drawn

{

PathFigureCollection thisStep = drawWireLine(savedCoords, mouseCoords); // Get the latest segment of the wire

for (int i = 0; i < 2; i++)

{

entireWire.Figures.Add(thisStep[i]); // Add the segments

}

savedCoords = mouseCoords; // Save the coords of the turnpoint

currentWire.addTurnPoint(mouseCoords); // Add the turnpoint at the mouse location

}

else if (gateList.SelectedIndex > -1) // If a gate is selected

{

createNewGate(mouseCoords, selection, currentNot);

selection = -1; // Reset the selection

currentNot = false;

}

else if (selection == 0) // If wire is selected

{

drawingWire = true; // A wire is now being drawn

currentWire = new Wire(mouseCoords); // Create a new wire to add to the circuit

savedCoords = mouseCoords; // Save the coords

}

else if (selection == 1) // If pin is selected

{

Path thisPin = drawPin(mouseCoords, true); // Draw the new pin

mainCircuit.addPin(new Pin(mouseCoords));

pinPaths.Add(thisPin); // Add the pin to the list of paths

cnvMain.Children.Add(thisPin); // Add the pin to the grid

Canvas.SetLeft(thisPin, mouseCoords[0]);

Canvas.SetTop(thisPin, mouseCoords[1] + 10); // Offset the pin

selection = -1; // Reset the selection

}

else if(selection == 2) // If a label is selected

{

TextBox thisLabel = new TextBox(); // Create a new textbox

thisLabel.FontWeight = FontWeights.ExtraBold;

thisLabel.FontSize = 20;

thisLabel.Background = Brushes.Transparent;

thisLabel.BorderBrush = Brushes.Transparent;

thisLabel.Text = " "; // Set the content of the label

thisLabel.SelectAll();

labels.Add(thisLabel); // Add the label to the canvas

cnvMain.Children.Add(thisLabel);

Canvas.SetLeft(thisLabel, mouseCoords[0]);

Canvas.SetTop(thisLabel, mouseCoords[1] - 5);

selection = -1;

}

else if (selection == -1) // If nothing is selected

{

deselectComponent(); // Deselect the previous component

selectedIndex = selectComponent(mouseCoords); // Select the new component

}

else // To catch any unforseen actions

{

deselectComponent(); // Deselect the previous component

selectedIndex = -1; // Reset the selection

}

parseCircuit(); // Evaluate any changes to the circuit

compList.SelectedIndex = -1; // Reset the selection of each list

gateList.SelectedIndex = -1;

}

}

private void createNewGate(int[] location, int selection, bool isNot)

{

Path objectToDraw = drawGate(location, selection, isNot); // Get the gate to draw

gatePaths.Add(objectToDraw); // The gate's index is determined by the order it was drawn in so it will have the same index in this list

cnvMain.Children.Add(objectToDraw); // Draw the gate

mainCircuit.addGate(new Gate(location, selection, isNot)); // Add the new gate to the circuit

}

private int selectComponent(int[] mouseCoords)

{

for (int i = 0; i < mainCircuit.getNum(0); i++) // For loop runs for the number of gates in the circuit

{

if (mainCircuit.getGate(i).getLocation()[0] <= mouseCoords[0] && mainCircuit.getGate(i).getLocation()[0] + 80 >= mouseCoords[0] && mainCircuit.getGate(i).getLocation()[1] + 60 >= mouseCoords[1] && mainCircuit.getGate(i).getLocation()[1] <= mouseCoords[1])// If a gate is clicked on

{

gatePaths[i].Stroke = Brushes.Blue; // Outline the selected gate in blue

selectedComponentType = 0;

labComponentIndex.Content = mainCircuit.getGate(i).getIndex(); // Display the index of the gate

switch (mainCircuit.getGate(i).getGateType()) // Display the type of gate in the correct label

{

case 0:

if (mainCircuit.getGate(i).getNot())

{

labComponentName.Content = "NAND";

}

else

{

labComponentName.Content = "AND";

}

break;

case 1:

if (mainCircuit.getGate(i).getNot())

{

labComponentName.Content = "NOR";

}

else

{

labComponentName.Content = "OR";

}

break;

case 2:

if (mainCircuit.getGate(i).getNot())

{

labComponentName.Content = "XNOR";

}

else

{

labComponentName.Content = "XOR";

}

break;

case 3:

labComponentName.Content = "NOT";

break;

}

if (mainCircuit.getGate(i).getInput(0)) // Set the first state label to the gate's state

{

labComponentState1.Content = "TRUE";

}

else

{

labComponentState1.Content = "FALSE";

}

if (mainCircuit.getGate(i).getInput(1) && mainCircuit.getGate(i).getGateType() != 3) // Set the second state label to the gate's state. NOT gates only have 1 state

{

labComponentState2.Content = "TRUE";

}

else if(mainCircuit.getGate(i).getGateType() != 3)

{

labComponentState2.Content = "FALSE";

}

if (mainCircuit.getGate(i).getState()) // Set the output state label to the gate's output state

{

labComponentOutput.Content = "TRUE";

}

else

{

labComponentOutput.Content = "FALSE";

}

return mainCircuit.getGate(i).getIndex(); // Return the index of the gate

}

}

for(int i = 0; i < mainCircuit.getNum(2); i++)

{

if (mouseCoords[0] == mainCircuit.getPin(i).getLocation()[0] && (mouseCoords[1] == mainCircuit.getPin(i).getLocation()[1] || mouseCoords[1] == mainCircuit.getPin(i).getLocation()[1] + 20)) // Check if a pin exists where the mouse has clicked

{

pinPaths[i].Stroke = Brushes.Blue; // Outline the selected pin in blue

labComponentName.Content = "Pin " + mainCircuit.getPin(i).getChar(); // Set the name label to the char of the pin

labComponentIndex.Content = mainCircuit.getPin(i).getIndex(); // Display the index of the gate

selectedComponentType = 2;

if (mainCircuit.getPin(i).getState())

{

labComponentState1.Content = "TRUE";

}

else

{

labComponentState1.Content = "FALSE";

}

if (mainCircuit.getPin(i).getIfOutput())

{

labComponentOutput.Content = "FUNCTION: OUTPUT";

}

else

{

labComponentOutput.Content = "FUNCTION: INPUT";

}

labComponentState2.Content = "";

Button thisButton = new Button(); // Create a new button

if (!mainCircuit.getPin(i).getIfOutput()) // If the pin is not an output pin

{

thisButton.Width = 88; // Add the state button

thisButton.Height = 33;

grdMain.Children.Add(thisButton);

thisButton.Margin = new Thickness(713, 230, 0, 0);

thisButton.Click += State\_Button\_Click;

thisButton.Content = "STATE";

infoButtons[0] = thisButton; // Add the button to the array

thisButton = new Button(); // Reset thisButton

}

thisButton.Width = 88; // Add the function button

thisButton.Height = 33;

grdMain.Children.Add(thisButton);

thisButton.Margin = new Thickness(895, 230, 0, 0);

thisButton.Click += Function\_Button\_Click;

thisButton.Content = "FUNCTION";

infoButtons[1] = thisButton; // Add the button to the array

return mainCircuit.getPin(i).getIndex(); // Return the pin index

}

}

return -1; // If nothing was selected, return -1

}

private void ButDelete\_Click(object sender, RoutedEventArgs e)

{

if (deleteMode) // If currently in delete mode

{

deleteMode = false; // Turn off delete mode

butDelete.Content = "Delete"; // Reset the button

butDelete.Background = Brushes.Pink;

deselectComponent(); // Deselect any selected component

for(int i = 0; i < labels.Count; i++) // Allow labels to be interacted with

{

labels[i].IsEnabled = true;

}

}

else // If not in delete mode

{

deleteMode = true; // Turn on delete mode

butDelete.Content = "Cancel"; // Set the button to its cancel state

butDelete.Background = Brushes.Gray;

deselectComponent(); // Deselect any selected component

for (int i = 0; i < labels.Count; i++) // Prevent labels from being interacted with

{

labels[i].IsEnabled = false;

}

}

}

private void State\_Button\_Click(object sender, RoutedEventArgs e)

{

for (int i = 0; i < mainCircuit.getNum(2); i++) // For loop runs for the number of gates in the circuit

{

if (mainCircuit.getPin(i).getIndex() == selectedIndex)

{

mainCircuit.getPin(i).flipState(); // Flip the pin's state

if (mainCircuit.getPin(i).getState()) // Set the state label to the new state of the pin

{

labComponentState1.Content = "TRUE";

}

else

{

labComponentState1.Content = "FALSE";

}

parseCircuit(); // Parse the circuit for any changes made

}

}

}

private void Function\_Button\_Click(object sender, RoutedEventArgs e)

{

for (int i = 0; i < mainCircuit.getNum(2); i++) // For loop runs for the number of gates in the circuit

{

if (mainCircuit.getPin(i).getIndex() == selectedIndex)

{

if (!mainCircuit.getPin(i).getIfOutput())

{

labComponentOutput.Content = "FUNCTION: OUTPUT";

Path newPin = drawPin(mainCircuit.getPin(i).getLocation(), false); // Create a new pin path with the opposite connector

cnvMain.Children.Remove(pinPaths[i]); // Remove the previous pin path

pinPaths[i] = newPin; // Replace the previous path with the new path

cnvMain.Children.Add(newPin); // Add the new path to the grid

mainCircuit.getPin(i).flipOutput(); // Flip the output of the pin in the circuit class

if (mainCircuit.getPin(i).getState()) // If the pin is on, turn it off

{

mainCircuit.getPin(i).flipState();

}

deselectComponent(); // Deselect the 'previous component'

selectedIndex = selectComponent(mainCircuit.getPin(i).getLocation()); // Select the 'new component'

grdMain.Children.Remove(infoButtons[0]);

infoButtons[0] = new Button();

Canvas.SetLeft(newPin, mainCircuit.getPin(i).getLocation()[0]);

Canvas.SetTop(newPin, mainCircuit.getPin(i).getLocation()[1] + 10);

listOutputPinsTruth.Items.Add(mainCircuit.getPin(i).getChar());

listOutputPinsBoolean.Items.Add(mainCircuit.getPin(i).getChar());

}

else

{

labComponentOutput.Content = "FUNCTION: INPUT";

Path newPin = drawPin(mainCircuit.getPin(i).getLocation(), true); // Create a new pin path with the opposite connector

cnvMain.Children.Remove(pinPaths[i]); // Remove the previous pin path

pinPaths[i] = newPin; // Replace the previous path with the new path

cnvMain.Children.Add(newPin); // Add the new path to the grid

mainCircuit.getPin(i).flipOutput(); // Flip the output of the pin in the circuit class

deselectComponent(); // Deselect the 'previous component'

selectedIndex = selectComponent(mainCircuit.getPin(i).getLocation()); // Select the 'new component'

Canvas.SetLeft(newPin, mainCircuit.getPin(i).getLocation()[0]);

Canvas.SetTop(newPin, mainCircuit.getPin(i).getLocation()[1] + 10);

listOutputPinsTruth.Items.Remove(mainCircuit.getPin(i).getChar());

listOutputPinsBoolean.Items.Remove(mainCircuit.getPin(i).getChar());

}

parseCircuit(); // Parse the circuit for any changes made

}

}

}

private void deselectComponent()

{

if(selectedComponentType == 0) // If a gate is selected

{

for (int i = 0; i < mainCircuit.getNum(0); i++) // For loop runs for the number of gates in the circuit

{

if (mainCircuit.getGate(i).getIndex() == selectedIndex)

{

gatePaths[i].Stroke = Brushes.Black; // Set the stroke of the deselected gate back to black

labComponentName.Content = ""; // Set the contents of the labels to their default state

labComponentState1.Content = "INPUT 1";

labComponentState2.Content = "INPUT 2";

labComponentIndex.Content = "";

labComponentOutput.Content = "OUTPUT";

}

}

}

else if(selectedComponentType == 2) // If a pin is selected

{

for (int i = 0; i < mainCircuit.getNum(2); i++) // For loop runs for the number of pins in the circuit

{

if (mainCircuit.getPin(i).getIndex() == selectedIndex)

{

pinPaths[i].Stroke = Brushes.Black; // Set the stroke of the deselected pin back to black

labComponentName.Content = ""; // Set the contents of the labels to their default state

labComponentState1.Content = "INPUT 1";

labComponentState2.Content = "INPUT 2";

labComponentIndex.Content = "";

labComponentOutput.Content = "OUTPUT";

grdMain.Children.Remove(infoButtons[0]); // Remove the buttons used to interact with the pins

grdMain.Children.Remove(infoButtons[1]);

}

}

}

selectedIndex = -1; // Reset selection indexes

selectedComponentType = -1;

}

#endregion

Component Drawing

These methods do all the drawing required for the main canvas to function. They draw the gridlines, the gates, the wires, the pins, the connectors for all components and they handle changing the colours of components as their states are updated.

#region Component Drawing

public void drawCanvas()// Draws the border of the canvas and gridlines on it

{

for (int i = 0; i < 24; i++)//Horizontal Gridlines

{

Rectangle horiLine = new Rectangle();//Define parameters for rectangles making up gridlines

horiLine.Height = 20;

horiLine.Width = cnvMain.Width;

horiLine.Stroke = Brushes.LightGray;//Colour of lines

cnvMain.Children.Add(horiLine);//Add gridlines to canvas

Canvas.SetTop(horiLine, i \* 20);//Place each rectangle 1 rectangle width apart

Canvas.SetLeft(horiLine, 0);

}

for(int i = 0; i < 40; i++)//Vertical Gridlines

{

Rectangle vertiLine = new Rectangle();//Define parameters for rectangles making up gridlines

vertiLine.Height = cnvMain.Height;

vertiLine.Width = 20;

vertiLine.Stroke = Brushes.LightGray;//Colour of lines

cnvMain.Children.Add(vertiLine);//Add gridlines to canvas

Canvas.SetLeft(vertiLine, i \* 20);//Place each rectangle 1 rectangle width apart

Canvas.SetTop(vertiLine, 0);

}

Rectangle canvasBorder = new Rectangle();//Black border for canvas

canvasBorder.Height = cnvMain.Height;

canvasBorder.Width = cnvMain.Width;

canvasBorder.Stroke = Brushes.Black;

canvasBorder.StrokeThickness = 2;

cnvMain.Children.Add(canvasBorder);//Add border to canvas

Canvas.SetTop(canvasBorder, 0);

Canvas.SetLeft(canvasBorder, 0);

}

public PathFigureCollection drawWireLine(int[] startPoint, int[] endPoint)

{

PathFigureCollection wholeWire = new PathFigureCollection();

PathFigure thisPathFigure = new PathFigure(); // Draw the horizontal portion of the wire segment

thisPathFigure.StartPoint = new Point(startPoint[0], startPoint[1]); // Starting at the startpoint

LineSegment thisLine = new LineSegment();

thisLine.Point = new Point(endPoint[0], startPoint[1]); // Ending in the same column as the endpoint but the same row as the startpoint

thisPathFigure.Segments.Add(thisLine);

wholeWire.Add(thisPathFigure);

thisPathFigure = new PathFigure(); // Draw the vertical portion of the wire segment

thisPathFigure.StartPoint = new Point(endPoint[0], startPoint[1]); // Starting where the horizontal portion finished

thisLine = new LineSegment();

thisLine.Point = new Point(endPoint[0], endPoint[1]); // Finishing at the endpoint

thisPathFigure.Segments.Add(thisLine);

wholeWire.Add(thisPathFigure);

return wholeWire;

}

public Path drawGate(int[] coords, int type, bool not) // Provides a Path for the specified gate

{

GeometryGroup finalGate = new GeometryGroup(); // This group will contain the path geometry for the gate and an ellipse geometry for the circle to represent a NOT form of a gate

PathGeometry gateGeometry = new PathGeometry(); // Define store for line segments

PathFigure thisPathFigure = new PathFigure(); // This will be instantiated multiple times for each line

EllipseGeometry NOTCircle = new EllipseGeometry(new Point(coords[0] + 90, coords[1] + 40), 10, 10); // Creating an ellipse geometry that will be added as a NOT circle if one is needed

switch (type)

{

case 0: // AND Gate

thisPathFigure.StartPoint = new Point(coords[0], coords[1]); // Top left point of figure

LineSegment backLineAND = new LineSegment();

backLineAND.Point = new Point(coords[0], coords[1] + 80); // Bottom left point of figure

thisPathFigure.Segments.Add(backLineAND); // Add line to the current pathfigure

gateGeometry.Figures.Add(thisPathFigure); // Add pathfigure to the overall store

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(coords[0], coords[1] + 80); // Bottom left point of figure

LineSegment bottomLine = new LineSegment();

bottomLine.Point = new Point(coords[0] + 50, coords[1] + 80); // Bottom right corner of figure

thisPathFigure.Segments.Add(bottomLine); // Add line to the current pathfigure

gateGeometry.Figures.Add(thisPathFigure); // Add pathfigure to the overall store

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(coords[0], coords[1]); // Top left point of figure

LineSegment topLine = new LineSegment();

topLine.Point = new Point(coords[0] + 50, coords[1]); // Top right corner of figure

thisPathFigure.Segments.Add(topLine); // Add line to the current pathfigure

gateGeometry.Figures.Add(thisPathFigure); // Add pathfigure to the overall store

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(coords[0] + 50, coords[1] + 80); // Bottom right point of figure

ArcSegment rightCurve = new ArcSegment();

rightCurve.Point = new Point(coords[0] + 50, coords[1]); // Top right corner of figure

rightCurve.Size = new Size(30, 40); // Radius of arc

thisPathFigure.Segments.Add(rightCurve); // Add line to the current pathfigure

gateGeometry.Figures.Add(thisPathFigure); // Add pathfigure to the overall store

break;

case 1: // OR Gate

thisPathFigure.StartPoint = new Point(coords[0], coords[1] + 80); // Bottom left point of figure

ArcSegment backCurveOR = new ArcSegment();

backCurveOR.Point = new Point(coords[0], coords[1]); // Top left point of figure

backCurveOR.Size = new Size(15, 40); // Radius of arc

thisPathFigure.Segments.Add(backCurveOR); // Add line to the current pathfigure

gateGeometry.Figures.Add(thisPathFigure); // Add pathfigure to the overall store

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(coords[0], coords[1]); // Top left point of figure

