Simple Projects

Project Logic Gates

A GCE Computing Project

By Qi-rui Chen

GENETATE TABLE OF CONTENTS

Analysis

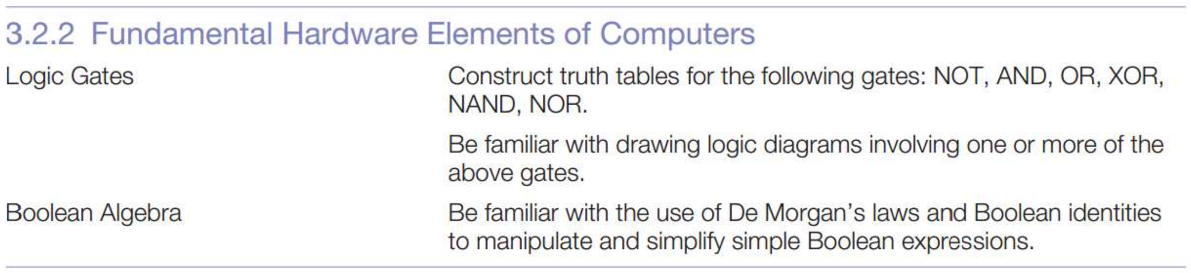
Target Client

Mr Starkings (LGS Computing Teacher)
Mr Statham (LGS Computing Teacher)

Background

The Client

Currently, AQA Computing is being taught at my school. Logic gates are a part of COMP2 and are examined in the written exam. It is taught to year 12 (16 to 18) doing Computing AS. Below is an extract from the AQA computing syllabus:



3.2.2 Fundamental Hardware Elements of Computers

Logic Gates	Construct truth tables for the following gates: NOT, AND, OR, XOR, NAND, NOR.
	Be familiar with drawing logic diagrams involving one or more of the above gates.
Boolean Algebra	Be familiar with the use of De Morgan's laws and Boolean identities to manipulate and simplify simple Boolean expressions.

Figure 1: an extract from the AQA Computing syllabus

Students are taught be either Mr Starkings or Mr Statham, members of the computing department. Prior to being taught about logic gates, students should have some basic programming experience (therefore understand logical AND, OR and NOT) and binary numbers.

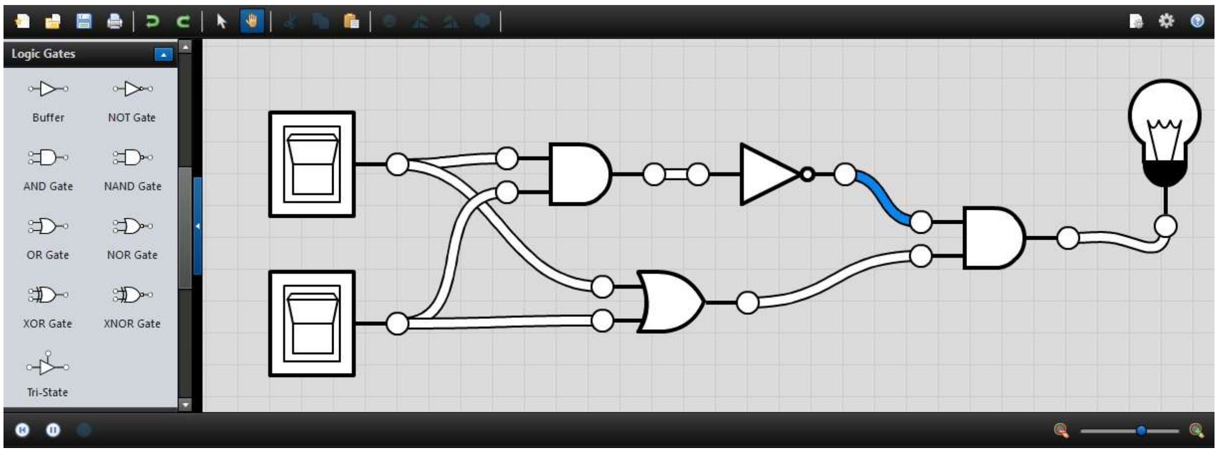


Figure 2: an XOR gate implemented using {AND, OR}

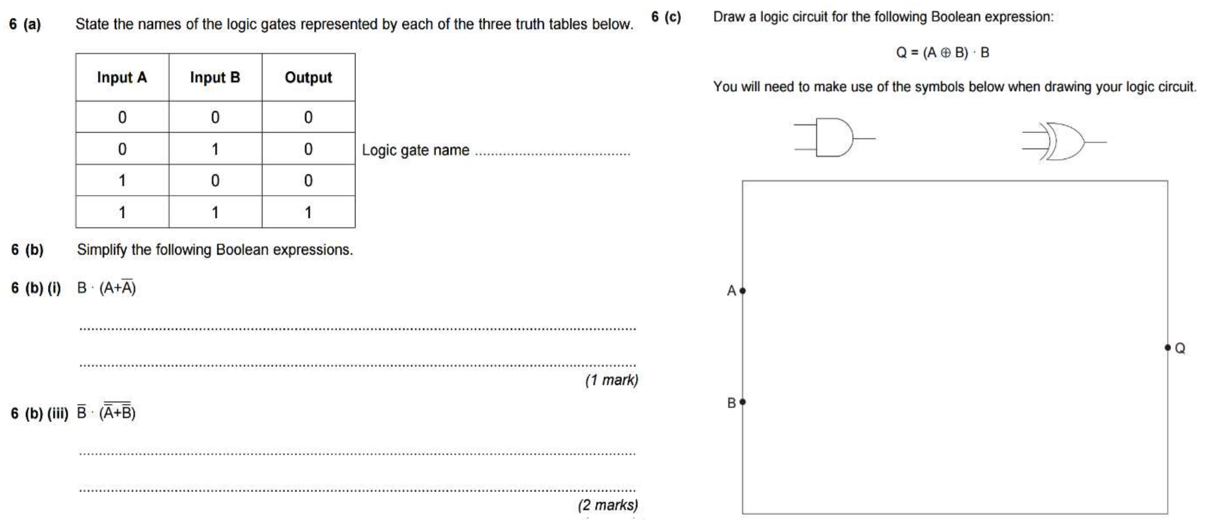
In lessons, students are given an opportunity to use logic.ly (http://logic.ly/demo/), a logic gate simulation program. This allows students to play around to learn a bit about logic gates for themselves. They are given time to experiment with putting different gates together to see what they can create. Figure 2 shows an example of what you can achieve in logic.ly.

Logic.ly is a paid program written by Joshua Tynjala using ActionScript. He provides a demo version at http://logic.ly/demo/ as well as a 30 day free trial for his windows application, which reverts to the demo version after 30 days. The demo version does not allow you to save the circuit or print it. The simulation aspect is not limited in any way.

Students can also refer to the AQA Computing AS Student Book which includes a reasonable sized section on Boolean algebra and logic gates. The book covers logic gates, truth tables and even functional completeness (although it does not use this term), asking the reader to construct logic circuits using only NAND and NOR gates.

The textbook covers De Morgan's Law and provides a list of identities which are used in the simplification of Boolean expressions. For simplification of complex Boolean expressions, the textbook also suggests the creation of a truth table. From this students can then identify if certain variables are not significant in the outcome of such expression. At the end of each section, it provides a list of questions to test the students' understanding and help them prepare for the COMP2 exam.

Since the start of this syllabus in 2011, there have been two COMP2 papers every year, except the first year. Students can access the papers from two years ago; therefore there are only five past papers that are publicly available to them. So far, there has been a question on logic gates every year[1]. Figure 3 shows an example question from the JUN13 paper. These are some pretty standard questions for the COMP2 exam, where students are asked to identify a logic gate from a truth table, simplify Boolean expressions or create the logic gate equivalent of a Boolean expression. Other questions include identifying an arithmetic function from a truth table and logic circuit and simplifying without using a truth table (JAN12), creating a truth table form a Boolean expression and identifying De Morgan's law (JAN13) and creating a Boolean expression from some text (JUN12).



6 (a) State the names of the logic gates represented by each of the three truth tables below.

Input A	Input B	Output
0	0	0
0	1	0
1	0	0
1	1	1

Logic gate name ....

6 (b) Simplify the following Boolean expressions.

6 (b) (i) $B \cdot (A+\overline{A})$

....

....

(1 mark)

6 (b) (iii) $\overline{B} \cdot (\overline{A+B})$

....

....

(2 marks)

6 (c) Draw a logic circuit for the following Boolean expression:

$Q = (A \oplus B) \cdot B$

You will need to make use of the symbols below when drawing your logic circuit.

Figure 3: an extract form the JUN13 COMP2 paper

[1] JUN11 Q3, JAN12 Q2, JUN12 Q8, JAN13 Q4 and JUN13 Q6

Mr Starkings or Mr Statham will often go through exam questions in lessons. For logic gates and Boolean expression type questions, this will involve drawing logic circuits, truth tables and the stages in simplifying a Boolean expression on the board.

Boolean algebra, logic gates and truth tables

This section will cover logic gates, Boolean algebra and truth tables for those who are unfamiliar. If you are familiar with these terms then feel free to skip this section.

Boolean algebra is the subarea of algebra in which the values of the variables are truth values true and false, usually denoted 1 or 0 respectively[2]. This means that instead of the numbers 1 to 9, you only have the 1 one and 0. Functions like addition, multiplication and log from elementary algebra do not exist in Boolean algebra, although you can still perform them by combining different Boolean functions, strictly speaking Boolean functions are only performed on the values 1 and 0, nor a list of them. A bitwise operator on the other hand takes a series of bits and performs Boolean operations on them. These are similar except they also include things like bit shifting and rotating.