BezierSegment topCurveOR = new BezierSegment();

topCurveOR.Point3 = new Point(coords[0] + 80, coords[1] + 40); // Right edge point of figure

topCurveOR.Point1 = new Point(coords[0] + 30, coords[1] + 6); // First Control Point

topCurveOR.Point2 = new Point(coords[0] + 50, coords[1]); // Second Control Point

thisPathFigure.Segments.Add(topCurveOR); // Add line to the current pathfigure

gateGeometry.Figures.Add(thisPathFigure); // Add pathfigure to the overall store

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(coords[0], coords[1] + 80); // Right edge point of figure

BezierSegment bottomCurveOR = new BezierSegment();

bottomCurveOR.Point3 = new Point(coords[0] + 80, coords[1] + 40); // Bottom left point of figure

bottomCurveOR.Point1 = new Point(coords[0] + 30, coords[1] + 74); // First Control Point

bottomCurveOR.Point2 = new Point(coords[0] + 50, coords[1] + 80); // Second Control Point

thisPathFigure.Segments.Add(bottomCurveOR); // Add line to the current pathfigure

gateGeometry.Figures.Add(thisPathFigure); // Add pathfigure to the overall store

break;

case 2: // XOR Gate

thisPathFigure.StartPoint = new Point(coords[0], coords[1] + 80); // Bottom left point of figure

ArcSegment backCurveXOR = new ArcSegment();

backCurveXOR.Point = new Point(coords[0], coords[1]); // Top left point of figure

backCurveXOR.Size = new Size(15, 40); // Radius of arc

thisPathFigure.Segments.Add(backCurveXOR); // Add line to the current pathfigure

gateGeometry.Figures.Add(thisPathFigure); // Add pathfigure to the overall store

thisPathFigure = new PathFigure();

ArcSegment backDoubleCurveXOR = new ArcSegment();

thisPathFigure.StartPoint = new Point(coords[0] - 10, coords[1] + 80); // Top left point of figure

backDoubleCurveXOR.Point = new Point(coords[0] - 10, coords[1]); // Behind the top left point of figure

backDoubleCurveXOR.Size = new Size(15, 40); // Radius of arc

thisPathFigure.Segments.Add(backDoubleCurveXOR); // Add line to the current pathfigure

gateGeometry.Figures.Add(thisPathFigure); // Add pathfigure to the overall store

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(coords[0], coords[1]); // Top left point of figure

BezierSegment topCurveXOR = new BezierSegment();

topCurveXOR.Point3 = new Point(coords[0] + 80, coords[1] + 40); // Right edge point of figure

topCurveXOR.Point1 = new Point(coords[0] + 30, coords[1] + 6); // First Control Point

topCurveXOR.Point2 = new Point(coords[0] + 50, coords[1]); // Second Control Point

thisPathFigure.Segments.Add(topCurveXOR); // Add line to the current pathfigure

gateGeometry.Figures.Add(thisPathFigure); // Add pathfigure to the overall store

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(coords[0], coords[1] + 80); // Right edge point of figure

BezierSegment bottomCurveXOR = new BezierSegment();

bottomCurveXOR.Point3 = new Point(coords[0] + 80, coords[1] + 40); // Bottom left point of figure

bottomCurveXOR.Point1 = new Point(coords[0] + 30, coords[1] + 74); // First Control Point

bottomCurveXOR.Point2 = new Point(coords[0] + 50, coords[1] + 80); // Second Control Point

thisPathFigure.Segments.Add(bottomCurveXOR); // Add line to the current pathfigure

gateGeometry.Figures.Add(thisPathFigure); // Add pathfigure to the overall store

break;

case 3: // NOT Gate

thisPathFigure.StartPoint = new Point(coords[0], coords[1]); // Top left point of figure

LineSegment backLineNOT = new LineSegment();

backLineNOT.Point = new Point(coords[0], coords[1] + 80); // Bottom left point of figure

thisPathFigure.Segments.Add(backLineNOT); // Add line to the current pathfigure

gateGeometry.Figures.Add(thisPathFigure); // Add pathfigure to the overall store

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(coords[0], coords[1]); // Top left point of figure

LineSegment topDiagonal = new LineSegment();

topDiagonal.Point = new Point(coords[0] + 80, coords[1] + 40); // Rightmost point of figure

thisPathFigure.Segments.Add(topDiagonal); // Add line to the current pathfigure

gateGeometry.Figures.Add(thisPathFigure); // Add pathfigure to the overall store

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(coords[0], coords[1] + 80); // Bottom left point of figure

LineSegment bottomDiagonal = new LineSegment();

bottomDiagonal.Point = new Point(coords[0] + 80, coords[1] + 40); // Rightmost point of figure

thisPathFigure.Segments.Add(bottomDiagonal); // Add line to the current pathfigure

gateGeometry.Figures.Add(thisPathFigure); // Add pathfigure to the overall store

break;

case -1: // No selection

break;

}

if (not) // If gate is a NOT version, add a circle to represent this

{

finalGate.Children.Add(NOTCircle);

}

finalGate.Children.Add(gateGeometry);

finalGate.Children.Add(drawGateConnectors(coords, not, type));

Path drawGate = new Path(); // Create the path to return

drawGate.Stroke = Brushes.Black;

drawGate.StrokeThickness = 2;

drawGate.Data = finalGate; // Add the drawn gate to the Path

return drawGate; // Return the path of the drawn gate

}

private Path drawPin(int[] mouseCoords, bool isInput)

{

GeometryGroup finalPin = new GeometryGroup();

RectangleGeometry thisPin = new RectangleGeometry(); // Create the pin

thisPin.Rect = new Rect(0, 0, 20, 20);

finalPin.Children.Add(thisPin);

finalPin.Children.Add(drawPinConnector(mouseCoords, isInput));

Path drawPin = new Path(); // Draw the new pin

drawPin.Fill = Brushes.LightBlue;

drawPin.Stroke = Brushes.Black;

drawPin.StrokeThickness = 4;

drawPin.Data = finalPin;

return drawPin; // Return the new pin

}

private PathGeometry drawPinConnector(int[] mouseCoords, bool isInput)

{

PathGeometry pinConnectorGeometry = new PathGeometry();

PathFigure thisPathFigure;

if (isInput)

{

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(20, 10); // input connector, coords are relative to position of mouse click

LineSegment inputConnector = new LineSegment(new Point(24, 10), true);

thisPathFigure.Segments.Add(inputConnector);

pinConnectorGeometry.Figures.Add(thisPathFigure);

}

else

{

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(0, 10); // output connector, coords are relative to position of mouse click

LineSegment inputConnector = new LineSegment(new Point(-4, 10), true);

thisPathFigure.Segments.Add(inputConnector);

pinConnectorGeometry.Figures.Add(thisPathFigure);

}

return pinConnectorGeometry; // Return the connector geometry

}

private PathGeometry drawGateConnectors(int[] mouseCoords, bool not, int selectedType) // Draw dots for connection points on gates

{

PathGeometry gateConnectorGeometry = new PathGeometry();

PathFigure thisPathFigure = new PathFigure();

if (selectedType == 3) // If gate is a NOT gate, only one connector is needed for input

{

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(mouseCoords[0] - 4, mouseCoords[1] + 40); // NOT connector

LineSegment bottomConnector = new LineSegment(new Point(mouseCoords[0], mouseCoords[1] + 40), true);

thisPathFigure.Segments.Add(bottomConnector);

gateConnectorGeometry.Figures.Add(thisPathFigure);

}

else if(selectedType == 1)// OR gates use 2 longer connectors as inputs

{

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(mouseCoords[0] - 4, mouseCoords[1] + 20); // Top connector

LineSegment topConnector = new LineSegment(new Point(mouseCoords[0] + 12, mouseCoords[1] + 20), true);

thisPathFigure.Segments.Add(topConnector);

gateConnectorGeometry.Figures.Add(thisPathFigure);

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(mouseCoords[0] - 4, mouseCoords[1] + 60); // Bottom connector

LineSegment bottomConnector = new LineSegment(new Point(mouseCoords[0] + 12, mouseCoords[1] + 60), true);

thisPathFigure.Segments.Add(bottomConnector);

gateConnectorGeometry.Figures.Add(thisPathFigure);

}

else if(selectedType == 2) // XOR gates use 2 slightly longer connectors as inputs

{

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(mouseCoords[0] - 4, mouseCoords[1] + 20); // Top connector

LineSegment topConnector = new LineSegment(new Point(mouseCoords[0] + 4, mouseCoords[1] + 20), true);

thisPathFigure.Segments.Add(topConnector);

gateConnectorGeometry.Figures.Add(thisPathFigure);

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(mouseCoords[0] - 4, mouseCoords[1] + 60); // Bottom connector

LineSegment bottomConnector = new LineSegment(new Point(mouseCoords[0] + 4, mouseCoords[1] + 60), true);

thisPathFigure.Segments.Add(bottomConnector);

gateConnectorGeometry.Figures.Add(thisPathFigure);

}

else // All other gates have the same connectors as inputs

{

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(mouseCoords[0] - 4, mouseCoords[1] + 20); // Top connector

LineSegment topConnector = new LineSegment(new Point(mouseCoords[0], mouseCoords[1] + 20), true);

thisPathFigure.Segments.Add(topConnector);

gateConnectorGeometry.Figures.Add(thisPathFigure);

thisPathFigure = new PathFigure();

thisPathFigure.StartPoint = new Point(mouseCoords[0] - 4, mouseCoords[1] + 60); // Bottom connector

LineSegment bottomConnector = new LineSegment(new Point(mouseCoords[0], mouseCoords[1] + 60), true);

thisPathFigure.Segments.Add(bottomConnector);

gateConnectorGeometry.Figures.Add(thisPathFigure);

}

thisPathFigure = new PathFigure();

LineSegment tipConnector;

if (not) // If the gate is a NOT version, draw the tip connector at the edge of the NOT circle

{

thisPathFigure.StartPoint = new Point(mouseCoords[0] + 104, mouseCoords[1] + 40);

tipConnector = new LineSegment(new Point(mouseCoords[0] + 100, mouseCoords[1] + 40), true);

}

else // If the gate is not a NOT version, draw the tip connector at the tip of the gate

{

thisPathFigure.StartPoint = new Point(mouseCoords[0] + 84, mouseCoords[1] + 40);

tipConnector = new LineSegment(new Point(mouseCoords[0] + 80, mouseCoords[1] + 40), true);

}

thisPathFigure.Segments.Add(tipConnector);

gateConnectorGeometry.Figures.Add(thisPathFigure);

return gateConnectorGeometry; // Return the connector geometry

}

private void parseCircuit()

{

for(int i = 0; i < mainCircuit.getNum(1) \* 2; i++) // For the number of wires in the circuit, evaluate the circuit

{

mainCircuit.evaluateCircuit(); // Evaluating this many times reduces the likelihood of an incorrect state occurring due to a loop

}

for (int j = 0; j < mainCircuit.getNum(1); j++) // Loop through all the wires and set their colour to match their state

{

if (mainCircuit.getWire(j).getState())

{

wirePaths[j].Stroke = Brushes.Aqua; // If the wire is ON, set the colour to aqua

}

else

{

wirePaths[j].Stroke = Brushes.Blue; // If the wire is OFF, set the colour to blue

}

}

for (int i = 0; i < mainCircuit.getNum(2); i++) // Loop through all the pins and set their colour to match their state

{

if (mainCircuit.getPin(i).getState())

{

pinPaths[i].Fill = Brushes.DarkCyan; // If the pin is on, set the colour to dark cyan

}

else

{

pinPaths[i].Fill = Brushes.LightBlue; // If the pin is off, set the colour to light blue

}

}

}

#endregion

ListBox Selection

These events handle what happens when an item in a listbox is selected, namely setting the selection index and currentNot to the correct values. They also set the other list’s selection index to -1 so that nothing is selected.

#region ListBox Selection

private void ListBoxItem\_Selected\_AND(object sender, RoutedEventArgs e)

{

selection = 0;

currentNot = false;

compList.SelectedIndex = -1;

}

private void ListBoxItem\_Selected\_OR(object sender, RoutedEventArgs e)

{

selection = 1;

currentNot = false;

compList.SelectedIndex = -1;

}

private void ListBoxItem\_Selected\_XOR(object sender, RoutedEventArgs e)

{

selection = 2;

currentNot = false;

compList.SelectedIndex = -1;

}

private void ListBoxItem\_Selected\_NOT(object sender, RoutedEventArgs e)

{

selection = 3;

currentNot = true;

compList.SelectedIndex = -1;

}

private void ListBoxItem\_Selected\_XNOR(object sender, RoutedEventArgs e)

{

selection = 2;

currentNot = true;

compList.SelectedIndex = -1;

}

private void ListBoxItem\_Selected\_NAND(object sender, RoutedEventArgs e)

{

selection = 0;

currentNot = true;

compList.SelectedIndex = -1;

}

private void ListBoxItem\_Selected\_NOR(object sender, RoutedEventArgs e)

{

selection = 1;

currentNot = true;

compList.SelectedIndex = -1;

}

private void ListBoxItem\_Selected\_WIRE(object sender, RoutedEventArgs e)

{

selection = 0;

gateList.SelectedIndex = -1;

}

private void ListBoxItem\_Selected\_PIN(object sender, RoutedEventArgs e)

{

selection = 1;

gateList.SelectedIndex = -1;

}

private void ListBoxItem\_Selected\_LABEL(object sender, RoutedEventArgs e)

{

selection = 2;

gateList.SelectedIndex = -1;

}

#endregion

Truth Tables

These are the methods for getting truth table outputs and drawing the tables themselves.

#region Truth Tables

private void drawTable(ref Canvas targetCanvas)

{

TextBlock heading;

List<char> inputs = new List<char>();

for (int i = 0; i < mainCircuit.getNum(2); i++) // Loop through the pins

{

if (!mainCircuit.getPin(i).getIfOutput()) // If the pin is an input pin

{

inputs.Add(mainCircuit.getPin(i).getChar()); // Add the pin's char to the list

}

}

if (mainCircuit.getNum(false) < 4 && mainCircuit.getNum(false) > 0 && mainCircuit.getNum(true) > 0 && listOutputPinsTruth.Text != "") // If a table can be created

{

Rectangle truthBorder = new Rectangle(); // Outer border for table

truthBorder.Width = 120 + (120 \* mainCircuit.getNum(false)); // The width is the number of input pins + 1 output pin

truthBorder.Height = 40 + 40 \* (Math.Pow(2, mainCircuit.getNum(false))); // The number of rows is 2^(Number of inputs) + the top row for labels

truthBorder.Fill = Brushes.White;

truthBorder.Stroke = Brushes.Black;

truthBorder.StrokeThickness = 2; // The outer edge should be thicker than the gridlines

targetCanvas.Children.Add(truthBorder);

Canvas.SetLeft(truthBorder, 0);

Canvas.SetTop(truthBorder, 0);

for (int i = 0; i < 1 + (Math.Pow(2, mainCircuit.getNum(false))); i++)//Horizontal Gridlines

{

Rectangle horiLine = new Rectangle();//Define parameters for rectangles making up gridlines

horiLine.Height = 40;

horiLine.Width = 120 + (120 \* mainCircuit.getNum(false));

horiLine.Stroke = Brushes.Black;//Colour of lines

if (i == 0)

{

horiLine.Fill = Brushes.LightGray; // Make the top row grey

horiLine.StrokeThickness = 2;

}

targetCanvas.Children.Add(horiLine);//Add gridlines to canvas

Canvas.SetTop(horiLine, i \* 40);//Place each rectangle 1 rectangle width apart

Canvas.SetLeft(horiLine, 0);

}

for (int i = 0; i < mainCircuit.getNum(false) && i < 2; i++)//Vertical Gridlines, Loop runs to less than 2 to prevent drawing too many rectangles as each rectangle count for 2 lines

{

Rectangle vertiLine = new Rectangle();//Define parameters for rectangles making up gridlines

vertiLine.Height = 40 + 40 \* (Math.Pow(2, mainCircuit.getNum(false)));

vertiLine.Width = 120;

vertiLine.Stroke = Brushes.Black;//Colour of lines

targetCanvas.Children.Add(vertiLine);//Add gridlines to canvas

Canvas.SetLeft(vertiLine, i \* 240);//Place each rectangle 1 rectangle width apart

Canvas.SetTop(vertiLine, 0);

}

for (int i = 0; i < mainCircuit.getNum(false) + 1; i++) // Loops through the number of input pins + 1 more for the output column

{

heading = new TextBlock(); // Create the heading textblock

heading.FontWeight = FontWeights.ExtraBold;

heading.FontSize = 25;

targetCanvas.Children.Add(heading); // Add the heading to the canvas

Canvas.SetLeft(heading, 50 + i \* 120); // Arrange the heading

Canvas.SetTop(heading, 2);

if (i == mainCircuit.getNum(false)) // If this is the output column

{

heading.Text = Convert.ToString(listOutputPinsTruth.SelectedItem); // Set the text to the output pin's char

}

else // If this is an input column

{

heading.Text = Convert.ToString(inputs[i]); // Set the text to the relevant input char

}

}

// Counting up in binary for the input columns

for (int i = 0; i < mainCircuit.getNum(false); i++) // i is for the column

{

for (int j = 1; j < Math.Pow(2, mainCircuit.getNum(false)) + 1; j++) // j is for the row

{

heading = new TextBlock(); // Reusing the heading variable

heading.FontWeight = FontWeights.ExtraBold;

heading.FontSize = 25;

targetCanvas.Children.Add(heading);

Canvas.SetLeft(heading, 50 + i \* 120);

Canvas.SetTop(heading, 2 + j \* 40);

heading.Text = Convert.ToString(Convert.ToString(j - 1, 2).PadLeft(mainCircuit.getNum(false), '0')[i]); // Get the relevant 1 or 0 for this part

}

}

// Calculating the output

for (int i = 1; i < Math.Pow(2, mainCircuit.getNum(false)) + 1; i++)

{

heading = new TextBlock(); // Reusing the heading variable

heading.FontWeight = FontWeights.ExtraBold;

heading.FontSize = 25;

targetCanvas.Children.Add(heading);

Canvas.SetLeft(heading, 50 + mainCircuit.getNum(false) \* 120);

Canvas.SetTop(heading, 2 + i \* 40);

if (evaluateForTable(Convert.ToString(i - 1, 2).PadLeft(mainCircuit.getNum(false), '0').ToCharArray())) // Use evaluateForTable to get the digit

{

heading.Text = "1";

}

else

{

heading.Text = "0";

}

}

}

else

{

if(mainCircuit.getNum(false) > 0)

{

MessageBox.Show("Cannot create truth table with more than 3 inputs", "Error", MessageBoxButton.OK);

}

else

{

MessageBox.Show("Cannot create truth table", "Error", MessageBoxButton.OK);

}

}

}

private void ButTruth\_Click(object sender, RoutedEventArgs e)

{

cnvTruth.Children.Clear(); // Clear the truth table canvas

drawTable(ref cnvTruth); // Draw the new truth table

parseCircuit(); // Parse the ciruit to reset the state of each wire

}

public bool evaluateForTable(char[] inputs)

{

bool[] boolInputs = new bool[inputs.Length]; // The input states required for this row

bool output = false;

for(int i = 0; i < boolInputs.Length; i++) // For each input

{

if(inputs[i] == '1') // Set the inputs to the relevant values

{

boolInputs[i] = true;

}

else

{

boolInputs[i] = false;

}

}

int offset = 0;

for(int i = 0; i < mainCircuit.getNum(2); i++) // Set the pin states to the values for this row

{

if (!mainCircuit.getPin(i).getIfOutput())

{

mainCircuit.getPin(i).setState(boolInputs[i-offset]);

}

else

{

offset++;

}

}

parseCircuit(); // Parse the circuit

for(int i = 0; i < mainCircuit.getNum(2); i++)

{

if(Convert.ToString(mainCircuit.getPin(i).getChar()) == listOutputPinsTruth.Text) // Find the correct output pin

{

output = mainCircuit.getPin(i).getState(); // Get the state of that pin as the output

}

}

for(int i = 0; i < mainCircuit.getNum(false); i++) // Reset the state of each pin to false

{

mainCircuit.getPin(i).setState(false);

}

parseCircuit(); // Parse the circuit

return output; // Return the output

}

#endregion

Boolean

These are the methods concerned with Boolean functionality. That includes generating trees recursively from an expression, analysing those trees to make a circuit and analysing a circuit to make an expression.

#region Boolean

private void SliNumInputs\_ValueChanged(object sender, RoutedPropertyChangedEventArgs<double> e)

{

labNumInputs.Content = Math.Floor(sliNumInputs.Value) + 1; // Set the label content to the integer value of the slider

if(numInputs < Math.Floor(sliNumInputs.Value) + 1) // If there are too few input buttons

{

addNewInputButton((int)Math.Floor(sliNumInputs.Value) + 1); // Add a new input buttong

}

else if(numInputs > Math.Floor(sliNumInputs.Value) + 1) // If there are too many input buttons

{

removeInputButton((int)Math.Floor(sliNumInputs.Value) + 2); // Remove an input button

}

numInputs = (int)Math.Floor(sliNumInputs.Value) + 1; // Set the new number of inputs

mainExpression.clear(); // Clear the main expression and the boolean canvas

cnvBoolInput.Children.Clear();

for (int i = 0; i < Convert.ToInt32(labNumInputs.Content); i++) // Re-enable all the input buttons

{

inputButtons[i].IsEnabled = true;

}

}

private void butExpression\_Click(object sender, RoutedEventArgs e)

{

try

{

labNumGatesCurrent.Content = "Number of Gates: " + mainCircuit.getNum(0); // Set the numbers of components

labNumWiresCurrent.Content = "Number of Wires: " + mainCircuit.getNum(1);

labNumPinsCurrent.Content = "Number of Pins: " + mainCircuit.getNum(2);

if (listOutputPinsBoolean.Text == "") // If no pin is selected

{

MessageBox.Show("Select an ouput pin.", "Error", MessageBoxButton.OK);

return;

}

for (int i = 0; i < mainCircuit.getNum(1); i++) // For the circuit to be represented as a binary tree, no loops can exist in the circuit wiring

{

for (int j = 0; j < mainCircuit.getNum(1); j++)

{

if (j != i && (mainCircuit.checkIfOnWire(j, mainCircuit.getWire(i).getLocation()) || mainCircuit.checkIfOnWire(j, mainCircuit.getWire(i).getEnd()))) // If any wire i connects to any wire j

{

MessageBox.Show("Cannot create binary tree, try removing loops in circuit", "Error", MessageBoxButton.OK); // If any wire to wire connection exists, the tree cannot be created

return;

}

}

}

int[] startPoint = new int[2];

for (int i = 0; i < mainCircuit.getNum(2); i++)

{

if (Convert.ToString(mainCircuit.getPin(i).getChar()) == listOutputPinsBoolean.Text) // Find the pin that corresponds to the selected character

{

mainCircuit.getPin(i).getLocation().CopyTo(startPoint, 0); // Copy the startpoint

startPoint[1] += 20; // Move the startpoint so it matches the y value of a connected wire

break;

}

}

TreeNode circuitTree = generateTreeFromCircuit(startPoint); // Generate a tree representing the circuit

List<string> infixForm = convertToChars(circuitTree.inOrderTraverse()); // Traverse the circuit to produce an infix expression, then convert it to a readable form

infixForm.Reverse();

string infixExpression = "";

foreach (string token in infixForm) // Turn the list into one string

{

infixExpression += token;

}

drawExpression(infixExpression, cnvOutput); // Draw the expression on the output canvas

infixExpression = "";

foreach (string token in simplifyExpression(infixForm)) // Turn the simplified expression list into one string

{

infixExpression += token;

}

drawExpression(infixExpression, cnvSimplified); // Draw the expression on the simplified canvas

}

catch

{

MessageBox.Show("Failed to generate expression", "Error", MessageBoxButton.OK);

}

}

private List<string> convertToChars(List<string> inputs)

{

List<string> output = new List<string>();

foreach(string thisIndex in inputs) // Swap each index for the symbol associated with it

{

if(thisIndex == ")") // If the index is a bracket, add a bracket

{

output.Add(")");

}

else if(thisIndex == "¬") // If the index indicates a NOT gate, add a NOT gate

{

output.Add("¬(");

}

else

{

for (int i = 0; i < mainCircuit.getNum(0); i++) // Check if the index represents a gate

{

if (mainCircuit.getGate(i).getIndex() == Convert.ToInt32(thisIndex)) // If the index matches a gate

{

switch (mainCircuit.getGate(i).getGateType()) // Add a string that represents the gate type

{

case 0:

output.Add(".");

break;

case 1:

output.Add("+");

break;

case 2:

output.Add("^");

break;

case 3:

output.Add("¬(");

break;

}

}

}

for (int i = 0; i < mainCircuit.getNum(2); i++) // Check if the index represents a pin

{

if (mainCircuit.getPin(i).getIndex() == -Convert.ToInt32(thisIndex)) // Gate indexes are negative

{

output.Add(Convert.ToString(mainCircuit.getPin(i).getChar())); // Add the char that represents that pin

}

}

}

}

return output;

}

private List<string> simplifyExpression(List<string> infixInput)

{

int numNOT;

string[] copiedInput = new string[infixInput.Count]; // Duplicate the input to an array

infixInput.CopyTo(copiedInput);

for(int i = 0; i < infixInput.Count() - 1; i++)

{

if(infixInput[i] == "¬(" && infixInput[i+1] == "¬(") // If two NOT gates follow each other

{

numNOT = 2;

for(int j = i + 2; j < infixInput.Count() - 1; j++)

{

if (infixInput[j] == ")" && infixInput[j + 1] == ")" && numNOT == 2) // If there is nothing between the NOT gates

{

infixInput.RemoveAt(i); // Remove the NOT gates

infixInput.RemoveAt(i);

infixInput.RemoveAt(j - 2);

infixInput.RemoveAt(j - 2);

break;

}

else if(infixInput[j] == ")") // If one of the NOTs finishes before the second one

{

numNOT--;

}

}

}

}

if (infixInput.SequenceEqual(copiedInput)) // If nothing was simplified

{

return new List<string>(); // Return a blank list

}

else // If something changed

{

return infixInput; // Return the new list

}

}

private TreeNode generateTreeFromCircuit(int[] startPoint)

{

for(int i = 0; i < mainCircuit.getNum(1); i++)

{

if (mainCircuit.getWire(i).getEnd().SequenceEqual(startPoint)) // Find the wire that connects to the startpoint

{

for (int j = 0; j < mainCircuit.getNum(0); j++) // Find the gate that connects to the other end of the wire

{

if (mainCircuit.getGate(j).getGateType() == 3) // If the gate is a NOT gate

{

if (mainCircuit.getGate(j).getLocation()[0] == mainCircuit.getWire(i).getLocation()[0] - 100 && mainCircuit.getGate(j).getLocation()[1] == mainCircuit.getWire(i).getLocation()[1] - 40)

{

return new TreeNode(Convert.ToString(mainCircuit.getGate(j).getIndex()), false, generateTreeFromCircuit(new int[2] { mainCircuit.getGate(j).getLocation()[0], mainCircuit.getGate(j).getLocation()[1] + 40 }), null); // Call the subroutine again to generate the left branch from this node

}

}

else if (mainCircuit.getGate(j).getNot()) // If the gate is a NOT version of another gate

{

if (mainCircuit.getGate(j).getLocation()[0] == mainCircuit.getWire(i).getLocation()[0] - 100 && mainCircuit.getGate(j).getLocation()[1] == mainCircuit.getWire(i).getLocation()[1] - 40)

{

return new TreeNode(Convert.ToString(mainCircuit.getGate(j).getIndex()), true, generateTreeFromCircuit(new int[2] { mainCircuit.getGate(j).getLocation()[0], mainCircuit.getGate(j).getLocation()[1] + 20 }), generateTreeFromCircuit(new int[2] { mainCircuit.getGate(j).getLocation()[0], mainCircuit.getGate(j).getLocation()[1] + 60 })); // Call the subroutine again to generate the right and left branches from this node

}

}

else if (mainCircuit.getGate(j).getLocation()[0] == mainCircuit.getWire(i).getLocation()[0] - 80 && mainCircuit.getGate(j).getLocation()[1] == mainCircuit.getWire(i).getLocation()[1] - 40)

{

return new TreeNode(Convert.ToString(mainCircuit.getGate(j).getIndex()), false, generateTreeFromCircuit(new int[2] { mainCircuit.getGate(j).getLocation()[0], mainCircuit.getGate(j).getLocation()[1] + 20 }), generateTreeFromCircuit(new int[2] { mainCircuit.getGate(j).getLocation()[0], mainCircuit.getGate(j).getLocation()[1] + 60 })); // Call the subroutine again to generate the right and left branches from this node

}

}

for (int j = 0; j < mainCircuit.getNum(2); j++) // If a gate doesn't connect to the wire, check for a pin instead

{

if (mainCircuit.getPin(j).getLocation()[0] == mainCircuit.getWire(i).getLocation()[0] - 20 && mainCircuit.getPin(j).getLocation()[1] == mainCircuit.getWire(i).getLocation()[1] - 20)

{

return new TreeNode(Convert.ToString(-mainCircuit.getPin(j).getIndex()));

}

}

}

else if(mainCircuit.getWire(i).getLocation().SequenceEqual(startPoint)) // Wires are bidirectional so the startpoint may match the start or the end of the wire

{

for (int j = 0; j < mainCircuit.getNum(0); j++) // Repeat the previous process, checking the wire ends instead of starts

{

if (mainCircuit.getGate(j).getGateType() == 3) // If the gate is a NOT gate

{

if (mainCircuit.getGate(j).getLocation()[0] == mainCircuit.getWire(i).getEnd()[0] - 100 && mainCircuit.getGate(j).getLocation()[1] == mainCircuit.getWire(i).getEnd()[1] - 40)

{

return new TreeNode(Convert.ToString(mainCircuit.getGate(j).getIndex()), false, generateTreeFromCircuit(new int[2] { mainCircuit.getGate(j).getLocation()[0], mainCircuit.getGate(j).getLocation()[1] + 40 }), null); // Call the subroutine again to generate the left branch from this node

}

}

else if (mainCircuit.getGate(j).getNot()) // If the gate is a NOT version of another gate

{

if (mainCircuit.getGate(j).getLocation()[0] == mainCircuit.getWire(i).getEnd()[0] - 100 && mainCircuit.getGate(j).getLocation()[1] == mainCircuit.getWire(i).getEnd()[1] - 40)

{

return new TreeNode(Convert.ToString(mainCircuit.getGate(j).getIndex()), true, generateTreeFromCircuit(new int[2] { mainCircuit.getGate(j).getLocation()[0], mainCircuit.getGate(j).getLocation()[1] + 20 }), generateTreeFromCircuit(new int[2] { mainCircuit.getGate(j).getLocation()[0], mainCircuit.getGate(j).getLocation()[1] + 60 })); // Call the subroutine again to generate the right and left branches from this node

}

}

else if (mainCircuit.getGate(j).getLocation()[0] == mainCircuit.getWire(i).getEnd()[0] - 80 && mainCircuit.getGate(j).getLocation()[1] == mainCircuit.getWire(i).getEnd()[1] - 40)

{

return new TreeNode(Convert.ToString(mainCircuit.getGate(j).getIndex()), false, generateTreeFromCircuit(new int[2] { mainCircuit.getGate(j).getLocation()[0], mainCircuit.getGate(j).getLocation()[1] + 20 }), generateTreeFromCircuit(new int[2] { mainCircuit.getGate(j).getLocation()[0], mainCircuit.getGate(j).getLocation()[1] + 60 })); // Call the subroutine again to generate the right and left branches from this node

}

}

for (int j = 0; j < mainCircuit.getNum(2); j++)

{

if (mainCircuit.getPin(j).getLocation()[0] == mainCircuit.getWire(i).getEnd()[0] - 20 && mainCircuit.getPin(j).getLocation()[1] == mainCircuit.getWire(i).getEnd()[1] - 20)

{

return new TreeNode(Convert.ToString(-mainCircuit.getPin(j).getIndex()));

}

}

break;

}

}

return null; // If nothing was found, return null

}

private TreeNode generateTreeFromString(List<string> expressionRPN)

{

try

{

if ("+.^".Contains(expressionRPN[0])) // If this token is a gate, add a gate and generate the branches

{

string token = expressionRPN[0];

expressionRPN.RemoveAt(0);

return new TreeNode(token, generateTreeFromString(expressionRPN), generateTreeFromString(expressionRPN));

}

else if (expressionRPN[0] == "¬") // If this token is a NOT gate, add a NOT gate and generate the left branch

{

string token = expressionRPN[0];

expressionRPN.RemoveAt(0);

return new TreeNode(token, null, generateTreeFromString(expressionRPN));

}

else // If this token is a pin, add it as an end node

{

string token = expressionRPN[0];

expressionRPN.RemoveAt(0);

return new TreeNode(token);

}

}

catch // If the expression was invalid

{

failedConstruction = true;

return null;

}

}

private void ButCircuit\_Click(object sender, RoutedEventArgs e)

{

labNumGatesNew.Content = "Number of Gates: " + mainExpression.getNumGates(); // Set the number of gates label

for(int i = 0; i < Convert.ToInt32(labNumInputs.Content); i++) // Loop through the input buttons

{

if(inputButtons[i].IsEnabled == true) // If the button is enabled and has not been pressed

{

MessageBox.Show("Cannot create circuit, unused inputs", "Error", MessageBoxButton.OK);

return; // Cancel the circuit creation

}

}

if (mainExpression.checkBracketValidity() && mainExpression.convertRPN().Count > 0 && mainExpression.checkStringValidity()) // If the expression is valid for brackets and not empty

{ resetCircuit(); // Reset the main canvas

List<string> expressionRPN = mainExpression.convertRPN();

int numNOTs = mainExpression.getNumNOTGates();

expressionRPN.Reverse(); // Reverse the RPN

TreeNode rootNode = generateTreeFromString(expressionRPN); // Generate a binary tree from the expression

if (failedConstruction) // If the tree failed to be constructed

{

MessageBox.Show("Failed to generate tree", "Error", MessageBoxButton.OK);

failedConstruction = false;

return; // Cancel the circuit creation

}

int[] location;

Path thisPin;

for(int i = 0; i < Convert.ToInt32(labNumInputs.Content); i++) // For the number of inputs

{

location = new int[2] { 40, i \* 80 + 20}; // Set the pin's location based on how many there are

thisPin = drawPin(location, true); // Draw the new pin

mainCircuit.addPin(new Pin(location));

pinPaths.Add(thisPin); // Add the pin to the list of paths

cnvMain.Children.Add(thisPin); // Add the pin to the grid

Canvas.SetLeft(thisPin, location[0]);

Canvas.SetTop(thisPin, location[1] + 10);

}

location = new int[2] { rootNode.getDepth() \* 120 + numNOTs \* 20, (Convert.ToInt32(labNumInputs.Content) - 1) \* 40 + 20}; // Set the output pin's location based on the depth of the tree and the number of inputs

thisPin = drawPin(location, false); // Draw the output pin

mainCircuit.addPin(new Pin(location));

mainCircuit.getPin(Convert.ToInt32(labNumInputs.Content)).flipOutput();

pinPaths.Add(thisPin); // Add the pin to the list of paths

cnvMain.Children.Add(thisPin); // Add the pin to the grid

Canvas.SetLeft(thisPin, location[0]);

Canvas.SetTop(thisPin, location[1] + 10);

listOutputPinsTruth.Items.Add(mainCircuit.getPin(Convert.ToInt32(labNumInputs.Content)).getChar()); // Add the output pin to the lists of output pins

listOutputPinsBoolean.Items.Add(mainCircuit.getPin(Convert.ToInt32(labNumInputs.Content)).getChar());