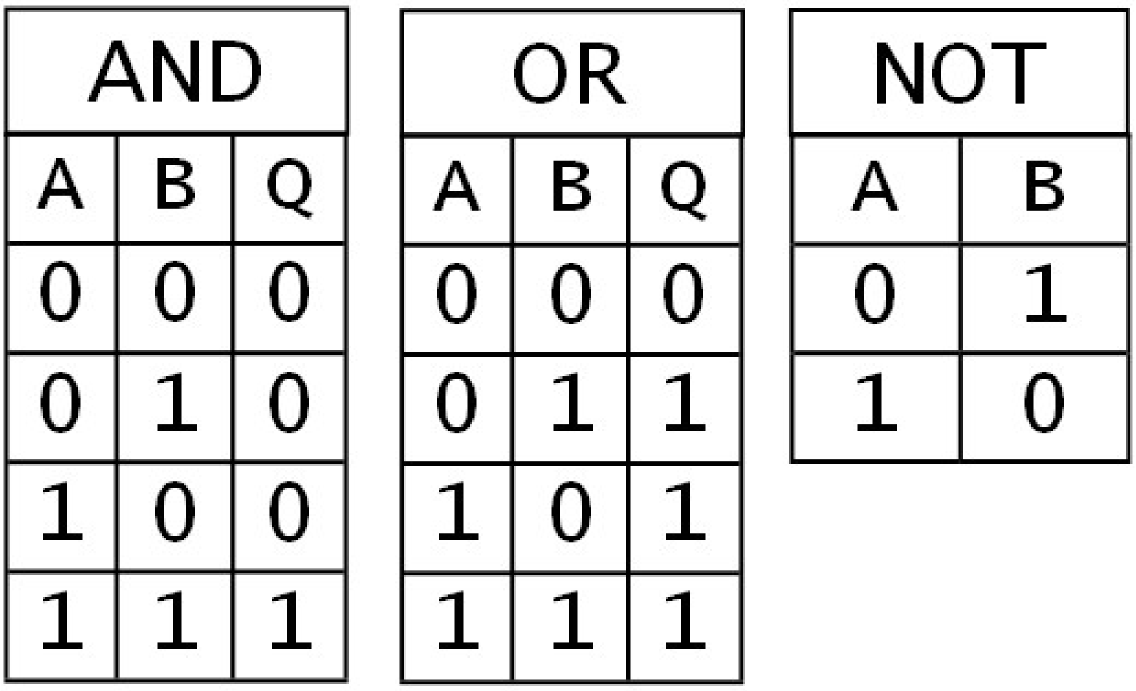
1 AND 1 = 1   1 AND 0 = 0

1 OR 1 = 1   1 OR 0 = 1

Figure 4: some results of binary AND and OR

The three most basic Boolean functions are AND, OR and NOT. By combining these you can make further functions like NAND, NOR and XOR. Figure 4 shows the results of AND and OR. As you can see, there are only 4 possible inputs for AND and OR, which we can represent using a table instead of a large list. This is called a truth table. Normally we use capital letters from the beginning of the alphabet to denote inputs, the textbook likes to use Q for outputs although this is not known standard. Figure 5 shows the truth table for the 3 most basic Boolean functions – AND, OR and NOT.

[2] Taken from http://en.wikipedia.org/wiki/Boolean_algebra



AND		
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

OR		
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

NOT	
A	B
0	1
1	0

Figure 5: the truth tables for AND, OR and NOT

For more complex Boolean algebra that is a combination of lots of functions you can write them as an expression. A Boolean expression for example could be A AND B OR C. Sometimes the Boolean operators are written out as words, normally in capital letters but most of the time you can use symbols for them.[3] AND is written as a round dot $(A \cdot B)$, or when typing a star $*$ or sometimes nothing at all. OR is written as a plus $(A + B)$. NOT is written as an over bar $(\bar{A})$ but when typing you can also use a tilde ~ in front or a prime $'$ after. I will use $\cdot$ for AND, $+$ for OR and an $\overline{over\ bar}$ for NOT. XOR for example can be written as $(A \cdot B) + \overline{(A + B)}$ and sometimes the circled plus $\oplus$ is used to denote it.

Logic gates can be seen as a visual representation of Boolean expressions. Currently there are two standards for drawing logic gates according to the ANSI/IEEE Standard 91a-1991: "Distinctive Shape" and "Rectangular Shape". For the write-up of this project, I am going to use distinctive shape gates, which is used in the COMP2 exam and textbook. Figure 6 shows the shapes for all logic gates.

[3] The symbols used are standard for engineering and computer science, in pure logic is it more common to see $\wedge$ for and, $\vee$ for or and $\neg$ for not

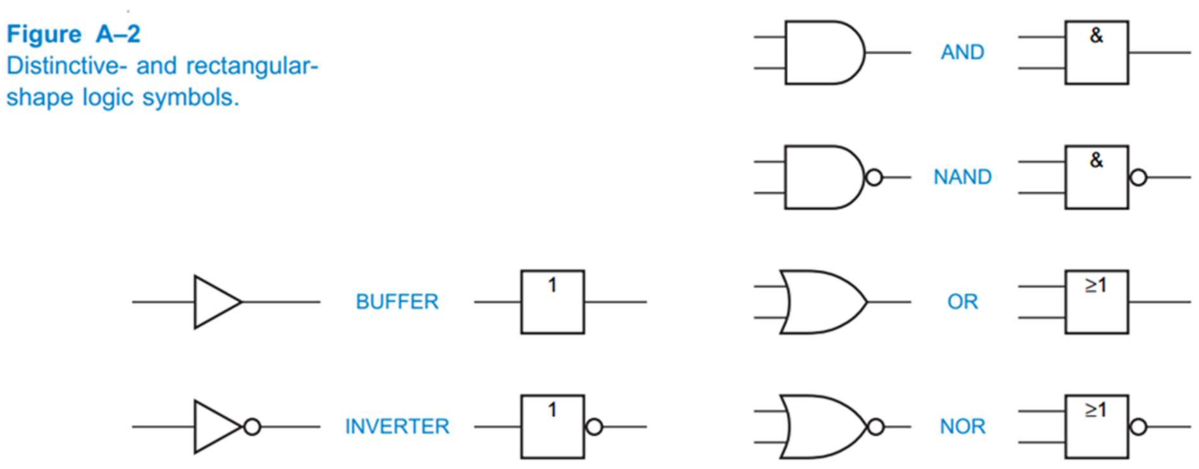


Figure 6: figure A-2 taken from the IEEE standard symbols for logic gates

Apart from representing Boolean expressions, as the output of a logic gate can go back into itself, it can actually represent much more. For example, if you take two NOR gates and feed the output of one into the other you create a NOR latch. Then if you add a few more gates you can create a D latch. Using similar constructions you can create a flip flop. This is not part of the AS COMP2 syllabus.

$$\overline{A \cdot B} \equiv \bar{A} + \bar{B}$$
$$\overline{A + B} \equiv \bar{A} \cdot \bar{B}$$

Figure 7: De Morgan's Law in symbols

De Morgan's law is one of the many Boolean laws. Using Boolean laws, you can simplify any given Boolean expression. You can also create any Boolean expression by only using {NOT, AND} (or only {NAND}) or {NOT, OR} (or only {NOR}). A set of Boolean operators which allows you to calculate any Boolean expression is a functional complete set. Although this is also not part of the AS COMP2 syllabus, the textbook does mention the idea when talking about De Morgan's law.

Problems with the Current System

The current system where logic gates, truth tables and binary expressions are drawn on the board is slow and without a tool like logic.ly, a small change will require lots of rubbing out and rewriting, especially for large logic circuits or binary expressions.

Currently there is no easy way to confirm the simplification of a binary expression or logic circuit, check if your truth table is correct or if the binary circuit created from a truth table is correct without asking a teacher or slowly checking it.

The syllabus is quite new, there are only a few practice papers.

Logic.ly

Logic.ly does not include the truth tables or binary expressions for the logic circuits you create and does not allow for the creation of logic circuit from a binary expression. Logic.ly also includes more complex things like flip flops, clocks and busses. These can distract the student when they are using logic.ly for AS computing. Figure 8 shows a ripple counter which uses a D Flip Flop. This is more than AS students need to know and although interesting is not required to pass the COMP2 exam.

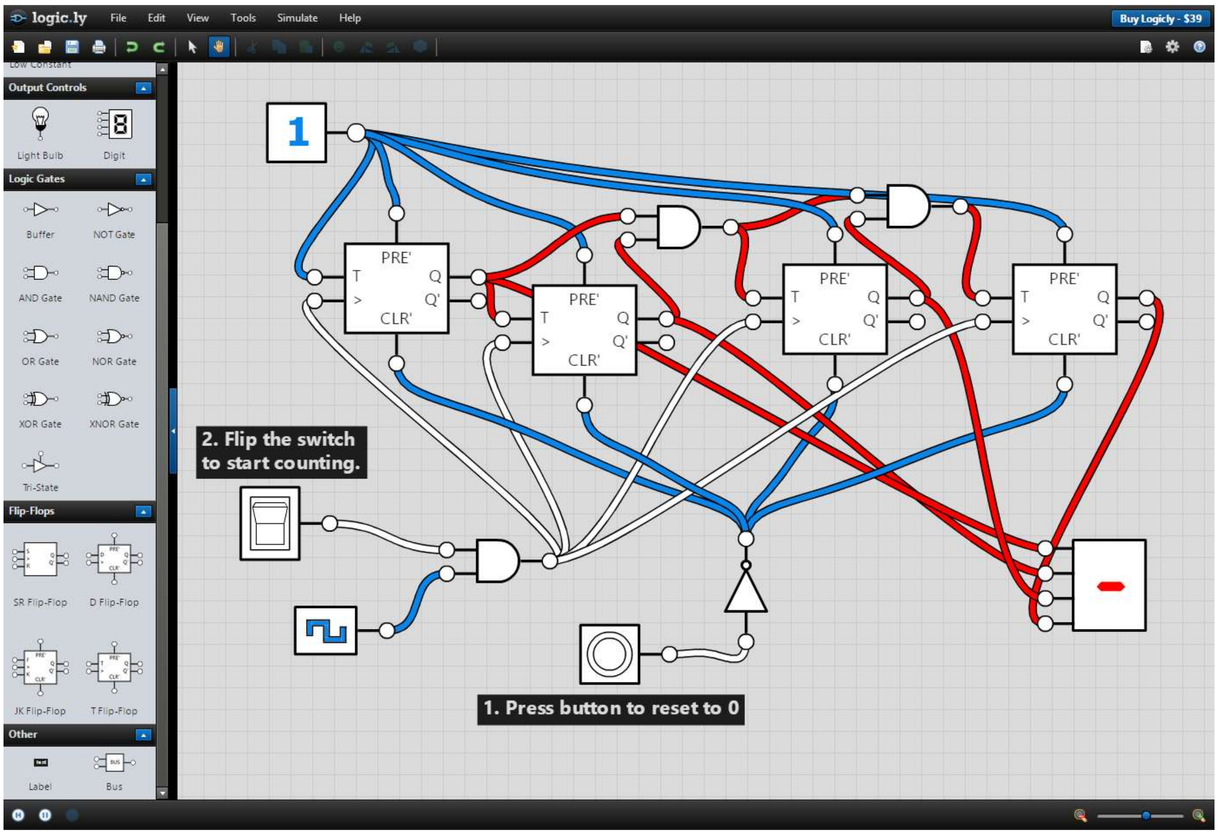


Figure 8: one of the examples show on logic.ly's start up, a ripple counter

In addition to this, because the school is using the free online demo version of logic.ly, it does not allow students to save given circuits, export integrated circuits (logic.ly's system of combining gates together) or print the given circuit.