Wire newWire = new Wire(new int[2] { location[0], location[1] + 20 }); // Start the wire between the output pin and the final gate

if(mainExpression.getNumGates() == 0) // If there are no gates, connect the wire directly to a pin

{

newWire.addTurnPoint(new int[2] { location[0] - 60, location[1] + 20 });

}

else // Otherwise connect it to where the first gate output should be

{

newWire.addTurnPoint(new int[2] { location[0] - 40, location[1] + 20 });

}

PathFigureCollection thisStep = drawWireLine(newWire.getLocation(), newWire.getTurnPoints()[0]); // Draw a new line for this new segment

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

Path wirePath = new Path(); // Create the final path for the wire

wirePath.Data = entireWire; // Add the wire data to the path

wirePath.Stroke = Brushes.Blue;

cnvMain.Children.Add(wirePath); // Add the whole wire to the canvas

newWire.finishWire(); // Set the endpoint of the wire

mainCircuit.addWire(newWire); // Add the wire to the circuit

wirePaths.Add(wirePath); // Add the new wire path to the list

entireWire = new PathGeometry();

drawWires(rootNode, new int[2] { location[0] - 40, location[1] + 20}); // Draw the rest of the wires for each node in the tree

location = new int[2] { (rootNode.getDepth() - 1) \* 120 + numNOTs \* 20, (Convert.ToInt32(labNumInputs.Content) - 1) \* 40}; // Set location to top left corner of final gate

if(rootNode.getID() == "¬") // If the final gate is a NOT gate

{

drawGates(rootNode, new int[2] { location[0] - 20, location[1] }, true); // Draw the gates with wasNOT as true

}

else // If the final gate is any other kind of gate

{

drawGates(rootNode, location, false); // Draw the gates with wasNOT as false

}

labNumWiresNew.Content = "Number of Wires: " + mainCircuit.getNum(1); // Set the number of wires label

parseCircuit(); // Parse the circuit

}

else

{

MessageBox.Show("Invalid Expression", "Error", MessageBoxButton.OK);

}

}

private void drawWires(TreeNode thisNode, int[] location)

{

Wire newWire = new Wire(new int[2] { location[0] - 80, location[1] - 20 }); // Set the start of the wire at the top connector of the node

if(thisNode.getID() == "¬" && thisNode.getLeft() != null && !"ABCD".Contains(thisNode.getLeft().getID())) // If this node is a NOT gate and the next node is a gate

{

newWire = new Wire(new int[2] { location[0] - 100, location[1]}); // Set the start of the wire to the input of the NOT gate

int[] turnPoint = new int[2] { location[0] - 140, location[1] }; // Add a turnpoint at the start of the next gate

PathFigureCollection thisStep = drawWireLine(newWire.getLocation(), turnPoint); // Draw a new line for this new segment

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

Path wirePath = new Path(); // Create the final path for the wire

wirePath.Data = entireWire; // Add the wire data to the path

wirePath.Stroke = Brushes.Blue;

cnvMain.Children.Add(wirePath); // Add the whole wire to the canvas

newWire.finishWire(); // Set the endpoint of the wire

mainCircuit.addWire(newWire); // Add the wire to the circuit

wirePaths.Add(wirePath); // Add the new wire path to the list

entireWire = new PathGeometry();

drawWires(thisNode.getLeft(), new int[2] { location[0] - 140, location[1]}); // Repeat for the left branch

}

else if (thisNode.getID() == "¬" && thisNode.getLeft() != null && "ABCD".Contains(thisNode.getLeft().getID())) // If this node is a NOT gate and the next node is a pin

{

newWire = new Wire(new int[2] { location[0] - 100, location[1] }); // Set the start of the wire to the input of the NOT gate

int[] turnPoint;

PathFigureCollection thisStep;

Path wirePath;

switch (thisNode.getLeft().getID()) // Check which pin the wire needs to connect to

{

case "A":

turnPoint = new int[2] { mainCircuit.getPin(0).getLocation()[0] + (((newWire.getLocation()[0] - mainCircuit.getPin(0).getLocation()[0]) / 2) / 20) \* 20, mainCircuit.getPin(0).getLocation()[1] + 20 }; // Turnpoint at the midpoint of the start and end of the wire in the x direction

thisStep = drawWireLine(newWire.getLocation(), turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

turnPoint = new int[2] { 20 + mainCircuit.getPin(0).getLocation()[0], 20 + mainCircuit.getPin(0).getLocation()[1] }; // Turnpoint at the pin

thisStep = drawWireLine(newWire.getTurnPoints()[0], turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

break;

case "B":

turnPoint = new int[2] { mainCircuit.getPin(1).getLocation()[0] + (((newWire.getLocation()[0] - mainCircuit.getPin(0).getLocation()[0]) / 2) / 20) \* 20, mainCircuit.getPin(1).getLocation()[1] + 20 }; // Turnpoint at the midpoint of the start and end of the wire in the x direction

thisStep = drawWireLine(newWire.getLocation(), turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

turnPoint = new int[2] { 20 + mainCircuit.getPin(1).getLocation()[0], 20 + mainCircuit.getPin(1).getLocation()[1] }; // Turnpoint at the pin

thisStep = drawWireLine(newWire.getTurnPoints()[0], turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

break;

case "C":

turnPoint = new int[2] { mainCircuit.getPin(2).getLocation()[0] + (((newWire.getLocation()[0] - mainCircuit.getPin(0).getLocation()[0]) / 2) / 20) \* 20, mainCircuit.getPin(2).getLocation()[1] + 20 }; // Turnpoint at the midpoint of the start and end of the wire in the x direction

thisStep = drawWireLine(newWire.getLocation(), turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

turnPoint = new int[2] { 20 + mainCircuit.getPin(2).getLocation()[0], 20 + mainCircuit.getPin(2).getLocation()[1] }; // Turnpoint at the pin

thisStep = drawWireLine(newWire.getTurnPoints()[0], turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

break;

case "D":

turnPoint = new int[2] { mainCircuit.getPin(3).getLocation()[0] + (((newWire.getLocation()[0] - mainCircuit.getPin(0).getLocation()[0]) / 2) / 20) \* 20, mainCircuit.getPin(3).getLocation()[1] + 20 }; // Turnpoint at the midpoint of the start and end of the wire in the x direction

thisStep = drawWireLine(newWire.getLocation(), turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

turnPoint = new int[2] { 20 + mainCircuit.getPin(3).getLocation()[0], 20 + mainCircuit.getPin(3).getLocation()[1] }; // Turnpoint at the pin

thisStep = drawWireLine(newWire.getTurnPoints()[0], turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

break;

}

wirePath = new Path(); // Create the final path for the wire

wirePath.Data = entireWire; // Add the wire data to the path

wirePath.Stroke = Brushes.Blue;

cnvMain.Children.Add(wirePath); // Add the whole wire to the canvas

newWire.finishWire(); // Set the endpoint of the wire

mainCircuit.addWire(newWire); // Add the wire to the circuit

wirePaths.Add(wirePath); // Add the new wire path to the list

entireWire = new PathGeometry();

}

else if(thisNode.getLeft() != null && !"ABCD".Contains(thisNode.getLeft().getID())) // If the left node is a gate

{

int[] turnPoint = new int[2] { location[0] - 100, location[1] - 80 \* Convert.ToInt32(labNumInputs.Content) / 4 }; // Midpoint of the wire in the x direction

PathFigureCollection thisStep = drawWireLine(newWire.getLocation(), turnPoint); // Draw a new line for this new segment

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

turnPoint = new int[2] { location[0] - 120, location[1] - 80 \* Convert.ToInt32(labNumInputs.Content) / 4 }; // Enpoint of the wire in the x direction

thisStep = drawWireLine(newWire.getTurnPoints()[0], turnPoint); // Draw a new line for this new segment

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

Path wirePath = new Path(); // Create the final path for the wire

wirePath.Data = entireWire; // Add the wire data to the path

wirePath.Stroke = Brushes.Blue;

cnvMain.Children.Add(wirePath); // Add the whole wire to the canvas

newWire.finishWire(); // Set the endpoint of the wire

mainCircuit.addWire(newWire); // Add the wire to the circuit

wirePaths.Add(wirePath); // Add the new wire path to the list

entireWire = new PathGeometry();

drawWires(thisNode.getLeft(), new int[2] { location[0] - 120, location[1] - 80 \* Convert.ToInt32(labNumInputs.Content) / 4 }); // Repeat for the left branch

}

else if(thisNode.getLeft() != null && "ABCD".Contains(thisNode.getLeft().getID())) // If the left node is a pin

{

int[] turnPoint;

PathFigureCollection thisStep;

Path wirePath;

switch (thisNode.getLeft().getID()) // Check which pin the wire needs to connect to

{

case "A":

turnPoint = new int[2] { mainCircuit.getPin(0).getLocation()[0] + (((newWire.getLocation()[0] - mainCircuit.getPin(0).getLocation()[0]) / 2) / 20) \* 20, mainCircuit.getPin(0).getLocation()[1] + 20}; // Turnpoint at the midpoint of the start and end of the wire in the x direction

thisStep = drawWireLine(newWire.getLocation(), turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

turnPoint = new int[2] { 20 + mainCircuit.getPin(0).getLocation()[0], 20 + mainCircuit.getPin(0).getLocation()[1] }; // Turnpoint at the pin

thisStep = drawWireLine(newWire.getTurnPoints()[0], turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

break;

case "B":

turnPoint = new int[2] { mainCircuit.getPin(1).getLocation()[0] + (((newWire.getLocation()[0] - mainCircuit.getPin(0).getLocation()[0]) / 2) / 20) \* 20, mainCircuit.getPin(1).getLocation()[1] + 20 }; // Turnpoint at the midpoint of the start and end of the wire in the x direction

thisStep = drawWireLine(newWire.getLocation(), turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

turnPoint = new int[2] { 20 + mainCircuit.getPin(1).getLocation()[0], 20 + mainCircuit.getPin(1).getLocation()[1] }; // Turnpoint at the pin

thisStep = drawWireLine(newWire.getTurnPoints()[0], turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

break;

case "C":

turnPoint = new int[2] { mainCircuit.getPin(2).getLocation()[0] + (((newWire.getLocation()[0] - mainCircuit.getPin(0).getLocation()[0]) / 2) / 20) \* 20, mainCircuit.getPin(2).getLocation()[1] + 20 }; // Turnpoint at the midpoint of the start and end of the wire in the x direction

thisStep = drawWireLine(newWire.getLocation(), turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

turnPoint = new int[2] { 20 + mainCircuit.getPin(2).getLocation()[0], 20 + mainCircuit.getPin(2).getLocation()[1] }; // Turnpoint at the pin

thisStep = drawWireLine(newWire.getTurnPoints()[0], turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

break;

case "D":

turnPoint = new int[2] { mainCircuit.getPin(3).getLocation()[0] + (((newWire.getLocation()[0] - mainCircuit.getPin(0).getLocation()[0]) / 2) / 20) \* 20, mainCircuit.getPin(3).getLocation()[1] + 20 }; // Turnpoint at the midpoint of the start and end of the wire in the x direction

thisStep = drawWireLine(newWire.getLocation(), turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

turnPoint = new int[2] { 20 + mainCircuit.getPin(3).getLocation()[0], 20 + mainCircuit.getPin(3).getLocation()[1] }; // Turnpoint at the pin

thisStep = drawWireLine(newWire.getTurnPoints()[0], turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

break;

}

wirePath = new Path(); // Create the final path for the wire

wirePath.Data = entireWire; // Add the wire data to the path

wirePath.Stroke = Brushes.Blue;

cnvMain.Children.Add(wirePath); // Add the whole wire to the canvas

newWire.finishWire(); // Set the endpoint of the wire

mainCircuit.addWire(newWire); // Add the wire to the circuit

wirePaths.Add(wirePath); // Add the new wire path to the list

entireWire = new PathGeometry();

}

newWire = new Wire(new int[2] { location[0] - 80, location[1] + 20 });

if (thisNode.getRight() != null && !"ABCD".Contains(thisNode.getRight().getID())) // If the right node is a gate

{

int[] turnPoint = new int[2] { location[0] - 100, location[1] + 80 \* Convert.ToInt32(labNumInputs.Content) / 4 }; // Midpoint of the wire in the x direction

PathFigureCollection thisStep = drawWireLine(newWire.getLocation(), turnPoint); // Draw a new line for this new segment

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

turnPoint = new int[2] { location[0] - 120, location[1] + 80 \* Convert.ToInt32(labNumInputs.Content) / 4 }; // Enpoint of the wire in the x direction

thisStep = drawWireLine(newWire.getTurnPoints()[0], turnPoint); // Draw a new line for this new segment

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

Path wirePath = new Path(); // Create the final path for the wire

wirePath.Data = entireWire; // Add the wire data to the path

wirePath.Stroke = Brushes.Blue;

cnvMain.Children.Add(wirePath); // Add the whole wire to the canvas

newWire.finishWire(); // Set the endpoint of the wire

mainCircuit.addWire(newWire); // Add the wire to the circuit

wirePaths.Add(wirePath); // Add the new wire path to the list

entireWire = new PathGeometry();

drawWires(thisNode.getRight(), new int[2] { location[0] - 120, location[1] + 80 \* Convert.ToInt32(labNumInputs.Content) / 4 }); // Repeat for the right branch

}

else if (thisNode.getRight() != null && "ABCD".Contains(thisNode.getRight().getID())) // If the right node is a pin

{

int[] turnPoint;

PathFigureCollection thisStep;

Path wirePath;

switch (thisNode.getRight().getID()) // Check which pin the wire needs to connect to

{

case "A":

turnPoint = new int[2] { mainCircuit.getPin(0).getLocation()[0] + (((newWire.getLocation()[0] - mainCircuit.getPin(0).getLocation()[0]) / 2) / 20) \* 20, mainCircuit.getPin(0).getLocation()[1] + 20 }; // Turnpoint at the midpoint of the start and end of the wire in the x direction

thisStep = drawWireLine(newWire.getLocation(), turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

turnPoint = new int[2] { 20 + mainCircuit.getPin(0).getLocation()[0], 20 + mainCircuit.getPin(0).getLocation()[1] }; // Turnpoint at the pin

thisStep = drawWireLine(newWire.getTurnPoints()[0], turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

break;

case "B":

turnPoint = new int[2] { mainCircuit.getPin(1).getLocation()[0] + (((newWire.getLocation()[0] - mainCircuit.getPin(0).getLocation()[0]) / 2) / 20) \* 20, mainCircuit.getPin(1).getLocation()[1] + 20 }; // Turnpoint at the midpoint of the start and end of the wire in the x direction

thisStep = drawWireLine(newWire.getLocation(), turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

turnPoint = new int[2] { 20 + mainCircuit.getPin(1).getLocation()[0], 20 + mainCircuit.getPin(1).getLocation()[1] }; // Turnpoint at the pin

thisStep = drawWireLine(newWire.getTurnPoints()[0], turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

break;

case "C":

turnPoint = new int[2] { mainCircuit.getPin(2).getLocation()[0] + (((newWire.getLocation()[0] - mainCircuit.getPin(0).getLocation()[0]) / 2) / 20) \* 20, mainCircuit.getPin(2).getLocation()[1] + 20 }; // Turnpoint at the midpoint of the start and end of the wire in the x direction

thisStep = drawWireLine(newWire.getLocation(), turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

turnPoint = new int[2] { 20 + mainCircuit.getPin(2).getLocation()[0], 20 + mainCircuit.getPin(2).getLocation()[1] }; // Turnpoint at the pin

thisStep = drawWireLine(newWire.getTurnPoints()[0], turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

break;

case "D":

turnPoint = new int[2] { mainCircuit.getPin(3).getLocation()[0] + (((newWire.getLocation()[0] - mainCircuit.getPin(0).getLocation()[0]) / 2) / 20) \* 20, mainCircuit.getPin(3).getLocation()[1] + 20 }; // Turnpoint at the midpoint of the start and end of the wire in the x direction

thisStep = drawWireLine(newWire.getLocation(), turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

turnPoint = new int[2] { 20 + mainCircuit.getPin(3).getLocation()[0], 20 + mainCircuit.getPin(3).getLocation()[1] }; // Turnpoint at the pin

thisStep = drawWireLine(newWire.getTurnPoints()[0], turnPoint); // Draw a line between the end and the turnpoint of the wire

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(turnPoint);

break;

}

wirePath = new Path(); // Create the final path for the wire

wirePath.Data = entireWire; // Add the wire data to the path

wirePath.Stroke = Brushes.Blue;

cnvMain.Children.Add(wirePath); // Add the whole wire to the canvas

newWire.finishWire(); // Set the endpoint of the wire

mainCircuit.addWire(newWire); // Add the wire to the circuit

wirePaths.Add(wirePath); // Add the new wire path to the list

entireWire = new PathGeometry();

}

}

private void drawGates(TreeNode thisNode, int[] location, bool wasNOT)

{

int[] thisPlace;

switch (thisNode.getID()) // Check the type of gate in this node and create a new one at the location

{

case ".":

thisPlace = new int[2];

location.CopyTo(thisPlace, 0);

createNewGate(thisPlace, 0, false);

break;

case "+":

thisPlace = new int[2];

location.CopyTo(thisPlace, 0);

createNewGate(thisPlace, 1, false);

break;

case "^":

thisPlace = new int[2];

location.CopyTo(thisPlace, 0);

createNewGate(thisPlace, 2, false);

break;

case "¬":

thisPlace = new int[2];

location.CopyTo(thisPlace, 0);

createNewGate(thisPlace, 3, true);

break;

}

if (!wasNOT) // If the previous gate was not a NOT gate

{

location[1] -= 80 \* Convert.ToInt32(labNumInputs.Content) / 4; // Offset the location upwards in the y direction

}

if (thisNode.getLeft() != null && thisNode.getLeft().getID() == "¬") // If the left node is a NOT gate