The program is not very intuitive in general:

- Sometimes when selecting a gate, it brings up the properties for the gate but sometimes it doesn't.
- The differences between blue, red and white wires are not explained.
- You have to select the pan tool to move the screen around
- Gates do not output unless connected to some kind of input (even the NOT gate)
- To create an integrated circuit, you must use switches as inputs and bulbs as output, which you must name

Wolfram Alpha[4]

The only tool that does simplify Boolean algebra that I have found is Wolfram Alpha. To get Wolfram Alpha to show the stages in simplifying a Boolean expression requires you to pay a

[4] The computational knowledge engine that does (nearly) everything

monthly fee for Wolfram Alpha Pro. It can also generate truth tables for Boolean expressions but it also does not show the stages to arrive there, which is not an option even for those who subscribe to Wolfram Alpha Pro.

Unfortunately as it is a general computation engine it does not recognise + or ·, and prints using the pure logic notation which is not taught by the AS syllabus. This may cause confusion when answering questions as the use of proposition logic symbols are never mentioned in the COMP2 paper or mark scheme.

Wolfram Alpha only lets you enter a Boolean Expression, and sometimes can be a little quirky when interpreting your input.

Prospective Users

The client is Mr Starkings and Mr Statham. Being computing teachers, both are very competent at using computers. At school all the computers run Windows 7. The teacher has access to a computer that is connected to a projector while teaching, which he can connect to using an iPad. The main users of this program will be AS students learning COMP2. During lessons each student has access to their own computer.

I decided to take a poll of my current set about which operating system they used, and the majority used windows 7.

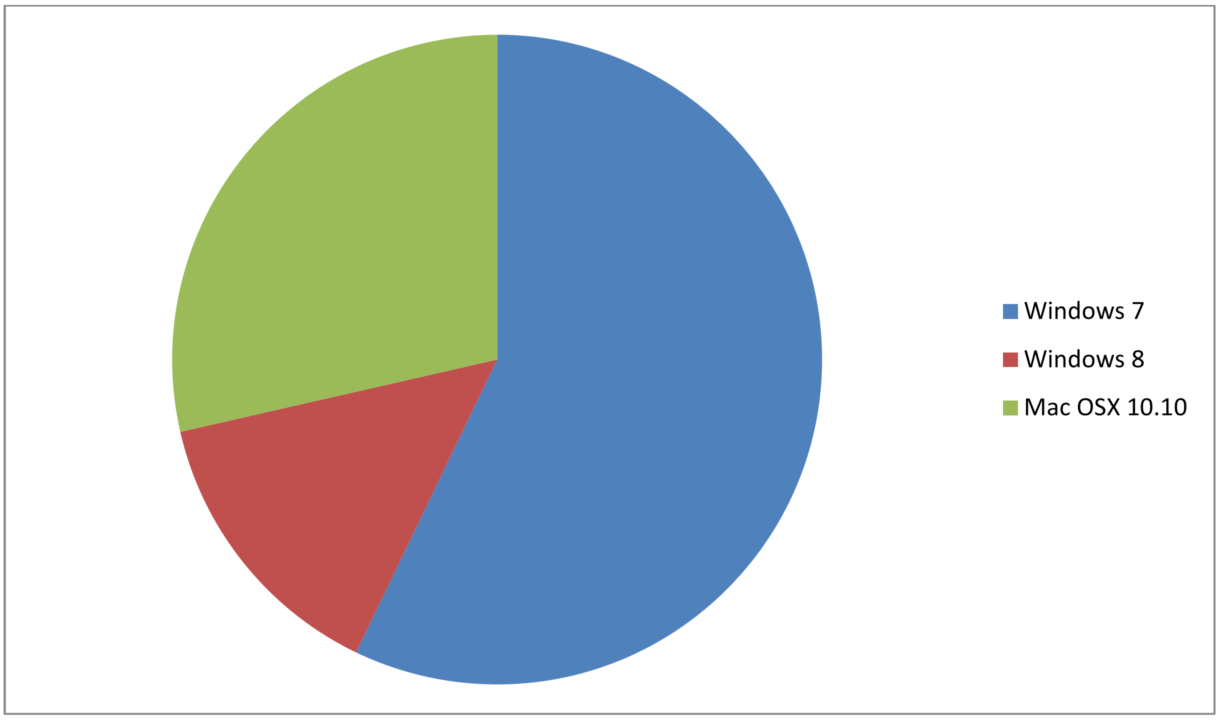


Figure 9: pie chart for which operating system is used in my set

This is similar to the research done by www.netmarketshare.com, which lists windows as 90% of all operating systems used by consumers.[5]

[5] 58% Windows 7, 17% Windows XP, 11% Windows 8.1, 3.5% Windows 8, 2% Windows Vista and the majority of the rest is Max OS X.

Acceptable Limitations

The program does not need to go beyond the AS COMP2 syllabus; this means it does not have to include things like flip flops and latches.

Hardware and Software

It does not need to be very memory efficient as school computers have 3GB of ram. It does not need to run on all operating systems, at school all computers run windows 7 and most programs written for it will be compatible with other versions of windows.

Programming Difficulty

The solution to the problem cannot be more complex than my programming capability, for example if I attempted to create a program that could parse a series of text to create a Boolean expression or vice versa, it would not be possible with the scope of my current knowledge.

Time Constraints

This program must be finished by the deadline, with enough time to ask for user feedback to make adequate tweaks. This includes not only the programming but also testing, maintenance and user guide.

Objectives

1. The system must work at school
2. Boolean expressions should be formatted correctly using over bars, dots and plus signs
3. Boolean expressions that are inputted must be able to be simplified as much as possible
4. The system must have a list of Boolean identities that can be shown to the user
5. Boolean expressions must be able to be converted into truth tables, with all the steps
6. The system must be able to simulate a logic circuit
7. Logic gates must be of the ANSI/IEEE Standard 91a-1991: "Distinctive Shape"
8. Given a Boolean expression and logic circuit, the system should be able to verify if they are equivalent
9. The system should be able to create a Boolean expression from a logic circuit and vice versa
10. Boolean expressions should be able to be saved
11. Logic circuits should be able to be saved
12. Truth table should be able to be saved

13. The system must have an example for De Morgan's Law
14. The system could create questions for the user to answer
15. The system could have a tutorial section which teaches the basics of Logic Gates
16. The system could have a tutorial section which teaches the basics of Boolean algebra

Proposed Solution

Logic.ly

Purchase

The school could purchase logic.ly, although this does not give them many more features. It would still not allow you to simplify Boolean expressions and create truth tables. It would also be very expensive as the school would need to purchase a license that allows them to use it in the classroom and for students to use it at home.

Other

I could attempt to create an add-on for logic.ly that provides all of the missing features. Unfortunately the program is not open source so this would require decompiling it. It would also be illegal for me to distribute the program. As the program is written in ActionScript, I would also have to purchase Adobe Flash, which costs over £300.

Mobile Application

As most people nowadays have a smartphone or tablet, an app for a mobile device would allow almost all users to use it. Currently there is a trail period where select students and teachers at our school are given iPads. Creating an app would allow students to learn on the go and could be possibly integrated into "firefly", the school's virtual learning environment.

Android

Android development is free and although would not take advantage of the iPads, it is still the most popular mobile ecosystem. To submit your app on the Google Play Store on the other hand costs a $25 developer fee. This does not mean you cannot side load an application, but this may be difficult for those who are less familiar. Android applications are written in Java and there are many libraries for the UI. Unfortunately I am not very familiar with Java and do not have an android device to test the app. Android is also a very fragmented ecosystem so an app may work fine on one device but not at all on another.

iPhone/iPad

Apple app development for the iPhone and iPad technically costs $99, which includes the developer fee to submit apps to the app store. In theory one could find a copy of development kit online but this would be illegal and side loading apps to apple devices is very difficult. I do not have an apple device either so this is not a very feasible solution.

Web App

A web app would work well as it would be usable on all devices. Unfortunately this would require the hosting of a website. Although this could be provided by the school, it would also mean coding in JavaScript. I am not very familiar with JavaScript and even less to with the new HTML5 features. Web development to ensure that the app works on all browsers, devices and aspect ratios will be very difficult and time consuming.

Windows App

For development on windows, there are many options for programming languages and windows managers. Applications made for one version of windows will normally work for another version, although this has some exceptions. Some programming languages and windows managers are cross platform which could allow me to create a program that works on any operating system.

Programming Language

Last year I learnt to program using Visual Basic. Visual Basic is a programming language made by Microsoft. It has a very nice GUI builder and coding IDE. I am reasonable familiar with VB but for this project, I will need to be able to draw logic circuits, which is very difficult with VB. VB does not provide the ability to do custom rendering and cannot run on other operating systems.

I am very familiar with C++. If I used C++, I would have an option of windows managers. There are many options to choose from including Microsoft's .NET, which is used by VB. This would allow me to use the same IDE as VB, which can speed up designing the GUI. Another option is Qt, a cross platform library. It is similar as it also provides a GUI builder and some base objects. It is also much easier to create custom shapes. Lastly I could use something like SDL or a wrapper for SDL like SFML. This would give me the most control over rendering but it would mean that I have to program my own classes for most of the GUI. Unfortunately C++ is a compiled language so if I wanted to code at school, I would have no way of testing it as students do not have permission to compile files.

Lastly, the programming language I have chosen is Python. Python is a high level interpreted language that is very easy to use. As it is written is C++, it can easily interface with any C or C++ library. This means that I had the option of using a python wrapper for Qt or SFML. Instead I chose to go with Pygame, which similar to SFML is a wrapper for SDL. The reason I have chosen it is because pySFML is for SFML version 1.2 and is no longer supported while pySFML2 for SFML version 2.0+ is still in development. I am also more familiar with Pygame as I have used it before. Although this means that I will have to write code for the GUI, it also gives me more control over the rendering in general. For solution I choose I still would have to write some custom rendering code to ensure objective 2 is met.

Object Analysis Diagrams

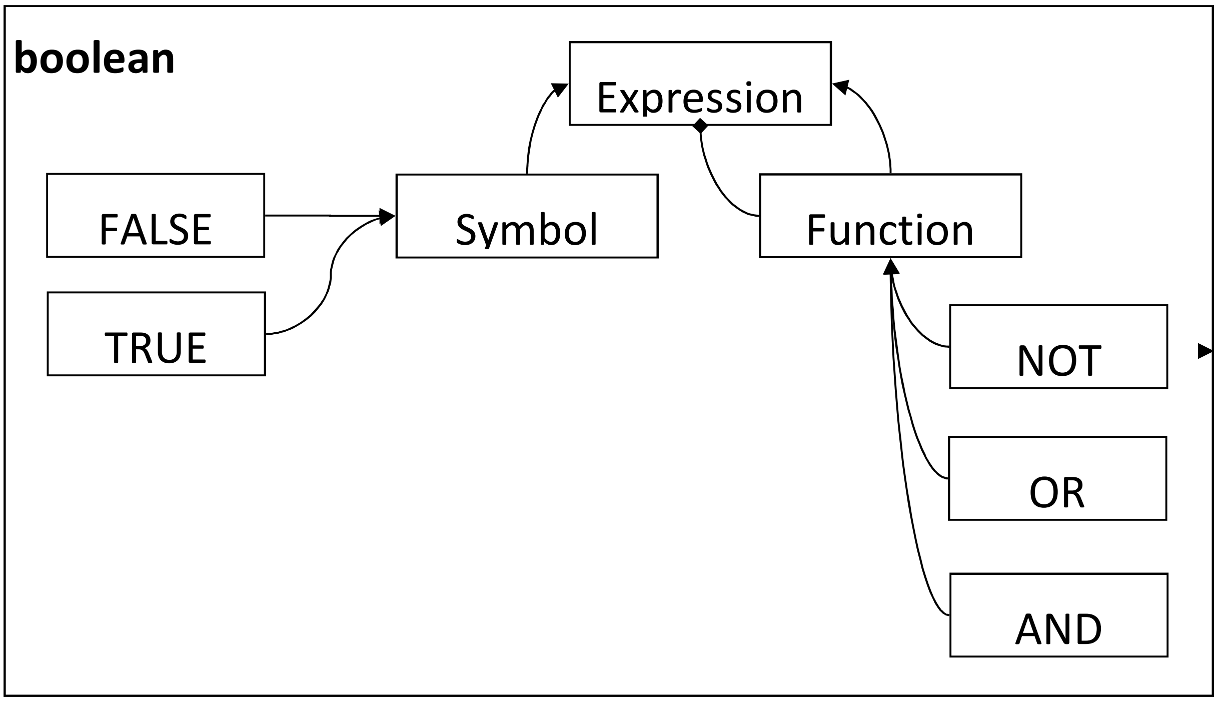


Figure 10: an object analysis diagram for Boolean expressions

Figure 10 shows an object analysis diagram Boolean expressions. An arrow shows inherence and a diamond shows aggregation. The base class Expression is inherited by Symbol and Function. Special cases of Symbol are TRUE and FALSE, which are constants. AND, OR and NOT are functions, so they inherit from Function. Although I only need NOT and one other function, I have decided to include all three for ease of programming.

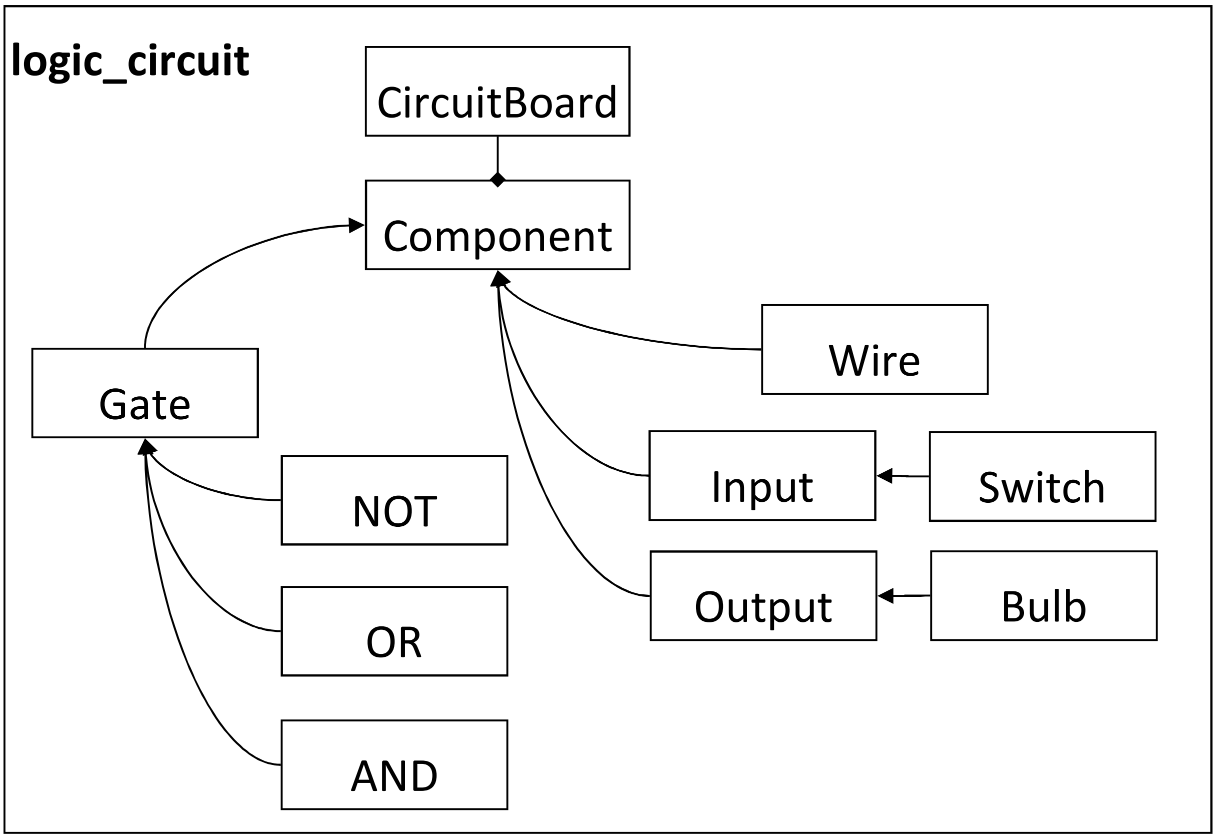


Figure 11: an object analysis diagram for logic circuits

Figure 11 shows the object analysis for logic circuits. Every part of a logic circuit is inherited from Component, as all components have an input and output. Gates contain Expressions, and are connected together by Wires. Once again I have created the 3 most basic expressions as logic gates, AND, OR and NOT. Instead of just letting Switch or Bulb inherit from Component, I have decided to include an Input and Output class in case there is a need to create different inputs or outputs.

A more in depth overview will be made in the design section.

Data

The type of program created will not need any kind of database to store information. The program run on the user's machine and will not need to connect to an online service.

Data Dictionary and Data Flow Diagram

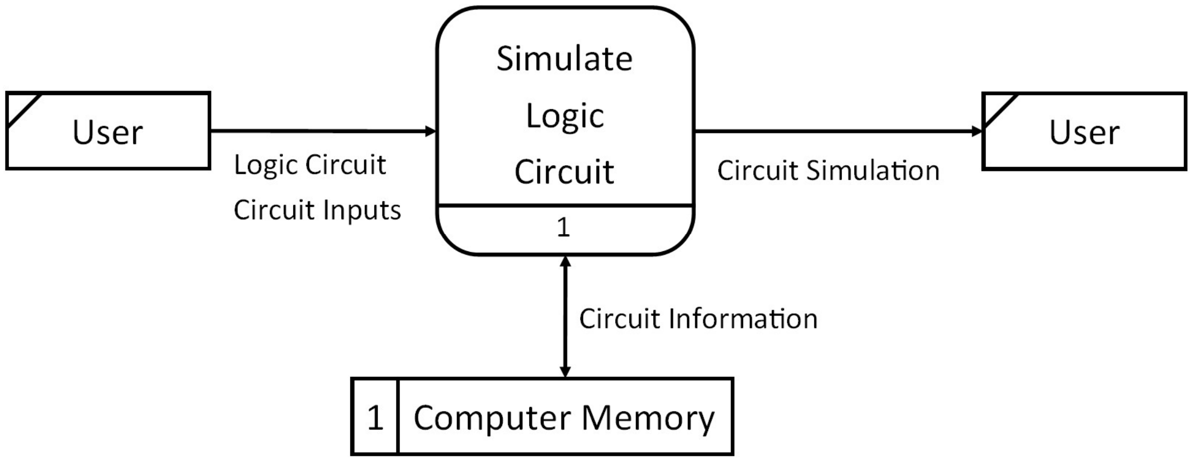


Figure 12: data flow diagram

A data dictionary is a very hard thing to create for such a dynamic program, there is no knowing how many logic gates are on screen at one time. The user simply uses the program and he receives an output via the simulation on the display. Therefore the simple data flow diagram would look like figure 12.