{

location[0] -= 140; // Offset the location by 140 in the x direction

thisPlace = new int[2]; // Create the gate

location.CopyTo(thisPlace, 0);

drawGates(thisNode.getLeft(), thisPlace, true); // Repeat the process for the left node

location[0] += 140; // Undo the location offset so the right branch can be drawn

}

else if (thisNode.getLeft() != null && !"ABCD".Contains(thisNode.getLeft().getID())) // If the left node is not a NOT gate

{

location[0] -= 120; // Offset the location by 120 in the x direction

thisPlace = new int[2]; // Create the gate

location.CopyTo(thisPlace, 0);

drawGates(thisNode.getLeft(), thisPlace, false); // Repeat the process for the left node

location[0] += 120; // Undo the location offset so the right branch can be drawn

}

location[1] += 80 \* Convert.ToInt32(labNumInputs.Content) / 4; // Offset the direction downwards in the y direction

if(thisNode.getRight() != null && thisNode.getRight().getID() == "¬") // If the right node is a NOT gate

{

location[0] -= 140; // Offset the location by 140 in the x direction

location[1] += 80 \* Convert.ToInt32(labNumInputs.Content) / 4; // Offset the location downwards in the y direction

thisPlace = new int[2]; // Create the gate

location.CopyTo(thisPlace, 0);

drawGates(thisNode.getRight(), thisPlace, true); // Repeat the process for the right node

}

else if (thisNode.getRight() != null && !"ABCD".Contains(thisNode.getRight().getID())) // If the right node is not a NOT gate

{

location[0] -= 120; // Offset the location by 120 in the x direction

location[1] += 80 \* Convert.ToInt32(labNumInputs.Content) / 4; // Offset the location downwards in the y direction

thisPlace = new int[2];

location.CopyTo(thisPlace, 0);

drawGates(thisNode.getRight(), thisPlace, false); // Repeat the process for the right node

}

}

private void Input\_Button\_Click(object sender, RoutedEventArgs e)

{

for(int i = 0; i < 4; i++)

{

if (sender.Equals(inputButtons[i])) // Find the button that was pressed

{

mainExpression.addChar(i); // Add the char of the button

inputButtons[i].IsEnabled = false; // Disable the button

}

}

drawExpression(mainExpression.getExpression(), cnvBoolInput); // Redraw the expression

}

private void addNewInputButton(int buttonNum)

{

Button newButton = new Button(); // Create a new button

inputButtons[buttonNum-1] = newButton; // Add the button to the array

newButton.Width = 46;

newButton.Height = 37;

grdBoolean.Children.Add(newButton);

newButton.BorderBrush = Brushes.Black;

newButton.Margin = new Thickness(-1013 + 102\*buttonNum, -150, 0, 0); // Set the location in the row of the button

newButton.Click += Input\_Button\_Click;

newButton.Content = Convert.ToString((char)(64 + buttonNum));

}

private void removeInputButton(int buttonNum)

{

grdBoolean.Children.Remove(inputButtons[buttonNum - 1]); // Remove the specified button

inputButtons[buttonNum - 1] = new Button(); // Reset the item in the array

}

private void drawExpression(string literal, Canvas thisCanvas)

{

thisCanvas.Children.Clear(); // Ckear the canvas

List<string> outputText = new List<string>();

List<TextBlock> textInputs = new List<TextBlock>();

bool drawingNOT = false;

if (!GCSESelection) // For A-Level Notation

{

int otherBrackets = 0; // The number of intermediate brackets

outputText.Add(""); // Start the list

for (int i = 0; i < literal.Length; i++)

{

if(i != 0 && literal[i] == '(' && literal[i-1] != '¬')

{

otherBrackets++; // Increase the number of intermediate brackets

outputText[outputText.Count - 1] += literal[i] + "";

}

else if (drawingNOT && literal[i] == ')' && otherBrackets == 0) // If closing a bracket

{

outputText[outputText.Count - 1] += literal[i] + ""; // Add the bracket and a space

outputText.Add(""); // Add a new item to the list

drawingNOT = false; // Finish drawing a NOT hat if one was being drawn

}

else if(literal[i] == ')') // If the character is a close bracket

{

otherBrackets--; // Reduce the number of intermediate brackets

outputText[outputText.Count - 1] += literal[i] + "";

outputText.Add("");

}

else if (literal[i] == '¬') // If the character is a NOT operator

{

outputText.Add("¬"); // Add the NOT operator

drawingNOT = true; // Begin drawing a NOT hat

}

else

{

outputText[outputText.Count - 1] += literal[i] + ""; // If nothing NOT related is happening, add the character to the end of the list

}

}

int notCount = 1;

if(literal.Length > 0 && literal[0] == '¬') // If the expression starts with a NOT

{

notCount = 0;

}

Pen overLinePen = new Pen(Brushes.Black, 1); // Create a new pen for the overlines

for (int i = 0; i < outputText.Count; i++)

{

textInputs.Add(new TextBlock()); // Start the list of textblocks

textInputs[textInputs.Count - 1].FontSize = 20; // Set the font size

if (outputText[i] != "" && outputText[i][0] == '¬') // If this item is under a NOT hat

{

notCount++; // Increase the count of NOT gates

}

else if(outputText[i] != "") // If the item is not under a NOT hat

{

notCount--; // Reduce the number of NOT gates

}

if(notCount == 4) // Prevent more than 3 levels of NOT

{

MessageBox.Show("Cannot display more than 3 levels of NOT", "Error", MessageBoxButton.OK);

mainExpression.clear();

return;

}

for (int j = 0; j < notCount; j++) // For the number of NOT hats over a section of the expression

{

textInputs[i].TextDecorations.Add(new TextDecoration(TextDecorationLocation.OverLine, overLinePen, j \* -0.1, TextDecorationUnit.FontRecommended, TextDecorationUnit.FontRecommended)); // Add an overline

}

if (mainExpression.checkBracketValidity()) // If the circuit is valid

{

if (outputText[i] != "" && outputText[i][0] == '¬')

{

outputText[i] = outputText[i].Remove(0, 1); // Remove the NOT operator

}

}

textInputs[textInputs.Count - 1].Text = outputText[i]; // Set the text

}

}

else // For GCSE Notation

{

outputText.Add("");

for (int i = 0; i < literal.Length; i++)

{

if (literal[i] == '¬') // These if statements swap the A Level notation used in the literal expression for the GCSE notation

{

outputText[0] += '¬';

outputText[0] += '(';

i++;

}

else if (i != literal.Length - 1 && literal[i + 1] == ')')

{

outputText[0] += literal[i];

}

else if (literal[i] == '(')

{

outputText[0] += literal[i];

}

else if(literal[i] == '+')

{

outputText[0] += 'ᐱ';

outputText[0] += " ";

}

else if(literal[i] == '.')

{

outputText[0] += 'ᐯ';

outputText[0] += " ";

}

else

{

outputText[0] += literal[i];

outputText[0] += " ";

}

}

textInputs.Add(new TextBlock()); // Add a new textblock

textInputs[0].Text = outputText[0]; // Set the text to this character

textInputs[0].FontSize = 20;

}

thisCanvas.Children.Add(textInputs[0]); // Add the text to the canvas

Canvas.SetLeft(textInputs[0], 5);

Canvas.SetTop(textInputs[0], 4);

for (int i = 1; i < textInputs.Count; i++)

{

thisCanvas.Measure(new Size(Width, Height)); // Measure the canvas to get the ActualWidth

thisCanvas.Arrange(new Rect(0, 0, thisCanvas.DesiredSize.Width, thisCanvas.DesiredSize.Height));

thisCanvas.Children.Add(textInputs[i]);

Canvas.SetLeft(textInputs[i], Canvas.GetLeft(textInputs[i - 1]) + textInputs[i - 1].ActualWidth); // Arrange the text on the canvas

Canvas.SetTop(textInputs[i], 4);

}

}

private void ButtonALEVEL\_Click(object sender, RoutedEventArgs e)

{

butGCSE.Background = Brushes.DarkGray; // Set the selection of the buttons

butALEVEL.Background = Brushes.LightGray;

GCSESelection = false;

cnvBoolInput.Children.Clear(); // Clear all the canvases and the main expression

cnvOutput.Children.Clear();

cnvSimplified.Children.Clear();

mainExpression.clear();

for(int i = 0; i < Convert.ToInt32(labNumInputs.Content); i++) // Re-enable all the input buttons

{

inputButtons[i].IsEnabled = true;

}

}

private void ButGCSE\_Click(object sender, RoutedEventArgs e)

{

butGCSE.Background = Brushes.LightGray; // Set the selection of the buttons

butALEVEL.Background = Brushes.DarkGray;

GCSESelection = true;

cnvBoolInput.Children.Clear(); // Clear all the canvases and the main expression

cnvOutput.Children.Clear();

cnvSimplified.Children.Clear();

mainExpression.clear();

for (int i = 0; i < Convert.ToInt32(labNumInputs.Content); i++) // Re-enable all the input buttons

{

inputButtons[i].IsEnabled = true;

}

}

private void ButtonClear\_Click(object sender, RoutedEventArgs e)

{

mainExpression.clear(); // Clear the main expression

drawExpression(mainExpression.getExpression(), cnvBoolInput); // Redraw the canvas with the empty expression

for (int i = 0; i < Convert.ToInt32(labNumInputs.Content); i++) // Re-enable all the input buttons

{

inputButtons[i].IsEnabled = true;

}

}

// These events handle adding the relevant tokens to the expression based on different buttons being pressed

private void butNOT\_Click(object sender, RoutedEventArgs e)

{

mainExpression.startNOT();

drawExpression(mainExpression.getExpression(), cnvBoolInput);

}

private void ButAND\_Click(object sender, RoutedEventArgs e)

{

mainExpression.addAND();

drawExpression(mainExpression.getExpression(), cnvBoolInput);

}

private void ButOR\_Click(object sender, RoutedEventArgs e)

{

mainExpression.addOR();

drawExpression(mainExpression.getExpression(), cnvBoolInput);

}

private void ButXOR\_Click(object sender, RoutedEventArgs e)

{

mainExpression.addXOR();

drawExpression(mainExpression.getExpression(), cnvBoolInput);

}

private void ButOBracket\_Click(object sender, RoutedEventArgs e)

{

mainExpression.openBracket();

drawExpression(mainExpression.getExpression(), cnvBoolInput);

}

private void ButCBracket\_Click(object sender, RoutedEventArgs e)

{

mainExpression.closeBracket();

drawExpression(mainExpression.getExpression(),cnvBoolInput);

}

#endregion

Canvas Movement

These methods deal with moving the objects on the canvas when the user presses the arrow buttons of the main interface.

#region Canvas Movement

// These events handle the movement buttons being pressed and moving the canvas around on the canvas

private void butDown\_Click(object sender, RoutedEventArgs e)

{

moveComponents(-20, 0);

}

private void butUp\_Click(object sender, RoutedEventArgs e)

{

moveComponents(20, 0);

}

private void butRight\_Click(object sender, RoutedEventArgs e)

{

moveComponents(0, -20);

}

private void butLeft\_Click(object sender, RoutedEventArgs e)

{

moveComponents(0, 20);

}

private void moveComponents(int verticalShift, int horizontalShift)

{

deselectComponent();

if(totalOffset[0] - horizontalShift < 0 || totalOffset[1] - verticalShift < 0) // If the offset is 0 in either direction and the user tries to keep going

{

MessageBox.Show("Edge of canvas reached.", "Cannot move", MessageBoxButton.OK);

return; // Cancel the movement

}

totalOffset[0] -= horizontalShift; // Update the total offset

totalOffset[1] -= verticalShift;

for (int i = 0; i < labels.Count; i++) // For moving labels

{

Canvas.SetLeft(labels[i], Canvas.GetLeft(labels[i]) + horizontalShift); // Offset the labels

Canvas.SetTop(labels[i], Canvas.GetTop(labels[i]) + verticalShift);

}

for (int i = 0; i < mainCircuit.getNum(0); i++) // For moving gates

{

int[] newLocation = new int[2] { mainCircuit.getGate(0).getLocation()[0] + horizontalShift, mainCircuit.getGate(0).getLocation()[1] + verticalShift}; // Get the new location with the offset

Path objectToDraw = drawGate(newLocation, mainCircuit.getGate(0).getGateType(), mainCircuit.getGate(0).getNot()); // Get the gate to draw

cnvMain.Children.Add(objectToDraw); // Draw the gate

cnvMain.Children.Remove(gatePaths[0]); // Remove the previous path from the canvas

gatePaths.Add(objectToDraw); // The gate's index is determined by the order it was drawn in so it will have the same index in this list

gatePaths.Remove(gatePaths[0]); // Remove the previous path from the list

mainCircuit.addGate(new Gate(newLocation, mainCircuit.getGate(0).getGateType(), mainCircuit.getGate(0).getNot())); // Add the new gate to the circuit

mainCircuit.removeComponent(0, 0); // Remove the previous gate

}

for (int i = 0; i < mainCircuit.getNum(2); i++) // For moving pins

{

int[] newLocation = new int[2] { mainCircuit.getPin(0).getLocation()[0] + horizontalShift, mainCircuit.getPin(0).getLocation()[1] + verticalShift}; // Set the new location

Path thisPin;

mainCircuit.addPin(new Pin(newLocation, mainCircuit.getPin(0).getChar())); // Add the new pin at the new location to the circuit

mainCircuit.getPin(mainCircuit.getNum(2) - 1).setState(mainCircuit.getPin(0).getState()); // Set the new pin's state to the previous pin's state

if (mainCircuit.getPin(0).getIfOutput()) // If the previous pin was an output pin, set the new pin to be an output pin

{

thisPin = drawPin(newLocation, false); // Create the path for the new pin at the new location

mainCircuit.getPin(mainCircuit.getNum(2) - 1).flipOutput(); // Set the new pin to be an output pin

}

else

{

thisPin = drawPin(newLocation, true); // Create the path for the new pin at the new location

}

pinPaths.Add(thisPin); // Add the pin to the list of paths

cnvMain.Children.Add(thisPin); // Add the pin to the grid

Canvas.SetLeft(thisPin, newLocation[0]);

Canvas.SetTop(thisPin, newLocation[1] + 10);

cnvMain.Children.Remove(pinPaths[0]); // Remove the old pin

pinPaths.Remove(pinPaths[0]);

mainCircuit.removeComponent(2, 0);

}

for (int i = 0; i < mainCircuit.getNum(1); i++) // For moving wires

{

int[] newCoords = mainCircuit.getWire(0).getLocation();

newCoords[0] += horizontalShift; // Add the offset to the location

newCoords[1] += verticalShift;

Wire newWire = new Wire(newCoords); // Create a new wire

entireWire = new PathGeometry();

if(mainCircuit.getWire(0).getTurnPoints().Count == 0 && mainCircuit.getWire(0).getEnd()[0] == 0 && mainCircuit.getWire(0).getEnd()[1] == 0) // If a wire is just a point (and probably a mistake by the user) only remove it as it cannot have any effect on the circuit

{

cnvMain.Children.Remove(wirePaths[0]);

wirePaths.Remove(wirePaths[0]);

mainCircuit.removeComponent(1, 0);

continue; // Skip the rest of this iteration

}

else if (mainCircuit.getWire(0).getTurnPoints().Count == 0) // If a wire has no turnpoints

{

int[] endCoords = mainCircuit.getWire(0).getEnd();

endCoords[0] += horizontalShift; // Shift the end of the wire as required

endCoords[1] += verticalShift;

PathFigureCollection thisStep = drawWireLine(newCoords, endCoords); // Draw the new line between the new start and new end

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(endCoords); // Add the endpoint as a turnpoint so the wire can be finished later

}

else // If a wire has turnpoints

{

int[] lastEnd = mainCircuit.getWire(0).getTurnPoints()[0];

lastEnd[0] += horizontalShift; // Add the shift to the end of the previous segment

lastEnd[1] += verticalShift;

PathFigureCollection thisStep = drawWireLine(newCoords, lastEnd); // Draw a new line for this new segment

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(lastEnd); // Add the turnpoint to the wire

for (int j = 1; j < mainCircuit.getWire(0).getTurnPoints().Count; j++) // Repeat the process for every section of the wire

{

newCoords = mainCircuit.getWire(0).getTurnPoints()[j];

newCoords[0] += horizontalShift; // Add the shift to the location

newCoords[1] += verticalShift;

thisStep = drawWireLine(lastEnd, newCoords); // Draw the new line for the segment

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(newCoords); // Add the turnpoint to the wire

lastEnd = newCoords;

}

newCoords = mainCircuit.getWire(0).getEnd(); // Repeat the process for the final segment of the wire

newCoords[0] += horizontalShift; // Add the shift to the location

newCoords[1] += verticalShift;

thisStep = drawWireLine(lastEnd, newCoords);

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

newWire.addTurnPoint(newCoords); // Add the turnpoint to the wire

}

Path wirePath = new Path(); // Create the final path for the wire

wirePath.Data = entireWire; // Add the wire data to the path

wirePath.Stroke = Brushes.Blue;

cnvMain.Children.Add(wirePath); // Add the whole wire to the canvas

cnvMain.Children.Remove(wirePaths[0]); // Remove the old wire path from the canvas

newWire.finishWire(); // Set the endpoint of the wire

mainCircuit.addWire(newWire); // Add the wire to the circuit

wirePaths.Add(wirePath); // Add the new wire path to the list

wirePaths.Remove(wirePaths[0]); // Remove the old wire path from the list

mainCircuit.removeComponent(1, 0); // Remove the old wire from the circuit

entireWire = new PathGeometry();

}

parseCircuit(); // Parse the circuit to set the correct colours for every component

}

#endregion

File Interaction

These methods handle saving, loading and exporting circuits as well as clearing the canvas when needed.