Objective 3 is about getting an input from the user. This is the only part of the program which requires the user to input information. All other inputs will be using the mouse to click buttons or select things. As python strings are dynamic, it is impossible to tell how large this value can get (in a 64bit system with 64bit python, a string can take all the remaining memory on the system) , but the minimum size of a string is 51 bytes according to the function sys.getsizeof.

Field Name	Field Type	Start Value	Validation
user_expression	string	“”	Must be a valid Boolean expression

Figure 13: data dictionary

Research

Questions with the end user

What is your existing system for teaching students Boolean algebra?
Statham: Normally there is some chalk and talk, board work mainly. We use some program (logic.ly) to aid and enforce the concepts that we teach. It is used for students to explore and practice but it doesn’t explain the concepts.
Starkings: A lot of board work and logic.ly.

What are the benefits of the current program?
Statham: Students can go at their own pace. It allows them to experiment about things outside of the syllabus.
Starkings: Logic.ly doesn’t quite match the syllabus.

What do you use to aid your teaching?
Statham: Just the program (logic.ly) and past paper questions.
Starkings: Logic.ly.

What new features would you like to see?
Statham: It would be good if there was a way to demonstrate de Morgan's law using logic gates. This is especially useful for non-mathematical people. It would also be nice there was a way to setup challenges, for example if it had to create a given output with logic gates. Practical applications of how logic gates could also help, for example if you could have some diagram for a car interior light to demonstrate the use of an OR gate.
Starkings: Rest the allowed logic gates to only the gates on the syllabus.

What area do students struggle the most at, or is hardest to teach?
Statham: The practical side of de Morgan's law and the algebra for non-mathematical people.
Starkings: Students struggle at converting a Boolean expression into a logic circuit. The hardest part to teach is accurately creating truth tables.

What existing features do you find the most useful?
Statham: Drag and Drop.
Starkings: The ability to answer questions in more than one way.

Why do you teach the truth table method of simplifying Boolean expressions when it could have all been done using the laws of Boolean algebra?
Starkings: Truth tables are often much easier for weaker students and may help point out a few things that are not obvious to them. It is especially helpful if there are many inputs but only few operations.

Raw data from OS questionnaire

Windows 7:	4
Windows 8.1:	1
Mac OSX 10.10:	2

Design

Modules Overview

In order to achieve the objectives, the program will be split into modules, each which attempt to achieve different parts of the objectives. Each module will fall under 2 categories: logic and rendering. The logic modules should be stand alone and work without and rendering modules. The rendering modules will depend on logic modules and pygame.

The file structure for development is shown in figure 14.

```
logic_gates
│   <module_name>.py
│   <module_name>_gui.py
│   setup.py
│
├───csetup
│       csetup.py
│
├───dist
│   │   logic_gates.exe
│   │
│   ├───images
│   │       <image_name>.png
│   │
│   └───saves
│           <saved_image>.png
│           <saved_logic_circuit>.lgs
│
├───images
│       <image_name>.png
│       vector_gates.svg
│
├───saves
│       test_load_file_<module_name>.lgs
│       test_image_equivalent_<module_name>.png
│
└───test
        test_<module_name>.py
```

Figure 14: file structure for development

Logic Modules

boolean.py

This module will contain all the necessary code to create, manipulate and parse Boolean algebra. This module targets objectives 3 and 5. Figure 15 shows the new proposed object analysis diagram for this module.

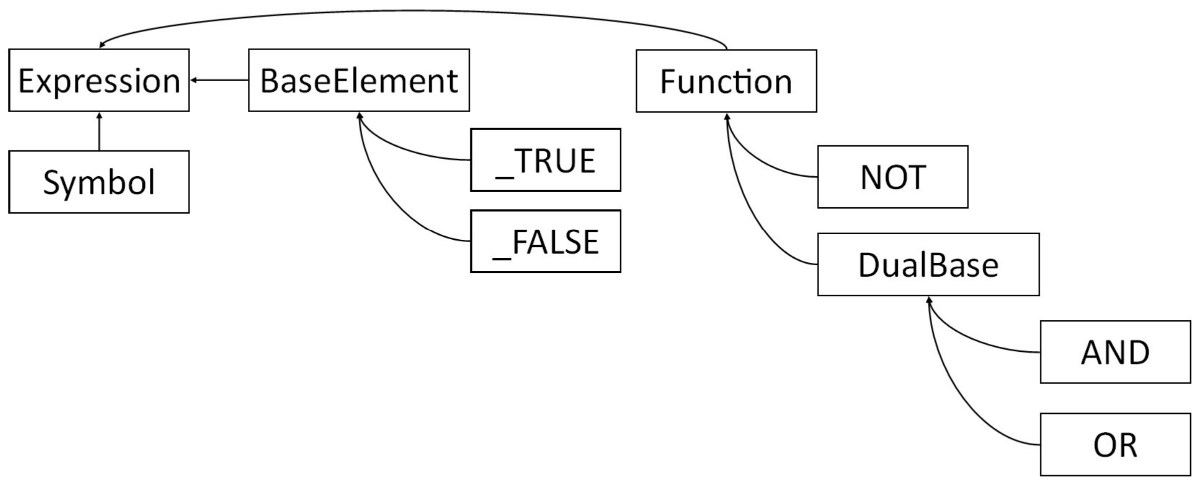


Figure 15: object analysis diagram for boolean.py

True and false are singletons therefore instead of creating new instance of them each time, a private class is made which then constant singletons are created from. It should also include a function to parse Boolean expressions from a given string and a function that returns a truth table given a Boolean expression.

logic_circuit.py

This module will include all the necessary classes to create and simulate logic gates. Figure 11 showed an early idea for an object analysis diagram. The new object analysis diagram is figure 19. In addition to classes, this module should also include a function that generates a circuit board from an expression and vice versa. This module targets objectives 6, 8 and 9.

Render Modules

alignment.py

This holds the global constants for alignments, which may be used in other render modules.

interfaces.py

This module contains the interfaces for things that can be rendered, can be interacted with and dragged. It also contains a container class which can contain renderable objects. Figure 16 shows the object analysis diagram for the interfaces. This module can implement a system for objective 11 by making all IRenderable objects saveable.

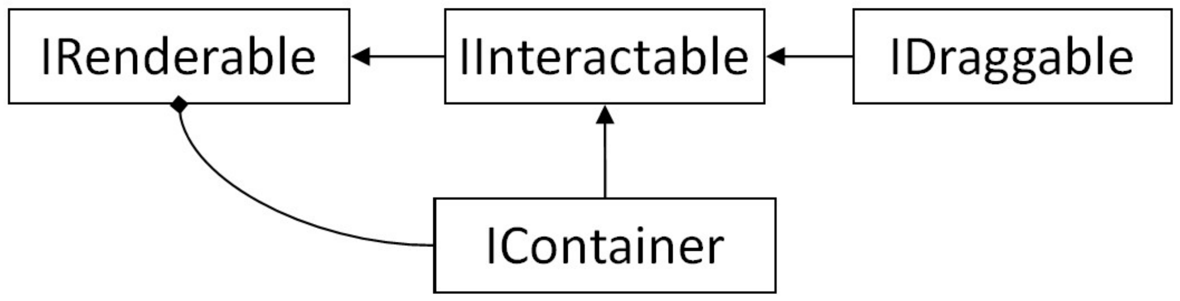


Figure 16: object analysis diagram for interfaces.py

text.py

This module contains classes that are wrappers around pygame fonts. By themselves, pygame fonts are difficult to manage so by creating a wrapper that also inherits from IRenderable, it makes them easier to use. It also contains the rendering code for Boolean expressions and truth tables. These need special rendering as python strings and Unicode strings do not support more than 2 over bars above letters. This satisfies objective 2, and will be used by other rendering modules when rendering Boolean expressions.

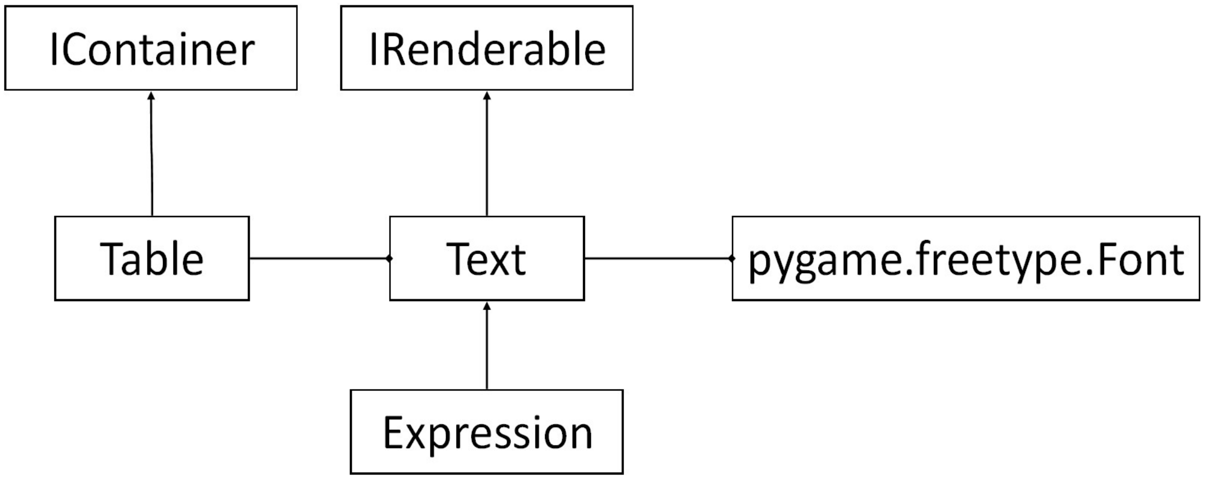


Figure 17: object analysis diagram for text.py

gui.py

This module contains some basic GUIs, an input box and text button. Although it is not strictly necessary to make the classes their own module, if the program needed more types of GUIs, they can also be added here.

logic_circuit_gui.py

This module contains the code to render logic gates. Standard symbols should be used to satisfy objective 7.

logic_gates.py

This module will contain the code that runs the program itself. This means it contains the logic loop, positioning of GUI elements, handling different screen resolutions, etc. It will attempt to satisfy any of the remaining objectives.

Other modules

setup.py

Although python is a very portable programming language, it is not very easy to install pygame. In order for users to easily install this program, py2exe will be used to create an executable which includes the python interpreter and required modules.

Algorithms

This section goes more in depth into each module, and outlines the complex algorithms that they use. I have split them into 2 sections again as the algorithms for rendering depend on pygame libraries. The style guide for programming will be similar to PEP8 except there is no enforced line limit. The most important part of this is that functions are snake_case, class names are CamelCase and module names are alllowercase.

Figure 18 shows the format that I will use for class definitions. It will be used in the following explanations. Classes will not list their inherited methods or fields.

class <class name> (<further information>)		
<type> <field name>	<comment or explanation>	
<class name> <return type> <method name>	<arg type> <arg1 type> <arg2 type>	A function called <class name> is the constructor. The type can be void. A function with multiple arguments is done like so.

Figure 18: format for detailed class definition

Logic Modules

boolean.py

For this project, I will be using an open source boolean.py which satisfies some of my criteria. In its original state, the program:

- Does not fully expressions that are simplified using de Morgan’s law.
- Does not print the expressions using dot and prime notation.
- Does not parse prime notation.
- Cannot create truth tables.
- Can simplify other kinds of expressions.
- Can parse “~” as NOT, “+” as OR, “*” as AND.

In order for this to be used, the first four statements must be achievable.

logic_circuit.py

Classes

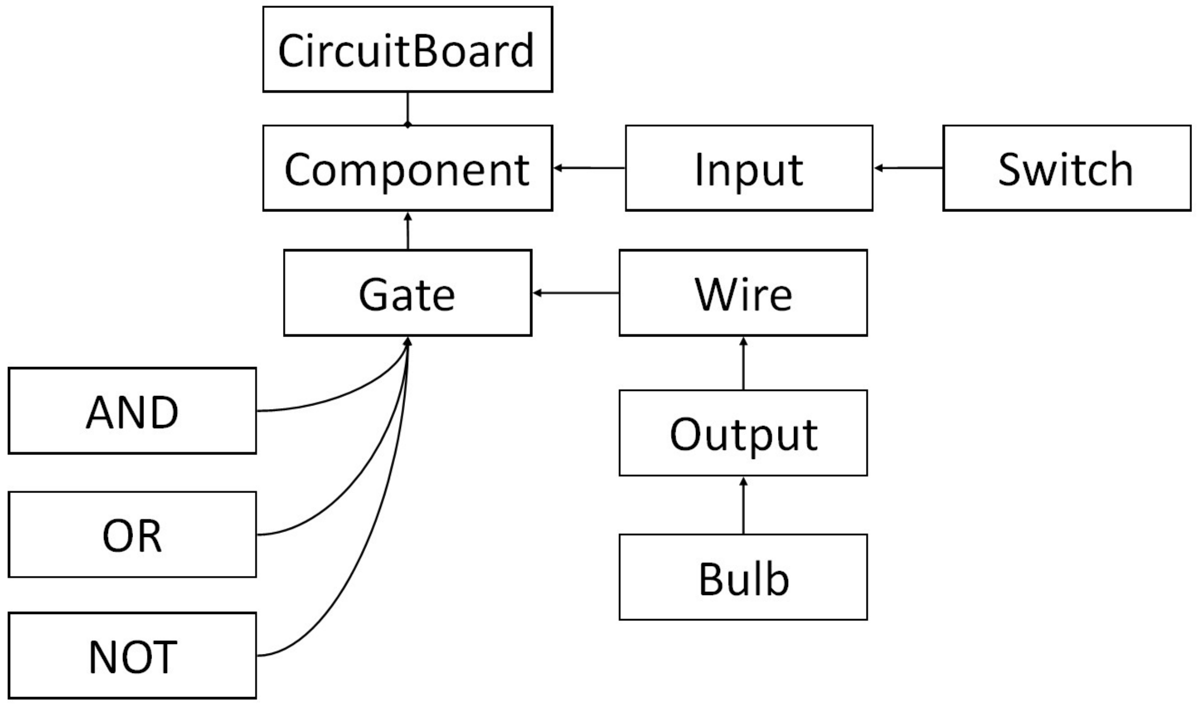


Figure 20: a new object analysis diagram for logic circuits

Figure 17 shows the new object analysis diagram for creating logic circuits. The changes that have been made are: Wire now inherits from Gate and output from Wire. The logic behind this is that wires, in a way are just repeaters (two not gates) and therefore inherits from gate. An output is the same as the wire in every way, except that it should not be able to output to other gates. This is why it now inherits from wire.

The following is a more detailed look at each class through the use of class diagrams.

The component class defines an object with an output and two functions: update and remove.

class Component (abstract)		
bool output	The output value of this component	
Component void update void remove	void void array	Re-evaluates its output Takes an array of components and removes itself from any components that are connected to it

Gates contains a Boolean expression, which is used to evaluate the output. It also defines the idea of components containing inputs. Each symbol in the expression is mapped to a component. The output of said component is then inputted into the expression when an update occurs.

class Gate (subclass of Component)

dict inputs expr expression	A dictionary mapping symbols to components The boolean expression that this gate evaluates	
Component	expr dict	 A dictionary mapping inputs to symbols

The following classes for AND, OR and NOT gates are not technically needed but they allow the programmer to create these simple gates easily.

class And (subclass of Gate)		
Component	array	An array of length 2, the 2 input components.

The same goes for or, and the not class only takes a component as an input instead of an array of 2 components. Wires are exactly the same as not gates, except their expression is simply a symbol. For ease of programming, they will also include a getter and setter for their input as they only have 1.

An input is something that can input true or false into a Boolean expression. They do not define any extra fields or methods. A switch on the other hand defines two functions, press and click.

class Switch (subclass of Input)		
press click	void void	Both functions toggle the output for the component. This function technically not necessary.

An output is no different from a wire. As the current class structure only keeps track of what is outputting into the component, there is no way strictly enforce this. They are only provided for convenience.

A circuit board inherits from an array. Depending on what programming language you use, this may or may not be the easiest thing to do. If this is difficult, then consider encapsulating the array or list type and provide the same functions that is does in your class.

class CircuitBoard (subclass of <array_like_type>)		
update remove	void component	Updates all components Calls the components remove method before removing it from the list

The reason that we override the remove function is so that we can call remove on the component, removing it the other components in the list.

The actual class code can be seen in the system maintenance section.

Functions

One of the objectives was to be able to convert from logic gates to an expression and vice versa. With this in mind, it makes sense for components to contain Boolean expressions.

This means that this module relies on the Boolean module. The module will contain 2 functions – expression and circuit_board.

Component to Boolean expression

Expression will take a component, and create a Boolean expression from it. This function is one way as the expression created will be simplified. The function will rely on a helper function which will recursively traverse the component. As this function is recursive I have not included the pseudo code, but it should be easy enough to translate from the python code:

```python
seen = set()

component_dict = {}
def recursive_expression(c):
    if c in (None, True, False):
        return boolean.Symbol(None)
    elif c in seen:
        raise RecursionError("The logic circuit is self refrencing and 
cannot be converted into a boolean expression")
    if not isinstance(c, Input):
        seen.add(c)

    if isinstance(c, Input):
        if c in component_dict.keys():
            return component_dict[c]
        else:
            component_dict[c] = boolean.Symbol(None)
                return component_dict[c]
    elif isinstance(c, (Output, Wire)):
        return recursive_expression(c.input)
    elif isinstance(c, Gate):
        subs_dict = {}
        for k, v in c.inputs.items():
            subs_dict[k] = recursive_expression(v)
        return c.expression.subs(subs_dict)
```

Figure 21: python code for the recursion in converting a component to Boolean expression

Please not that RecursionError has been defined previously in this module and is not a standard python exception. This is done so that if needed, you can catch only this exception.

The variable seen is created outside of the scope of the function to avoid passing it to the function every time. This means that the actual function in pseudocode looks something like this:

```
$component \leftarrow INPUT$
$seen \leftarrow set()$
$RETURN\ recursive\_expression(component)$
```

The python version has an additional feature of converting the symbols into ones that are easy to read. This depends on generators and itertools (part of the python standard library), therefore I have not included it in the pseudo code. The idea behind it is as follows:

$Define\ A^n\ as\ A\ repeated\ N\ times.$
$Return\ A^1 \rightarrow A^1, A^2 \rightarrow A^2 \ldots A^n \rightarrow A^n.$

The entire python code for expression is shown in figure 22. The yield statement turns the function into a generator, which is similar to defining a sequence in mathematics.

```
def expression(component, anonymous_symbols=False):
    seen = set()
    if anonymous_symbols:
        return recursive_expression(component)
    else:
        def letters_generator():
            def multiletters(seq):
                for n in itertools.count(1):
                    for s in itertools.product(seq, repeat=n):
                        yield "".join(s)
            letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
            return multiletters(letters)

        expression = recursive_expression(component)
        subs_dict = {}
        g = letters_generator()
        for s in expression.symbols:
            subs_dict[s]=g.__next__()

        return expression.subs(subs_dict)
```

Figure 22: python code that makes an expression look nicer by replacing symbols with letters