#region File Interaction

private void ButNew\_Click(object sender, RoutedEventArgs e) // reset the main circuit

{

if (MessageBox.Show("Unsaved work may be lost, are you sure?", "New File", MessageBoxButton.YesNo) == MessageBoxResult.Yes)

{

resetCircuit(); // Reset the circuit

mainExpression = new Expression(); // Reset the mainExpression

cnvOutput.Children.Clear(); // Clear the boolean canvases

cnvSimplified.Children.Clear();

cnvBoolInput.Children.Clear();

}

}

private void resetCircuit()

{

listOutputPinsTruth.Items.Clear(); // Clear the output pin lists

listOutputPinsBoolean.Items.Clear();

totalOffset = new int[2] { 0, 0 }; // Reset to total offset to 0 in both directions

deselectComponent(); // Deselect whatever component is selected

if (mainCircuit.getNum(0) > 0) // Reset the next indexes for the pins and gates

{

mainCircuit.getGate(0).resetNextIndex();

}

if (mainCircuit.getNum(2) > 0)

{

mainCircuit.getPin(0).resetNextIndex();

}

for (int i = 0; i < pinPaths.Count; i++) // If the children of the main canvas were cleared, the grid lines would be removed. Therefore, each component has to be manually removed.

{

cnvMain.Children.Remove(pinPaths[i]);

}

pinPaths.Clear();

for (int i = 0; i < gatePaths.Count; i++)

{

cnvMain.Children.Remove(gatePaths[i]);

}

gatePaths.Clear();

for (int i = 0; i < wirePaths.Count; i++)

{

cnvMain.Children.Remove(wirePaths[i]);

}

wirePaths.Clear();

for (int i = 0; i < labels.Count; i++)

{

cnvMain.Children.Remove(labels[i]);

}

labels.Clear();

cnvMain.Children.Remove(currentComponent[0]); // Remove any remaining ghost

mainCircuit = new Circuit(); // Reset the main circuit

selectedIndex = -1; // Reset the selected index

wireHead = new Path[2]; // Reset the wireHead

entireWire = new PathGeometry(); // Reset the entireWire geometry

compList.SelectedIndex = -1; // Reset the selection of each list

gateList.SelectedIndex = -1;

selection = -1; // Reset selection

}

private void butSave\_Click(object sender, RoutedEventArgs e)

{

saveFile(); // Save the file... duh

}

private bool saveFile()

{

SaveFileDialog sf = new SaveFileDialog(); // Create a new dialog to get the file location

sf.Title = "Save Circuit";

sf.Filter = "Circuit File | \*.cf";

string fileText = mainCircuit.getNum(0) + "," + mainCircuit.getNum(1) + "," + mainCircuit.getNum(2) + "," + labels.Count + "," + totalOffset[0] + "," + totalOffset[1] + "\n"; // Metadata for the division of the components listed in the file and the offset of the circuit view

for (int i = 0; i < mainCircuit.getNum(0); i++) // Writing gates

{

Gate currentGate = mainCircuit.getGate(i); // Get the gate to save

fileText += currentGate.getLocation()[0] + "," + currentGate.getLocation()[1] + "," + currentGate.getGateType() + ","; // Add the gate's data to the file text

if (currentGate.getNot()) // Add whether the gate is a NOT version to the file text

{

fileText += "1\n";

}

else

{

fileText += "0\n";

}

}

for (int i = 0; i < mainCircuit.getNum(1); i++) // Writing wires

{

Wire currentWire = mainCircuit.getWire(i); // Get the wire to save

fileText += currentWire.getLocation()[0] + ";" + currentWire.getLocation()[1]; // Add the start of the wire to the file text

for (int j = 0; j < currentWire.getTurnPoints().Count; j++) // For each turnpoint

{

fileText += "," + currentWire.getTurnPoints()[j][0] + ";" + currentWire.getTurnPoints()[j][1]; // Add the turnpoint to the file text

}

fileText += "," + currentWire.getEnd()[0] + ";" + currentWire.getEnd()[1] + "\n"; // Add the end of the wire to the file text

}

for (int i = 0; i < mainCircuit.getNum(2); i++) // Writing pins

{

Pin currentPin = mainCircuit.getPin(i); // Get the pin to save

fileText += currentPin.getLocation()[0] + "," + currentPin.getLocation()[1] + ","; // Add the pin location to the file text

if (currentPin.getIfOutput()) // Add whether the pin is an output pin to the file text

{

fileText += "1\n";

}

else

{

fileText += "0\n";

}

}

for (int i = 0; i < labels.Count; i++) // Writing labels

{

fileText += Canvas.GetLeft(labels[i]) + "," + Canvas.GetTop(labels[i]) + "," + labels[i].Text + "\n"; // Add the label's data to the file text

}

if (sf.ShowDialog() == true) // If the user wishes to save

{

File.WriteAllText(sf.FileName, fileText); // Add all the text to a file

return true;

}

else // If the user does not wish to save

{

return false;

}

}

private void butLoad\_Click(object sender, RoutedEventArgs e)

{

OpenFileDialog lf = new OpenFileDialog(); // Create a new dialog to get the file location

lf.Title = "Load Circuit";

lf.Filter = "Circuit File | \*.cf";

string[] fileText;

int[] metaData = new int[6];

if (lf.ShowDialog() == true) // If the user wishes to load

{

try

{

fileText = File.ReadAllLines(lf.FileName); // Get the file text

for (int i = 0; i < 6; i++) // Get the metadata from the first line

{

metaData[i] = Convert.ToInt32(fileText[0].Split(',')[i]); // Split the first line by commas

}

resetCircuit(); // Reset the circuit

totalOffset[0] = metaData[4]; // Get the total offset from the metadata

totalOffset[1] = metaData[5];

// Loading Gates

int[,] gateInfo = new int[metaData[0], 4];

for (int i = 0; i < metaData[0]; i++) // For the number of gates specified in the metadata

{

for (int j = 0; j < 4; j++)

{

gateInfo[i, j] = Convert.ToInt32(fileText[i + 1].Split(',')[j]); // Get the information about ther gate

}

int[] location = new int[2];

location[0] = gateInfo[i, 0]; // Get the location from the gate info

location[1] = gateInfo[i, 1];

bool isNot;

if (gateInfo[i, 3] == 1) // Get whether the gate is a NOT version from the gate infor

{

isNot = true;

}

else

{

isNot = false;

}

createNewGate(location, gateInfo[i, 2], isNot); // Create a new gate with the gate info

}

// Loading Wires

List<int[]> wireInfo = new List<int[]>();

for (int i = 0; i < metaData[1]; i++) // For the number of wires specified in the metadata

{

for (int j = 0; j < fileText[i + metaData[0] + 1].Split(',').Length; j++) // Getting the wire info

{

wireInfo.Add(new int[2]);

wireInfo.Last()[0] = Convert.ToInt32(fileText[i + metaData[0] + 1].Split(',')[j].Split(';')[0]); // Add the x coord

wireInfo.Last()[1] = Convert.ToInt32(fileText[i + metaData[0] + 1].Split(',')[j].Split(';')[1]); // Add the y coord

}

Wire currentWire = new Wire(wireInfo[0]); // Create a new wire

entireWire = new PathGeometry();

if (wireInfo.Count == 2) // If the wire has no turnpoints

{

PathFigureCollection thisStep = drawWireLine(wireInfo[0], wireInfo.Last()); // Draw the wire line

for (int j = 0; j < 2; j++)

{

entireWire.Figures.Add(thisStep[j]); // Add the segments

}

currentWire.addTurnPoint(wireInfo[1]); // Add the end as a turnpoint

}

else // If the wire has turnpoints

{

for (int j = 0; j < wireInfo.Count - 1; j++) // For the number of turnpoints

{

PathFigureCollection thisStep = drawWireLine(wireInfo[j], wireInfo[j + 1]); // Draw a new segment

for (int k = 0; k < 2; k++)

{

entireWire.Figures.Add(thisStep[k]); // Add the segments

}

currentWire.addTurnPoint(wireInfo[j + 1]); // Add a new turnpoint

}

}

wireInfo.Clear(); // Clear the wire info

Path wirePath = new Path(); // Finish the wire

wirePath.Data = entireWire;

wirePath.Stroke = Brushes.Blue;

cnvMain.Children.Add(wirePath); // Add the whole wire to the canvas

currentWire.finishWire(); // Set the endpoint of the wire

mainCircuit.addWire(currentWire); // Add the wire to the circuit

wirePaths.Add(wirePath); // Add the wire path to the list

entireWire = new PathGeometry();

}

// Loading Pins

int[,] pinInfo = new int[metaData[2], 3];

for (int i = 0; i < metaData[2]; i++) // For the number of pins specified in the metadata

{

for (int j = 0; j < 3; j++)

{

pinInfo[i, j] = Convert.ToInt32(fileText[i + metaData[0] + metaData[1] + 1].Split(',')[j]); // Get the pin info

}

int[] location = new int[2];

location[0] = pinInfo[i, 0]; // Get the location

location[1] = pinInfo[i, 1];

bool isOutput;

Pin currentPin = new Pin(location); // Create a new pin

if (pinInfo[i, 2] == 1) // If the pin is an output pin

{

isOutput = true;

currentPin.flipOutput(); // Flip the pin output

listOutputPinsTruth.Items.Add(currentPin.getChar()); // Add the pin to the lists of output pins

listOutputPinsBoolean.Items.Add(currentPin.getChar());

}

else

{

isOutput = false;

}

Path thisPin = drawPin(location, !isOutput); // Draw the new pin

mainCircuit.addPin(currentPin); // Add the pin to the circuit

pinPaths.Add(thisPin); // Add the pin to the list of paths

cnvMain.Children.Add(thisPin); // Add the pin to the grid

Canvas.SetLeft(thisPin, location[0]);

Canvas.SetTop(thisPin, location[1] + 10);

}

// Loading Labels

for (int i = 0; i < metaData[3]; i++) // For the number of labels specified in the metadata

{

string[] labelInfo = fileText[i + metaData[0] + metaData[1] + metaData[2] + 1].Split(','); // Get the label info

TextBox thisLabel = new TextBox(); // Create a new label

thisLabel.FontWeight = FontWeights.ExtraBold;

thisLabel.FontSize = 20;

thisLabel.Background = Brushes.Transparent;

thisLabel.BorderBrush = Brushes.Transparent;

thisLabel.Text = labelInfo[2]; // Add the label text

labels.Add(thisLabel); // Add the label to the list

cnvMain.Children.Add(thisLabel); // Add the label to the canvas

Canvas.SetLeft(thisLabel, Convert.ToInt32(labelInfo[0])); // Set the label location

Canvas.SetTop(thisLabel, Convert.ToInt32(labelInfo[1]));

}

parseCircuit(); // Parse the new circuit

}

catch // If the file failed to load

{

MessageBox.Show("Failed to load file", "Error", MessageBoxButton.OK);

}

}

else

{

return;

}

}

private void butExport\_Click(object sender, RoutedEventArgs e)

{

try

{

moveComponents(totalOffset[1], totalOffset[0]); // Reset the positions of components relative to the origin

totalOffset = new int[2] { 0, 0 }; // Reset the total offset as components are now in original positions

PngBitmapEncoder encoder = new PngBitmapEncoder(); // Create an encoder to generate the png file

SaveFileDialog sf = new SaveFileDialog(); // Create a dialog to get where to save the image

sf.Title = "Export Circuit";

sf.Filter = "Portable Network Graphic | \*.png";

if (sf.ShowDialog() == true)

{

Canvas imageToSave = new Canvas(); // Create a new canvas to add paths to

int[] canvasWidths = new int[gatePaths.Count + pinPaths.Count + wirePaths.Count + labels.Count]; // These arrays will store x and y coordinates of every component on the grid

int[] canvasHeights = new int[gatePaths.Count + pinPaths.Count + wirePaths.Count + labels.Count];

for (int i = 0; i < gatePaths.Count; i++)

{

cnvMain.Children.Remove(gatePaths[i]); // Transfer each gate path from the main canvas to the new one so it can be exported

imageToSave.Children.Add(gatePaths[i]);

canvasWidths[i] = mainCircuit.getGate(i).getLocation()[0]; // Add the x and y coordinates of the gate to the respective arrays

canvasHeights[i] = mainCircuit.getGate(i).getLocation()[1];

}

for (int i = 0; i < pinPaths.Count; i++)

{

cnvMain.Children.Remove(pinPaths[i]); // Transfer each pin path from the main canvas to the new one so it can be exported

imageToSave.Children.Add(pinPaths[i]);

canvasWidths[i + gatePaths.Count] = mainCircuit.getPin(i).getLocation()[0]; // Add the x and y coordinates of the pin to the respective arrays

canvasHeights[i + gatePaths.Count] = mainCircuit.getPin(i).getLocation()[1];

}

for (int i = 0; i < wirePaths.Count; i++)

{

cnvMain.Children.Remove(wirePaths[i]); // Transfer each wire path from the main canvas to the new one so it can be exported

imageToSave.Children.Add(wirePaths[i]);

canvasWidths[i + gatePaths.Count + pinPaths.Count] = Math.Max(mainCircuit.getWire(i).getLocation()[0], mainCircuit.getWire(i).getEnd()[0]); // Add the x and y coordinates of the furthest vertex from the origin to the respective arrays

canvasHeights[i + gatePaths.Count + pinPaths.Count] = Math.Max(mainCircuit.getWire(i).getLocation()[1], mainCircuit.getWire(i).getEnd()[1]);

for (int j = 0; j < mainCircuit.getWire(i).getTurnPoints().Count; j++) // Check if any turnpoints on the wire are further from the origin and if so, add them to the array instead

{

if (mainCircuit.getWire(i).getTurnPoints()[j][0] > canvasWidths[i])

{

canvasWidths[i + gatePaths.Count + pinPaths.Count] = mainCircuit.getWire(i).getTurnPoints()[j][0];

}

if (mainCircuit.getWire(i).getTurnPoints()[j][1] > canvasHeights[i])

{

canvasHeights[i + gatePaths.Count + pinPaths.Count] = mainCircuit.getWire(i).getTurnPoints()[j][1];

}

}

}

for (int i = 0; i < labels.Count; i++)

{

int[] labelPosition = new int[2] { (int)Canvas.GetLeft(labels[i]), (int)Canvas.GetTop(labels[i]) }; // Get the position of each label on the canvas

cnvMain.Children.Remove(labels[i]); // Transfer each label from the main canvas to the new one so it can be exported

imageToSave.Children.Add(labels[i]);

Canvas.SetLeft(labels[i], labelPosition[0]); // Set the position of the label on the new canvas

Canvas.SetTop(labels[i], labelPosition[1]);

canvasWidths[i + gatePaths.Count + pinPaths.Count + wirePaths.Count] = labelPosition[0]; // Add the x and y coordinates of the label to the respective arrays

canvasHeights[i + gatePaths.Count + pinPaths.Count + wirePaths.Count] = labelPosition[1];

}

Array.Sort(canvasWidths); // Sort and reverse the coordinates in each array

Array.Sort(canvasHeights);

canvasWidths.Reverse();

canvasHeights.Reverse();

imageToSave.Width = canvasWidths.Last() + 120; // The exported image will be as wide and tall as the furthest point from the origin of any component

imageToSave.Height = canvasHeights.Last() + 120;

RenderTargetBitmap bitmap = new RenderTargetBitmap((int)imageToSave.Width, (int)imageToSave.Height, 96, 96, PixelFormats.Pbgra32); // Creat a bitmap to render the image onto

bitmap.Render(imageToSave);

BitmapFrame frame = BitmapFrame.Create(bitmap);

encoder.Frames.Add(frame);

using (FileStream stream = File.Create(sf.FileName))

{

encoder.Save(stream); // Use the encoder to save the file

}

for (int i = 0; i < gatePaths.Count; i++) // Add the component paths back to the main canvas

{

imageToSave.Children.Remove(gatePaths[i]);

cnvMain.Children.Add(gatePaths[i]);

}

for (int i = 0; i < pinPaths.Count; i++)

{

imageToSave.Children.Remove(pinPaths[i]);

cnvMain.Children.Add(pinPaths[i]);

}

for (int i = 0; i < wirePaths.Count; i++)

{

imageToSave.Children.Remove(wirePaths[i]);

cnvMain.Children.Add(wirePaths[i]);

}

for (int i = 0; i < labels.Count; i++) // Labels require a bit more work as they are more integrated into the canvas than the other components

{

int[] labelPosition = new int[2] { (int)Canvas.GetLeft(labels[i]), (int)Canvas.GetTop(labels[i]) };

imageToSave.Children.Remove(labels[i]);

cnvMain.Children.Add(labels[i]);

Canvas.SetLeft(labels[i], labelPosition[0]);

Canvas.SetTop(labels[i], labelPosition[1]);

}

}

}

catch

{

MessageBox.Show("Failed to export", "Error", MessageBoxButton.OK);

}

}

private void Window\_Closing(object sender, System.ComponentModel.CancelEventArgs e)

{

MessageBoxResult instruction = MessageBox.Show("Save circuit?", "Close Window", MessageBoxButton.YesNoCancel); // Ask the user if they want to save

if (instruction == MessageBoxResult.Yes) // If they want to save

{

if (!saveFile()) // If they do not save

{

e.Cancel = true; // Cancel the window closing

}

}

else if (instruction == MessageBoxResult.Cancel) // If they want to cancel the window closing

{

e.Cancel = true; // Cancel the window closing

}

}

private void butTableExport\_Click(object sender, RoutedEventArgs e)

{

PngBitmapEncoder encoder = new PngBitmapEncoder(); // Create an encoder to generate the png file

SaveFileDialog sf = new SaveFileDialog(); // Create a dialog to get where to save the image

sf.Title = "Export Table";

sf.Filter = "Portable Network Graphic | \*.png";

if(sf.ShowDialog() == true)

{

Canvas imageToSave = new Canvas(); // Create a canvas to draw to

drawTable(ref imageToSave); // Draw the table onto the canvas

imageToSave.Measure(new Size(Width, Height)); // Initialise the canvas to get an accurate width and height

imageToSave.Arrange(new Rect(0, 0, imageToSave.DesiredSize.Width, imageToSave.DesiredSize.Height));

imageToSave.Width = 120 + (120 \* mainCircuit.getNum(false)); // Set the required height and width

imageToSave.Height = 40 + 40 \* (Math.Pow(2, mainCircuit.getNum(false)));

RenderTargetBitmap bitmap = new RenderTargetBitmap((int)imageToSave.Width, (int)imageToSave.Height, 96, 96, PixelFormats.Pbgra32); // Creat a bitmap to render the image onto

bitmap.Render(imageToSave); // Render the image on the bitmap

BitmapFrame frame = BitmapFrame.Create(bitmap);

encoder.Frames.Add(frame);

using (FileStream stream = File.Create(sf.FileName))