Boolean expression to circuit board

Like the expression function, circuit_board uses recursion to convert a Boolean expression into a list of components. Unlike the expression function, it will not attempt to simplify the expression before converting it into a list of components.

```
b = CircuitBoard()
symbol_dict = {}
def recursive_gate(e):
    if isinstance(e, boolean.BaseElement):
        s = Switch()
        b.append(s)
        return s
    elif isinstance(e, boolean.Symbol):
        if e in symbol_dict.keys():
            return symbol_dict[e]
        else:
            s = Switch()
            symbol_dict[e] = s
            b.append(s)
            return s
    elif isinstance(e, boolean.Function):
        if e.order == (1, 1):
            pre = recursive_gate(e.args[0])
            w = Wire(pre)
            b.append(w)

            s = boolean.Symbol(None)
            g = Gate(e.subs({e.args[0]: s}),
                     {s: w})
            b.append(g)
            return g
        else:
            input_dict = {}
            expr = e
            for arg in e.args:
                pre = recursive_gate(arg)
                w = Wire(pre)
                b.append(w)
                s = boolean.Symbol(None)
                expr = expr.subs({arg: s})
                input_dict[s] = w
            g = Gate(expr,
                     input_dict)
            b.append(g)
            return g
```

Figure 23: python code for the recursion in converting an expression into a list of components

The function itself will have a Boolean argument called bulb. When this is true the function will add a bulb to the end of the created circuit.

```
def circuit_board(expression, bulb=True):
    if isinstance(expression, str):
        expression = boolean.parse(expression, eval=False)
    if not isinstance(expression, boolean.Expression):
        raise TypeError(
            "Argument must be str or Expression but it is %" %
expr.__class__)

    b = CircuitBoard()
    recursive_gate(expression)
    if bulb:
        w = Wire(b[-1])
        o = Bulb(w)
        b.append(w)
        b.append(o)
    return b
```

Figure 24: python code for the function itself, which adds the additional feature of adding a bulb

Render Modules

alignment.py

This module is simply a bunch of constants for different alignments that an object can have. Aligning to the left or to the top takes the value 0, the middle or centre takes the value 1 and the bottom or right takes the value 2. Alignments are in the format (x, y) so something that is in the bottom left has the value (0, 2) for (left, bottom).

interfaces.py

This module depends on a lot of pygame classes, the alignment module and the pickle module from the standard python library. The pickle module allows you to convert (nearly) any python object into an array of byes that can be saved. It also means that you can load them from a file.

Classes

These classes depend on many pygame defined classes like surfaces and rectangles. A surface is similar an array of pixels, and a rectangle is defined by a position, width and height. Both classes have many functions that are used.

class IRenderable

rect hitbox	This defines the position, width and height of the object	
bool hidden	Hidden objects are not drawn	
bool dirty	If an object is dirty, it has to redraw itself	
surf surface	The surface that it draws on, this acts as a cache as when the object is not dirty it does not have to redraw itself	
IRenderable	rect bool	Creating renderable objects requires a hitbox and the Boolean for if it is hidden.
void update_surface	void	
surf render_surface	void	
void update	[IRenderable, …] [bool, …] [events, …]	This functions receives an array of other renderable objects, an array of keys pressed and an array of pygame events.
void move	(x, y)	
void align	(x, y) or (x, y, w, h) align	The first argument is either a position or a rect style object. Align is a value from the alignment module and aligning to rectangles means aligning to the inside of them.
void pickle	str	Takes a filename and saves itself.[6]

IInteractable is an interface for objects that can be interacted with. This means that this class defines some new functions like key_down, key_up, mouse_down, etc. All except the last 3 of these functions take the same inputs which are an array of other renderable objects (which may or may not include itself), an array of the keys that are pressed, the event that has triggered this function and an array of all pygame events, including this functions. For these functions, I will use the shorthand "<event>". The last 3 functions do not take the current event as an argument. For these functions I will use the shorthand "<update>" as they take the same arguments are the update function (inherited from IRenderable).

class IInteractable (subclass of IRenderable)		
bool selected		
void key_down	<event>	The first 2 functions are virtual. The other functions handles the selection of objects. Objects are selected if the object receives a left click event that is inside of its hitbox. This also beings the object to the front. While an object is selected, it will mark any mouse motion events as handled to prevent 2 objects attempting to handle the same mouse motion event.
void key_up	<event>	
void mouse_motion	<event>	
void mouse_down	<event>	
void mouse_up	<event>	
void on_select	<update>	
void on_delesect	<update>	
void while_selected	<update>	

IDraggable is a simple class that adds code to allow objects to be dragged.

[6] IRenderable objects leave their children behind.

class IDraggable (subclass of IInteractable)		
bool dragging		
void on_drag void on_release void while_dragging	<update> <update> <update>	Functions to handle events.

IContainer contains renderable objects. It itself is also an object that can interacted with. Objects inside of this class cannot interact with those outside of it. This is useful as it separated objects into their own container. It contains another pygame class, color which is in the format (red, blue, green, alpha) with values from 0 – 255. The alpha is optional and by default is 255.

class IContainer (subclass of IInteractable)		
array renderable_list colour bgolor	A list of renderable objects The background colour for this container	
IContainer	rect bool array colour	Takes a hitbox, hidden, array of IRenderables and a background colour as its constructer arguments.
void update_renderable_list	void	This virtual function is used by subclasses if they need to recreate the entire renderable_list, similar to the update_surface and render_surface methods that are inherited from IRenderable.

Containers become dirty if any of the objects in their renderable_list are dirty. A background colour is required because when need to redraw the surface, paint over the existing one.

text.py

Classes

These classes depend on some kind of font rendering library. For the examples I will use pygame's freetype library.

Pygame's freetype library contains a font class, which has the attributes for size, background colour, foreground colour, font file, etc. It has a method called render_to which takes a surface, position and text and draws the text in the specified font to the surface. It does not do word wrapping or multi line rendering.

If your font library includes things like word wrapping, then you do not need to define the Text class yourself. All classes in this module cannot be interacted with. This is because their implementation makes all text align to the top left of their hitbox.

class Text (subclass of IRenderable)

font font	A pygame freetpype font	
Text	font str pos	The font used to render. The string is the text to be rendered. The top left of the text

Expressions are a special case of text. They rely on the Boolean module as they need to be able to find the NOTs in an expression. A standalone function could also be written, and is included in the functions section of this module.

class Expression (subclass of Text)		
expr expression	The expression being rendered	
int max_not_count	expr	A helper function to find the number of NOTs in an expression.
surf overbar_surface	void	A helper function to generate a surface of the over bar if it covers the entire text.
array fit_surface	void	A helper function that returns what would be the width and height of the surface.

Finally, the truth table class is a container that contains multiple functions. This needs to be implemented even if your library includes a table like objects as the table contains expressions which have to be rendered differently.

A more generic Table class could also have been implemented, but as this program only requires the rendering of truth tables, I have decided to only create a truth table class. It also simplifies the code needed as I would have to rewrite most of the algorithms to accommodate for the larger size of expressions that contain NOTs.

Tables are defined in the Boolean module as an array of dictionaries, with the dictionary keys are the headings and the index of the array as rows.

class TruthTable (subclass of IContainer)		
expr expression table table	The expression the truth table is for. The truth table for the expression	
array box_size	void	This function returns the width and height for the largest renderable object.
array fit_surface	void	This function returns the width and height of the surface needed to fix the entire table.

Functions

As mentioned in the classes section of this module, a lot of functions will depend on the font rendering library used.

Rendering Multiple Lines

If your font library doesn't allow for multi-line rendering, then you need to implement the following algorithm:

```
$text \leftarrow INPUT$
$lines \leftarrow text.split('\backslash n')$
$height \leftarrow 0$
$FOR\ line \leftarrow lines[0]\ TO\ lines[last]$
  $IF\ font.getheight(line) > height:$
    $height \leftarrow font.getheight(line)$
$FOR\ i \leftarrow 0\ to\ numberof(lines)$
  $x = 0$
  $y = height * i$
  $font.render\big(lines[i], (x, y)\big)$
```

The python implementation looks very similar and can be found in the system maintenance section.

Maximum not count

This recursive function looks at an expression and calculates the maximum number of NOTs that the expression contains. For example the expression $A + \bar{B}$ has a max not count of 1 while $\overline{A + \overline{\bar{B} + \overline{D + \bar{E}}}}$ has a max not count of 4. This can be seen as the height the over bar must go if the outer function of this expression is a NOT.

Rendering an expression

If you do not have a Boolean module, you can still render an expression with the correct over bars. This method involves looking at expressions as strings and the python code is included in the system maintenance section. If your font library can handle over bars then you do not need the first few lines of the algorithm:

Render your text underlined using your font library. Measure the minimum vertical distance from the beginning of the underline to the start of the letters, saving this value. Measure how thick the underline and save this value.

Convert the NOTs in your expression into a prefix symbol like "~".
Find the last occurrence of the new NOT symbol in your expression and save the index.
While there are still new NOT symbols in your expression, calculate the bounding box of the font before index and the bounding box of the rectangle for the first sub-expression found after the index. Calculate the height multiplier for your sub-expression by splitting the sub-expression into further sub-expressions and calculating the height multiplier for them. Render a rectangle that is above your sub-expression, remembering to shift it to the left by the number of NOT symbols before the index times the width of the NOT symbol. Then remove the NOT symbol and update the index.

Instead of including the python code here, as it does not depend on any other modules I have decided to add it to the system maintenance section.