{

encoder.Save(stream); // Use the encoder to save the file

}

}

}

#endregion

## Program Logic Code

Circuit Class

class Circuit

{

private List<Gate> gates;

private List<Wire> wires;

private List<Pin> pins;

public Circuit()

{

gates = new List<Gate>();

wires = new List<Wire>();

pins = new List<Pin>();

}

public void evaluateCircuit()

{

for (int i = 0; i < wires.Count; i++)

{

for (int j = 0; j < pins.Count; j++) // Checking if the start or end points of a wire i are at the same location as an output of a pin j

{

if ((pins[j].getLocation()[0] + 20 == wires[i].getLocation()[0] && pins[j].getLocation()[1] + 20 == wires[i].getLocation()[1]) || (pins[j].getLocation()[0] + 20 == wires[i].getEnd()[0] && pins[j].getLocation()[1] + 20 == wires[i].getEnd()[1]))

{

wires[i].setState(pins[j].getState()); // Evaluate wires connected to input pins

}

}

for (int j = 0; j < gates.Count; j++) // These if statements check if the start or end points of wire i are at the same location as an input or output of a gate j

{

if(gates[j].getNot() == true)

{

if ((gates[j].getLocation()[0] == wires[i].getLocation()[0] - 100 && gates[j].getLocation()[1] == wires[i].getLocation()[1] - 40) || (gates[j].getLocation()[0] == wires[i].getEnd()[0] - 100 && gates[j].getLocation()[1] == wires[i].getEnd()[1] - 40))

{

wires[i].setState(gates[j].getState()); // Evaluate wires connected to gate NOT outputs

}

}

else if (gates[j].getLocation()[0] == wires[i].getLocation()[0] - 80 && gates[j].getLocation()[1] == wires[i].getLocation()[1] - 40 || (gates[j].getLocation()[0] == wires[i].getEnd()[0] - 80 && gates[j].getLocation()[1] == wires[i].getEnd()[1] - 40))

{

wires[i].setState(gates[j].getState()); // Evaluate wires connected to gate outputs

}

if (gates[j].getGateType() == 3)

{

if ((gates[j].getLocation()[0] == wires[i].getEnd()[0] && gates[j].getLocation()[1] + 40 == wires[i].getEnd()[1]) || (gates[j].getLocation()[0] == wires[i].getLocation()[0] && gates[j].getLocation()[1] + 40 == wires[i].getLocation()[1]))

{

gates[j].setState(0, wires[i].getState()); // Set NOT gate state to connected wire state

gates[j].Output(); // Evaluate gate output

}

}

else if ((gates[j].getLocation()[0] == wires[i].getLocation()[0] && gates[j].getLocation()[1] + 20 == wires[i].getLocation()[1]) || (gates[j].getLocation()[0] == wires[i].getEnd()[0] && gates[j].getLocation()[1] + 20 == wires[i].getEnd()[1]))

{

gates[j].setState(0, wires[i].getState()); // Set first gate state to connected wire state

gates[j].Output(); // Evaluate gate output

}

else if ((gates[j].getLocation()[0] == wires[i].getEnd()[0] && gates[j].getLocation()[1] + 60 == wires[i].getEnd()[1]) || (gates[j].getLocation()[0] == wires[i].getLocation()[0] && gates[j].getLocation()[1] + 60 == wires[i].getLocation()[1]))

{

gates[j].setState(1, wires[i].getState()); // Set second gate state to connected wire state

gates[j].Output(); // Evaluate gate output

}

}

for (int j = 0; j < wires.Count; j++) // Wire to wire connections to the wire j

{

if (checkIfOnWire(j, wires[i].getLocation()))

{

wires[i].setState(wires[j].getState());

}

else if (checkIfOnWire(j, wires[i].getEnd()))

{

wires[i].setState(wires[j].getState());

}

}

for (int j = 0; j < pins.Count; j++) // Checking if the start or end points of a wire i are at the same location as an input of a pin j

{

if ((pins[j].getLocation()[0] == wires[i].getEnd()[0] && pins[j].getLocation()[1] == wires[i].getEnd()[1] - 20) || (pins[j].getLocation()[0] == wires[i].getLocation()[0] && pins[j].getLocation()[1] == wires[i].getLocation()[1] - 20))

{

if (pins[j].getIfOutput() && pins[j].getState() != wires[i].getState()) // If pin is an output pin and its state is not the same as the wire connecting to it

{

pins[j].flipState(); // Flip the state of the pin

}

}

}

}

}

public bool checkIfOnWire(int j, int[] checkLocation) // J is index of wire to be checked

{

if (wires[j].getTurnPoints().Count > 0) // If the wire has turnpoints

{

// FOR FIRST SEGMENT

if (checkLocation[1] == wires[j].getLocation()[1]) // If the checkLocation is in the same y plane as the start of wire j

{

if (wires[j].getTurnPoints()[0][0] > wires[j].getLocation()[0]) // If wire j heads to the right initially

{

if (checkLocation[0] >= wires[j].getLocation()[0]) // If the checkLocation is to the right of the start of wire j

{

if (checkLocation[0] <= wires[j].getTurnPoints()[0][0]) // If the checkLocation is to the left of the first turnpoint

{

return true;

}

}

}

else if (wires[j].getTurnPoints()[0][0] < wires[j].getLocation()[0]) // If wire j heads to the left initially

{

if (checkLocation[0] <= wires[j].getLocation()[0]) // If the checkLocation is to the left of the start of wire j

{

if (checkLocation[0] >= wires[j].getTurnPoints()[0][0]) // If the checkLocation is to the right of the first turnpoint

{

return true;

}

}

}

}

else if (checkLocation[0] == wires[j].getTurnPoints()[0][0]) // If the checkLocation is in the same x plane as the first turnpoint

{

if (wires[j].getTurnPoints()[0][1] < wires[j].getLocation()[1]) // If wire j heads upwards initially

{

if (checkLocation[1] <= wires[j].getLocation()[1]) // If the checkLocation is above the start of wire j

{

if (checkLocation[1] >= wires[j].getTurnPoints()[0][1]) // If the checkLocation is below the first turnpoint

{

return true;

}

}

}

else if (wires[j].getTurnPoints()[0][1] > wires[j].getLocation()[1]) // If wire j heads downwards initially

{

if (checkLocation[1] >= wires[j].getLocation()[1]) // If the checkLocation is below the start of wire j

{

if (checkLocation[1] <= wires[j].getTurnPoints()[0][1]) // If the checkLocation is above the first turnpoint

{

return true;

}

}

}

}

// FOR MIDDLE SEGMENTS

for (int k = 0; k < wires[j].getTurnPoints().Count - 1; k++)

{

if (checkLocation[1] == wires[j].getTurnPoints()[k][1]) // If the checkLocation is in the same y plane as the kth turnpoint

{

if (wires[j].getTurnPoints()[k + 1][0] > wires[j].getTurnPoints()[k][0]) // If wire j heads to the right after the kth turnpoint

{

if (checkLocation[0] >= wires[j].getTurnPoints()[k][0]) // If the checkLocation is to the right of the kth turnpoint

{

if (checkLocation[0] <= wires[j].getTurnPoints()[k + 1][0]) // If the checkLocation is to the left of the k+1th turnpoint

{

return true;

}

}

}

else if (wires[j].getTurnPoints()[k + 1][0] < wires[j].getTurnPoints()[k][0]) // If wire j heads to the left after the kth turnpoint

{

if (checkLocation[0] <= wires[j].getTurnPoints()[k][0]) // If the checkLocation is to the left of the kth turnpoint

{

if (checkLocation[0] >= wires[j].getTurnPoints()[k + 1][0]) // If the checkLocation is to the right of the k+1th turnpoint

{

return true;

}

}

}

}

else if (checkLocation[0] == wires[j].getTurnPoints()[k + 1][0]) // If the checkLocation is in the same x plane as the kth turnpoint

{

if (wires[j].getTurnPoints()[k + 1][1] < wires[j].getTurnPoints()[k][1]) // If wire j heads upwards after the kth turnpoint

{

if (checkLocation[1] <= wires[j].getTurnPoints()[k][1]) // If the checkLocation is above the start of the kth turnpoint

{

if (checkLocation[1] >= wires[j].getTurnPoints()[k + 1][1]) // If the checkLocation is below the k+1th turnpoint

{

return true;

}

}

}

else if (wires[j].getTurnPoints()[k + 1][1] > wires[j].getTurnPoints()[k][1]) // If wire j heads downwards after the kth turnpoint

{

if (checkLocation[1] >= wires[j].getTurnPoints()[k][1]) // If the checkLocation is below the kth turnpoint

{

if (checkLocation[1] <= wires[j].getTurnPoints()[k + 1][1]) // If the checkLocation is above the k+1th turnpoint

{

return true;

}

}

}

}

}

// FOR FINAL SEGMENT

if (checkLocation[1] == wires[j].getTurnPoints()[wires[j].getTurnPoints().Count - 1][1]) // If the checkLocation is in the same y plane as the kth turnpoint

{

if (wires[j].getEnd()[0] > wires[j].getTurnPoints()[wires[j].getTurnPoints().Count - 1][0]) // If wire j heads to the right after the kth turnpoint

{

if (checkLocation[0] >= wires[j].getTurnPoints()[wires[j].getTurnPoints().Count - 1][0]) // If the checkLocation is to the right of the kth turnpoint

{

if (checkLocation[0] <= wires[j].getEnd()[0]) // If the checkLocation is to the left of the k+1th turnpoint

{

return true;

}

}

}

else if (wires[j].getEnd()[0] < wires[j].getTurnPoints()[wires[j].getTurnPoints().Count - 1][0]) // If wire j heads to the left after the kth turnpoint

{

if (checkLocation[0] <= wires[j].getTurnPoints()[wires[j].getTurnPoints().Count - 1][0]) // If the checkLocation is to the left of the kth turnpoint

{

if (checkLocation[0] >= wires[j].getEnd()[0]) // If the checkLocation is to the right of the k+1th turnpoint

{

return true;

}

}

}

}

else if (checkLocation[0] == wires[j].getEnd()[0]) // If the checkLocation is in the same x plane as the kth turnpoint

{

if (wires[j].getEnd()[1] < wires[j].getTurnPoints()[wires[j].getTurnPoints().Count - 1][1]) // If wire j heads upwards after the kth turnpoint

{

if (checkLocation[1] <= wires[j].getTurnPoints()[wires[j].getTurnPoints().Count - 1][1]) // If the checkLocation is above the start of the kth turnpoint

{

if (checkLocation[1] >= wires[j].getEnd()[1]) // If the checkLocation is below the k+1th turnpoint

{

return true;

}

}

}

else if (wires[j].getEnd()[1] > wires[j].getTurnPoints()[wires[j].getTurnPoints().Count - 1][1]) // If wire j heads downwards after the kth turnpoint

{

if (checkLocation[1] >= wires[j].getTurnPoints()[wires[j].getTurnPoints().Count - 1][1]) // If the checkLocation is below the kth turnpoint

{

if (checkLocation[1] <= wires[j].getEnd()[1]) // If the checkLocation is above the k+1th turnpoint

{

return true;

}

}

}

}

}

else // If there are no turnpoints in the wire, it is a single segment

{

if (checkLocation[1] == wires[j].getLocation()[1]) // If the checkLocation is in the same y plane as the start of wire j

{

if (wires[j].getEnd()[0] > wires[j].getLocation()[0]) // If wire j heads to the right initially

{

if (checkLocation[0] >= wires[j].getLocation()[0]) // If the checkLocation is to the right of the start of wire j

{

if (checkLocation[0] <= wires[j].getEnd()[0]) // If the checkLocation is to the left of the first turnpoint

{

return true;

}

}

}

else if (wires[j].getEnd()[0] < wires[j].getLocation()[0]) // If wire j heads to the left initially

{

if (checkLocation[0] <= wires[j].getLocation()[0]) // If the checkLocation is to the left of the start of wire j

{

if (checkLocation[0] >= wires[j].getEnd()[0]) // If the checkLocation is to the right of the first turnpoint

{

return true;

}

}

}

}

else if (checkLocation[0] == wires[j].getEnd()[0]) // If the checkLocation is in the same x plane as the first turnpoint

{

if (wires[j].getEnd()[1] < wires[j].getLocation()[1]) // If wire j heads upwards initially

{

if (checkLocation[1] <= wires[j].getLocation()[1]) // If the checkLocation is above the start of wire j

{

if (checkLocation[1] >= wires[j].getEnd()[1]) // If the checkLocation is below the first turnpoint

{

return true;

}

}

}

else if (wires[j].getEnd()[1] > wires[j].getLocation()[1]) // If wire j heads downwards initially

{

if (checkLocation[1] >= wires[j].getLocation()[1]) // If the checkLocation is below the start of wire j

{

if (checkLocation[1] <= wires[j].getEnd()[1]) // If checkLocation is above the first turnpoint

{

return true;

}

}

}

}

}

return false;

}

public void reduceGateIndexes(int startPoint)

{

for(int i = startPoint; i < gates.Count; i++)

{

gates[i].reduceIndex(); // Reduce the indexes of all the gates above a certain threshold index

}

}

public void reducePinIndexes(int startPoint)

{

for(int i = startPoint; i < pins.Count; i++)

{

pins[i].reduceIndex(); // Reduce the indexes of all the pins above a certain threshold index

}

}

public Gate getGate(int index)

{

return gates[index];

}

public Wire getWire(int index)

{

return wires[index];

}

public Pin getPin(int index)

{

return pins[index];

}

public int getNum(int type) // 0 Gates, 1 Wires, 2 Pins

{

switch (type)

{

case 0:

return gates.Count;

case 1:

return wires.Count;

case 2:

return pins.Count;

}

return -1;

}

public int getNum(bool isInput) // Returns the number of input or output pins

{

int counter = 0;

if (isInput)

{

for(int i = 0; i < pins.Count; i++) // Loop through pins

{

if (pins[i].getIfOutput()) // If the pin is an output pin

{

counter++; // Increase the counter

}

}

}

else

{

for (int i = 0; i < pins.Count; i++) // Loop through pins

{

if (!pins[i].getIfOutput()) // If the pin is not an output pin

{

counter++; // Increase the counter

}

}

}

return counter;

}

public void removeComponent(int type, int index) // 0 Gates, 1 Wires, 2 Pins

{

switch (type)

{

case 0:

gates.RemoveAt(index);

break;

case 1:

wires.RemoveAt(index);

break;

case 2:

pins.RemoveAt(index);

break;

}

}

public void addGate(Gate newGate)

{

gates.Add(newGate);

}

public void addWire(Wire newWire)

{

wires.Add(newWire);

}

public void addPin(Pin newPin)

{

pins.Add(newPin);

}

}

}

Expression Class

class Expression

{

private string literalString;

public Expression()

{

literalString = "";

}

public void addChar(int character) // Adding a variable char to the string

{

switch (character)

{

case 0:

literalString += "A";

break;

case 1:

literalString += "B";

break;

case 2:

literalString += "C";

break;

case 3:

literalString += "D";

break;

}

}

public int getNumGates()

{

int numGates = 0;

for(int i = 0; i < literalString.Length; i++)

{

if(literalString[i] == '.' || literalString[i] == '+' || literalString[i] == '^' || literalString[i] == '¬') // If the character is a gate

{

numGates++; // Increase the counter

}

}

return numGates;

}

public int getNumNOTGates()

{

int numGates = 0;

for (int i = 0; i < literalString.Length; i++)

{

if (literalString[i] == '¬') // If the character is a NOT gate

{

numGates++; // Increase the counter

}

}

return numGates;

}

public List<string> convertRPN()

{

Queue<string> input = new Queue<string>();

Stack<string> operatorStack = new Stack<string>();

for (int i = 0; i < literalString.Length; i++) // Add the characters in the literal string as tokens in the input queue

{

input.Enqueue(Convert.ToString(literalString[i]));

}

string variables = "ABCD";

string ops = "+.^¬";

List<string> output = new List<string>();

while (input.Count != 0) // Shunting Yard Algorithm

{

if (variables.Contains(input.Peek())) // If a variable is at the front of the input queue

{

output.Add(input.Dequeue()); // Add the variable to the output

}

else if (ops.Contains(input.Peek())) // If an operator is at the front of the input queue

{

if (input.Peek() == "+" || input.Peek() == "^") // If an OR or XOR gate is at the front of the input queue

{

while (operatorStack.Count != 0 && operatorStack.Peek() == "." || operatorStack.Count != 0 && operatorStack.Peek() == "¬") // AND and NOT gates take higher precedence over OR and XOR

{

output.Add(operatorStack.Pop()); // Add the operators to the output

}

}

else if (input.Peek() == ".") // If an AND gate is at the front of the input queue

{

while (operatorStack.Count != 0 && operatorStack.Peek() == "¬") // NOT gates take higher priority over AND gates

{

output.Add(operatorStack.Pop()); // Add the operators to the output

}

}

operatorStack.Push(input.Dequeue()); // Add the front of the input queue to the operator stack

}

else if (input.Peek() == "(") // If an open bracket is at the front of the input queue

{

operatorStack.Push(input.Dequeue()); // Add the bracke to the operator stack

}

else if (input.Peek() == ")") // If a close bracket is at the front of the input queue

{

while (operatorStack.Count != 0 && operatorStack.Peek() != "(") // If there are operators on the stack and not an openbracket

{

output.Add(operatorStack.Pop()); // Add the operators to the output

}

if (operatorStack.Count > 0) // Remove any remaining operators or open brackets

{

operatorStack.Pop();

}

else if (operatorStack.Count == 0) // If there are no more operators or open brackets

{

break;

}

input.Dequeue(); // Remove the input

}

}

while (operatorStack.Count != 0) // While there are operators remaining on the stack

{

output.Add(operatorStack.Pop()); // Add the operators to the output

}

return output;

}

public bool checkStringValidity()

{

for(int i = 0; i < literalString.Length - 1; i++)

{

if ("ABCD".Contains(literalString[i]) && "ABCD".Contains(literalString[i + 1]))

{

return false;

}

else if ("ABCD".Contains(literalString[i]) && literalString[i + 1] == '¬')

{

return false;

}

else if ("ABCD".Contains(literalString[i]) && literalString[i + 1] == '(')

{

return false;

}

else if (literalString[i] == ')' && "ABCD".Contains(literalString[i + 1]))

{

return false;

}

}

return true;

}

public bool checkBracketValidity()

{

int openBrackets = 0; // The number of unclosed brackets

for(int i = 0; i < literalString.Length; i++)

{

if(literalString[i] == '(') // If a bracket is opened

{

openBrackets++; // Increase the number of unclosed brackets

}

else if(literalString[i] == ')') // If a bracket is closed

{

openBrackets--; // Decrease the number of unclosed brackets

}

if(openBrackets < 0)

{

return false;

}

}

if(openBrackets == 0) // If all brackets are closed

{

return true;

}

else // If some brackets are open of too many are closed

{

return false;

}

}

public void addAND()

{

literalString += ".";

}

public void addOR()

{

literalString += "+";

}

public void addXOR()

{

literalString += "^";

}

public void openBracket()

{

literalString += "(";

}

public void closeBracket()

{

literalString += ")";

}

public void startNOT()

{

literalString += "¬(";

}

public void clear()

{

literalString = "";

}

public string getExpression()

{

return literalString;

}

}

TreeNode Class

class TreeNode

{

private string ID; // The ID represents the content of each node

private TreeNode left, right; // The left and right nodes are the branches from this node

private bool isNOT;

public TreeNode(string content) // This constructor is used to add pins to a tree

{

ID = content;

left = right = null;

}

public TreeNode(string content, TreeNode newRight, TreeNode newLeft) // This constructor is used when generating a tree from a string, in which case the right branch should be executed first

{

ID = content;

left = newLeft;

right = newRight;

}

public TreeNode(string content, bool NOT, TreeNode newLeft, TreeNode newRight) // This constructor is used when generating a tree from a circuit, in which case the left branch should be executed first

{

ID = content;

left = newLeft;

right = newRight;

isNOT = NOT;

}

public int getDepth()

{

if(left == null && right == null) // If an end node, return depth 1

{

return 1;

}

else if(left == null) // If left is null but right isn't, something is wrong. Return depth 0

{

return 0;

}

else if(right == null || left.getDepth() > right.getDepth()) // If right is null or left is deeper than right, return 1 + the depth of the left node

{

return 1 + left.getDepth();

}

else // If right is deeper than left, return 1 + the depth of the right node

{

return 1 + right.getDepth();

}

}

public List<string> inOrderTraverse() // This is an in-order traversal of the tree with some extra steps to make NOT gates with only one input work nicely

{

List<string> output = new List<string>();

if (right == null && Convert.ToInt32(ID) > 0 || isNOT) // If this node represents a NOT gate

{

output.Add(")"); // Add a 0 to represent a closed bracket

}

if (left != null) // If a left node exists

{

foreach (string element in left.inOrderTraverse()) // Repeat the traversal for the left node

{

output.Add(element); // Add each element from the left branch

}

}

output.Add(Convert.ToString(ID)); // Add the ID of this node

if (right != null && !isNOT) // If a right node exists and this is not the NOT form of a gate

{

foreach (string element in right.inOrderTraverse()) // Repeat the traversal for the right node

{

output.Add(element); // Add each element from the right branch

}

}

else if(right != null && isNOT)

{

foreach (string element in right.inOrderTraverse()) // Repeat the traversal for the right node

{

output.Add(element); // Add each element from the right branch

}

output.Add("¬"); // Add a NOT symbol to indicate a NOT gate

}

return output; // Return the output list

}

public TreeNode getLeft()

{

return left;

}

public TreeNode getRight()

{

return right;

}

public string getID()

{

return ID;

}

}

Component Classes

class Component

{

protected bool state;

protected int[] location;

protected int index;

public Component(int[] position)

{

state = false;

location = position;

}

public virtual void reduceNextIndex() // Method to be overwritten by children

{}

public virtual void resetNextIndex() // Method to be overwritten by children

{ }

public virtual void setState(bool newState)

{

state = newState;

}

public void reduceIndex()

{

index--;

}

public void flipState()

{

state = !state;

}

public int getIndex()

{

return index;

}

public int[] getLocation()

{

return location;

}

public bool getState()

{

return state;

}

}

class Pin : Component

{

private bool outputPin; // Is the pin an output pin

private static char nextChar = 'A';

private char pinChar; // The character of the pin

private static int nextIndex = 1;

private static List<char> spareChars = new List<char>();

public Pin(int[] position) : base(position) // This constructor is used to create pin with the next available char

{

index = nextIndex;

nextIndex++;

outputPin = false;

if(spareChars.Count > 0) // If a spare char exists use that

{

pinChar = spareChars.Last();

spareChars.Remove(pinChar);

}

else // Otherwise use the next char

{

pinChar = nextChar;

nextChar++;

}

}

public Pin(int[] position, char specificChar) : base(position) // This constructor is used to create a pin with a specific char

{

index = nextIndex;

nextIndex++;

outputPin = false;

pinChar = specificChar;

}

public void addSpareChar()

{

spareChars.Add(pinChar); // Add the pin's char to the list of spare chars

spareChars.Sort(); // Keep the spare chars in alphabetical order

spareChars.Reverse();

}

public override void resetNextIndex()

{

nextChar = 'A'; // Reset the indexes to their default values

nextIndex = 1;

}

public override void reduceNextIndex()

{

nextIndex--;

}

public bool getIfOutput()

{

return outputPin;

}

public void flipOutput()

{

outputPin = !outputPin;

}

public char getChar()

{

return pinChar;

}

}

class Wire : Component

{

private int[] endPoint; // Coordinate of the wire's end

private List<int[]> turnPoints; // Coordinates of every point the wire turns at

public Wire(int[] start) : base(start)

{

endPoint = new int[2];

turnPoints = new List<int[]>();

}

public void finishWire()

{

if(turnPoints.Count > 0)

{

endPoint = turnPoints.Last(); // Set the final turnpoint as the endpoint

turnPoints.RemoveAt(turnPoints.Count - 1); // Remove the final turnpoint from the list

}

}

public void addTurnPoint(int[] newPoint)

{

turnPoints.Add(newPoint);

}

public List<int[]> getTurnPoints()

{

return turnPoints;

}

public int[] getEnd()

{

return endPoint;

}

}

class Gate : Component

{

private bool[] inputs;

private bool not;

private int type; // 0 - And, 1 - Or, 2 - Xor, 3 - Not

private static int nextIndex = 1;

public Gate(int[] place, int gateType, bool isNot) : base(place)

{

inputs = new bool[2] { false, false };

type = gateType;

not = isNot;

Output();

index = nextIndex;

nextIndex++;

}

public override void reduceNextIndex()

{

nextIndex--;

}

public override void resetNextIndex()

{

nextIndex = 1;

}

public void setState(int i, bool newState)

{

inputs[i] = newState;

}

public bool getInput(int num)

{

return inputs[num];

}

public bool getNot()

{

return not;

}

public int getGateType()

{

return type;

}

public void setLocation(int[] newPlace)

{

location = newPlace;

}

public void Output()

{

switch (type)

{

case 0: // AND gate

state = inputs[0] & inputs[1];

break;

case 1: // OR gate

state = inputs[0] | inputs[1];

break;

case 2: // XOR gate

state = inputs[0] ^ inputs[1];

break;

case 3: // NOT gate

state = !inputs[0];

break;

}

if (not && type != 3) // If a NOT version of a gate, flip the state

{

state = !state;

}

}

}

# Evaluation

## Objectives

1. **Users must be able to easily create logic circuit diagrams that satisfy the requirements of GCSE and A Level courses.**

The program fulfils this objective almost completely. Circuit diagrams have no spatial constraints and can theoretically represent any valid logic circuit and evaluate it accurately. All the main types of gates are implemented correctly and Pins are fully integrated into the circuits. Some users reported initial confusion as to how the wire creation worked but generally it seemed once users understood the system they had no issues with it. During testing, all users successfully completed the tasks they were given using the program and most reported no issues in doing so. Most users reported taking very little time to complete the tasks set for them when filling out the questionnaire. This shows that the circuit creation tools are easy to use.

The quality of life features like the component ghosts on the canvas were well-received. Some users reported the wire creation was somewhat repetitive and requested a way to place multiple wires without having to re-select the wire button. This would have been possible to implement but would have required a major rework of the wire input functionality and would have made only minor improvements to user experience. Overall, this objective is fulfilled excellently.

1. **Users must be able to receive useful analysis of any logic circuit they create in the program.**

This objective has two parts, both of which are met adequately. The program generates truth tables accurately for most valid circuits. An arbitrary limit was imposed preventing the user from generating truth tables for circuits with more than three inputs. This limit was definitely necessary as the scope of truth tables becomes unmanageable very quickly as the number of inputs increases. The specific number of three inputs may have been set somewhat low but this was done to prevent tables being created that breached the boundaries of the program window. With more code, a system could be envisioned that could manage larger truth tables and the program is capable of this expansion. However, the circuits that will be created by the target users will rarely need to be so large as to meet this boundary so the limit was deemed acceptable.

Regarding the Boolean functionality, the program is capable of generating expressions representing any valid logic circuit in either GCSE or A Level notation. One oversight made in the programming is the lack of brackets used in the expression creation. This can lead to expression being created that do fit the circuit but could be read inaccurately. This issue would only be encountered for very large circuits and the Boolean analysis generates correctly for most circuits. The initial questionnaire saw respondents request simplification of the Boolean expressions. This has been implemented but not to the maximum. The program will cancel unnecessary NOT hats and double NOT gates in A level and GCSE notation. This is useful for demonstrating how NOT gates can cancel each other out. However, there are other simplifications that have not been implemented, mainly due to time pressure. For instance, the program will not remove redundant gates as was mentioned in the analysis. These features could be implemented fairly easily with the current program structure though, as expressions are stored as strings that could be analysed and edited as needed.

The program did not end up with an implementation of SVG file formats for saving. This would have required extra planning and work as vector graphics are created very differently to bitmap graphics. This is not a mjor issue though as SVG formats were only mentioned in passing during the initial questionnaire and are not a requirement for viewing an exported circuit properly. Karnaugh maps were also mentioned in the initial analysis as a potential extra feature but these were not implemented. This is the largest failing of this objective as the program would require heavy modification to include these diagrams. However, while the analysis they provide may be very interesting, it is not necessarily relevant to GCSEs or A Levels and so was overlooked. Overall, the program does meet the expectations of this objective as users can get accurate analysis of their circuits. The features could have been expanded further though.

1. **The program must have multiple ways to create logic circuits.**

The main part of this objective is the Boolean input functions. The program is capable of producing accurate circuits from Boolean expressions with up to four inputs. The limitation of four inputs was added after significant development without it. This does reduce the potential of the feature of the program as the most advanced circuits cannot be generated. However, anything relevant to the school’s qualifications will be able to be created by the program. The circuits themselves are always generated so that they output the correct logical states however, the form they take on the grid can be unintuitive for a user to understand when the inputs are not entered in alphabetical order. This is an issue with the approach taken in the code which is disappointingly rigid. This is intentional though as the scope of a feature that creates the circuit totally dynamically is enormous and could form an entire project on its own.

The input system of the Boolean menu works excellently. Excluding the limitation on the number of inputs, the keyboard can be used to create any expression the user requires. All test users reported no significant issues in creating expressions. One slight issue is the lack of a backspace button which a few test users said would be helpful. However, the keyboard does have a clear button that removes the entire expression and the time taken to re-enter the expression tends to be very low so this is not a major issue.

Overall, the program does meet this objective, even if it could be implemented better.

1. **The program must enable users to view and save their circuits in a useful formats.**

The program’s save system is robust and dynamic. The file structure provides no limit on the number of components in a circuit being saved and the system has never failed to load a valid file. This allows users to keep any circuit they could create and reload it at a later date. The only supported format for exporting is PNG and this provides all the visual requirements for the circuit. The size of the exported canvas scales to match the size of the circuit. Overall, this objective has been met.

1. **The program must be simple to use.**

Objective 5 is met as all users indicated that they were able to understand how to use the program quickly with little instruction. Regarding simplicity specifically, most users said that the program interface was not cluttered which is exactly the desired response. A few users did think the interface was a bit cluttered but they all reported that this was not an issue. I believe this is directly because of the design of the interface. The three main functions of the program (circuits, Boolean and truth tables) are accessible via the three labelled buttons which are always visible to the user and the currently selected window is always highlighted. This means that users always know where they are in the program and can always see how to get to a different part if they need to. The three functions are also totally separated in the GUI, allowing users to distinguish between the functions of each. One area that could have been improved was the interaction between the different functions. One user in the testing phase reported that they became confused when trying to create a circuit from a Boolean expression as the program did not automatically switch to the circuit menu. Most users did not experience this issue however, and that user did work out what was happening and did not repeat the issue. All buttons in the program are labelled descriptively so that users can predict their function without needing to click them.

The biggest issue with simplicity is in the wire placement on the circuit menu. Multiple test users expressed initial confusion as to how the system worked and it did need to be explained a few times. However, once users understood how the system functions, no further issues were reported. This could be an issue with the program if it is used by a user with no guidance. Since the original intention was for the program to be used in a classroom setting, teachers should be on hand to provide guidance so this oversight shouldn’t cause a major issue.

Overall, the program’s interface can generally be described as simple. Once users understand the different systems of input they can become quite proficient at circuit and expression creation. I consider this objective to be met.

1. **The program must feel enjoyable and intuitive to use.**

In the response to question 10 of the testing survey, many users expressed enthusiasm at the idea of using the program to learn about logic circuits and Boolean algebra. This response was not expected but is most welcome. It demonstrates that users come away from using the program with a good view of it. Users would welcome using the program in lessons and similar environments to learn about logic circuits and Boolean algebra because they have enjoyed using it. As for intuitiveness, already mentioned is the fact that some users were initially confused by the wire placement system. This is a failure as far as intuitive design goes however it isn’t fatal to he program as users are able to work out the system and use it effectively once they have done so. Other parts of the program have been shown to be quite intuitive. Users were able to create Boolean algebra expressions with little explanation of the system beforehand. Also, no users reported difficulty in navigating the program which again shows that its design is intuitive and users can figure out how to get between different interfaces without requiring explanation. Overall, though there are some issues in the program’s intuitiveness, the program is generally easy to learn to use and users report good feelings towards using it so this objective is met.

1. **The program must be useful to both younger and older students as well as teachers.**

Originally I had hoped to test the program directly with younger students however this did not take place. That doesn’t mean that the usefulness of the program to younger students cannot be measured. All the features of logic circuits taught at GCSE and below are found in the program. Any younger user wishing to create a circuit diagram can do so without needing to traverse menus or manage settings. This direct design is intentional as younger users had become often confused by the current system employed by the school. Students at GCSE level also have all the capability they require to create logic circuits. Though Boolean algebra does make up a smaller part of their course than that at A Level, the program has all the required functions for creating Boolean expressions in GCSE notation. It was originally intended for GCSE notation to be the default of the program to prevent GCSE students from needing to swap the setting when they use the program. However, it was decided to instead make A Level notation the default as this notation is not only easier to understand (assuming one has been taught what the notation means) but A Level students will make far more use of the Boolean functionality than GCSE students ever will. This ties into the usefulness for older students as an important part of the design was ensuring that older students did not find the program has been ‘dumbed down’ for younger students to understand. Regarding teachers, they are the main target of the program’s export functions as these will allow them to create circuits and then generate an image of the circuit that can be given to students during a lesson for demonstration purposes. The program’s UI also places a large focus on the visual aspects (i.e. the grid in the main circuit GUI) so that teachers can present the circuits or tables or expressions to a class. Overall, no group’s needs have been prioritised over any others which is good. Though younger students have not been questioned on this, the program contains all the features they could require and these are readily accessible upon loading the program. Older students have the functions they require for creating large circuits as well as demonstrating Boolean expressions. This objective has largely been met.

1. **The program should complete its functions in a reasonable time.**

This was probably the least important objective but it has definitely been met in totality. All program features execute almost instantly as none of the algorithms in code are particularly intensive to run. Though the program has not been tested for enormous circuits, the point at which the processes might start to slow down is long past the point that the intended user base would need to work at. This means that no target user should ever experience a slow down of the program when executing a function without deliberately trying to cause the program to slow down by creating a circuit much larger than required.

## Extensions

Extension objectives one and two were not met as their suggestions for the program fell outside the bounds of the needs of the target user base. The functions would have been interesting for users to observe but were fundamentally unnecessary.

Extension objective three was implemented as the infinite canvas of the program. Users are able to scroll infinitely to the right or downwards on the canvas. This removes any limitations on the size of circuit that users can create. The actual utility of this feature is debateable as target users will rarely need to generate a circuit that is much larger than the canvas. However, there are possibilities where users may wish to create circuits that must expand beyond the canvas (e.g. joining together adders) and so this extension will be useful to them.

The last extension objective was not implemented but it was kept in mind during development. The program could be expanded to include subcircuits and the object structure reflects this. The circuit class is only ever instatiated once in the current program as the mainCircuit variable. However, if someone were to implement a sub-circuit system, they would be able to create more instances of the circuit class to represent each of these circuits. A system could then be envisioned of evaluating the output of each subcircuit like the program currently evaluates the output of each gate and treating sub-circuits like gates on the grid. The GUI also has room for a sub-circuit feature. Another button could be added to the circuit, Boolean, truth table bar that took the user to a sub-circuit interface similar to the main circuit GUI but with the ability to save sub-circuits for use in the main circuit.

## Overall Comments

I am happy with the outcome of the project. All the primary objectives have been met in some way, with most of them being fulfilled totally. Only one extension objective was met but the program has room for expansion to include others. There are changes I would make if I could go back an redo the project. For one thing, the system I elected to use of handling component connections by checking if they overlapped could have been done better. This system did work for the circuit grid but having some kind of central record of the connections between components would have made the other parts of the program significantly easier.