With access to the Boolean module, the rendering itself can be done using a recursive function. This function depends on the class methods and only renders the over bars. An overview of the algorithm in words is:

Take an expression and the co-ordinates for the top left of the renderable area
If the outer function of this expression is a NOT, calculate the max not count for this function and draw a line that is the height of the over bar times the max not count above the top left position, ensuring that you crop the over bar so that it is the same size as the expression without the NOT symbol.
For every argument of this function, call this function again, ensuring that when you pass the co-ordinates, you pass the top of the renderable area above the beginning of the argument as the top left of the renderable area.[7]

The python code is part of the expression class and overrides update_surface from IRenderable. I have left the debug code inside but commented out to help those who are trying to understand the function.

```
def update_surface(self):
    interfaces.IRenderable.update_surface(self)

    # Cache the overbar surface before the update_overbar function. The
overbar surface
    # depends on the font size and expression, so it should not change
during an
    # update_surface call.
    overbar_surface = self.overbar_surface()

    # pygame.freetype.Font.render_to only fills the boxes for each line
    self.surface.fill(self.bgcolor)

    # DEBUG CODE
    # DEBUG_ORANGE = pygame.color.THECOLORS["orange"]
    # outline = pygame.rect.Rect(0, 0, self.hitbox.w, self.hitbox.h)
    # pygame.gfxdraw.rectangle(self.surface, outline, DEBUG_ORANGE)

    # recursively draw overbars
    def update_overbar(expr, top_left):
        # DEBUG CODE
        # DEBUG_TEXT_SIZE = str(expr).replace(boolean.NOT.operator, "")
        # DEBUG_TEXT_SIZE = self.font.get_rect(DEBUG_TEXT_SIZE)
        # DEBUG_TEXT_SIZE = (DEBUG_TEXT_SIZE.w, DEBUG_TEXT_SIZE.h)
        pygame.gfxdraw.rectangle(
            self.surface, pygame.rect.Rect(top_left, DEBUG_TEXT_SIZE),
DEBUG_ORANGE)
        if isinstance(expr, boolean.NOT):
            overbar_height = self.max_not_count(
                expr) * overbar_surface.get_rect().h
            dest = (top_left[0], top_left[1] - overbar_height)

            text = str(expr).replace(boolean.NOT.operator, "")
            expr_rect = self.font.get_rect(text)
            area = (0, 0, expr_rect.w, overbar_surface.get_rect().h)

            self.surface.blit(overbar_surface, dest, area)
            if isinstance(expr.args[0], boolean.NOT):
                update_overbar(expr.args[0], top_left)
            elif isinstance(expr.args[0], boolean.DualBase):
                # The "size" of a character depends on the next character
```

[7] You know what, it might be easier just to attempt to understand the python code.

```
                # If the argument of a NOT is a DualBase, it will always
have parenthesis
                # surrounding it
                arg_rect = self.font.get_rect(text[1:])
                parenthesis_w = expr_rect.w - arg_rect.w
                new_top_left = (top_left[0] + parenthesis_w, top_left[1])
                update_overbar(expr.args[0], new_top_left)
        elif isinstance(expr, boolean.DualBase):
            # The actual printed args, with extra parenthesis
            args = []
            for arg in expr.args:
                if arg.isliteral or isinstance(arg, boolean.NOT):
                    args.append(str(arg).replace(boolean.NOT.operator, ""))
                else:
                    args.append(

"({})".format(str(arg).replace(boolean.NOT.operator, "")))

            for arg in expr.args:
                if isinstance(arg, boolean.Symbol) or isinstance(arg,
boolean.BaseElement):
                    # End of the line, don't need to do anything
                    continue
                # Bounding box for the whole expression
                expr_rect = self.font.get_rect(
                    str(expr).replace(boolean.NOT.operator, ""))
                # Text for the arg and every arg after it
                after_text = expr.operator.join(
                    args[i] for i in range(expr.args.index(arg),
len(expr.args)))
                # Bounding box for the arg and every arg after it
                after_rect = self.font.get_rect(after_text)
                # Character size depends the next character
                w = expr_rect.w - after_rect.w
                new_top_left = (top_left[0] + w, top_left[1])
                # Literals are not surrounded by parenthesis
                # The NOT function puts an overbar around the parenthesis
                if arg.isliteral or isinstance(arg, boolean.NOT):
                    update_overbar(arg, new_top_left)
                else:
                    # Correct poisition for extra parenthesis
                    expr_rect = after_rect
                    after_rect = self.font.get_rect(after_text[1:])
                    parenthesis_w = expr_rect.w - after_rect.w
                    new_top_left = (
                        new_top_left[0] + parenthesis_w, new_top_left[1])
                    update_overbar(arg, new_top_left)

    text = self.text.replace(boolean.NOT.operator, "")
    overbar_height = self.max_not_count(self.expression) *
overbar_surface.get_rect().h

    x = 1
    y = 1 + overbar_height
    # Draw the expression
    self.font.render_to(self.surface,
                        (x, y),
                        text,
                        bgcolor=self.bgcolor)
    # Actually draw the overbar
    update_overbar(self.expression, (x, y))
```

Figure 25: python code for rendering an over bar above an expression

With all of the debug code on, this creates the image in the figure below. As you can see rectangles are drawn for each sub-expression, and for some of them an over bar is drawn above them. The Boolean expression was entered as:

$$\sim A * \sim(A + A) + \sim(\sim A * A) + \sim\sim\sim(\sim A + \sim\sim(A * \sim\sim A))$$

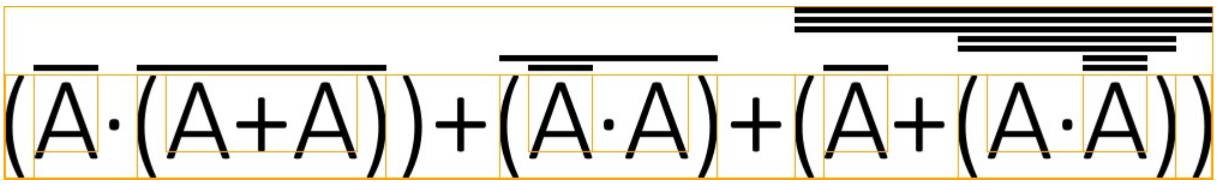


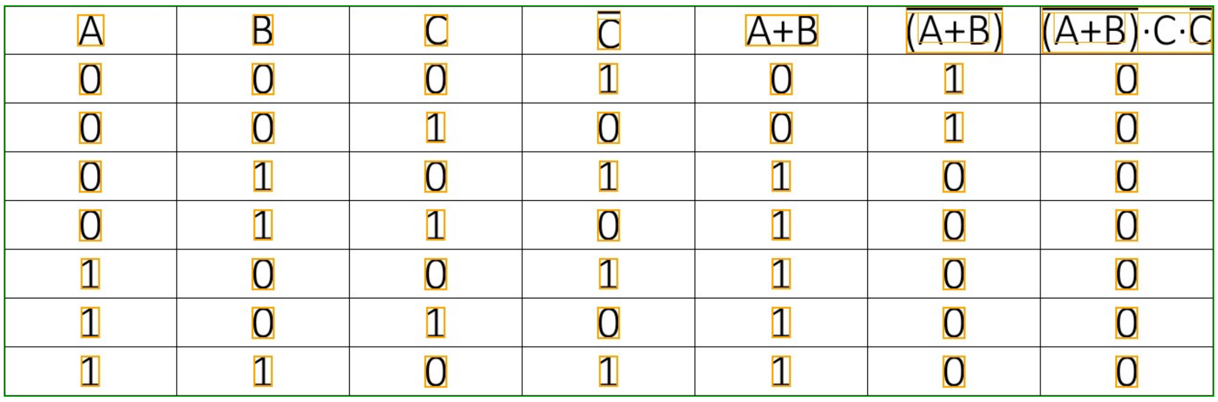
Figure 26: the resulting image when the debug code is left on

Rendering an truth table

To render a truth table, you need to know how to render a Boolean expression. This is explained above. These simple algorithms for rendering tables make each box the same size – the size needed to fit the largest object in the table. Then it is just an easy matter of aligning every renderable object to the correct location and drawing some lines.

To make it look nicer, you can sort the heading into symbols alphabetically followed by the length of the expression as a string. This will mostly ensure that things on the right hand side of the table depend on things on the left.

I have not included the python code here due to its simplicity, but it will be in the system maintenance section. The resulting image (with the debug code for drawing Boolean expression enabled) should look something like figure 27. From this you can easily tell that this expression is always false.



A	B	C	$\overline{C}$	A+B	$\overline{(A+B)}$	$\overline{(A+B)}\cdot C\cdot\overline{C}$
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	1	0	1	1	0	0
0	1	1	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	1	0	0
1	1	0	1	1	0	0

Figure 27: a truth table for $\overline{A+B}\cdot C\cdot\overline{C}$

gui.py

This module contains 2 helper classes, an input box and a text button. Apart from dragging, this will be the main way to interact with the user. This module is separated from the interfaces module in case other types of GUIs need to be defined.

This module is used by other modules if they require simple GUIs. Complex GUIs like asking the user for a directory can be achieved using tkinter (a python standard library).

Classes

An input box can be typed into, and validates the allowed characters to renderable ones. It does not support multiple lines or character repeating (if you hold down a key, windows has a short delay before it repeats the character).

class InputBox (subclass of IContainer)		
text render_text align align str allowed_characters	Text from the text module An alignment for how the text is aligned in the input box A string of allowed characters for this module	
InputBox	str pos font	The constructor takes a string for the maximum amount of text that it can contain, the top left of the bounding box and the font.

The only notable piece of code is that when the text is changed, it will realign itself to the alignment. This is in case the character entered takes up more height than previous ones.

Text buttons, unlike windows buttons activates on mouse down instead of mouse up. This can be changed very easily, but as selecting an object occurs on mouse down I have decided to keep it this way for consistency.

class TextButton (subclass of IContainer)		
text render_text func function	Text from the text module The function that is called when the button is clicked	
TextButton	text pos func font	The constructor takes the text for the text button, the top left of the bounding box, the function that is called when the button is pressed and the font.

Then the button is pressed, the function will be called with the following arguments: The instance of the text button that was pressed, an array of all IRenderable objects that were given to the text button, an array of keys that are currently pressed, an array of events that were given to the text button.

logic_circuit_gui.py

This module depends on the logic circuit module. It creates a GUI for logic circuits which allows for them to be created easily. Symbols have been taken from the IEEE Standard Symbols and exported to image files. They are then loaded by the module when it is started. This means that functions are restricted to only those with images.

The classes are very similar to the ones in the logic circuit module, except that each module contains the position for their output and their inputs. If their output is clicked, then they will spawn a wire. Wires will snap to outputs of logic gates.

A notable algorithm is how the wires are rendered. A Bezier curve is drawn between the input and output point. The algorithm to create the point defining the Bezier curve in pseudo code is:

$$distance = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$
$$niceness_{factor} = distance^{0.6} \times \ln(distance + 1)$$
$$points = [(x1, y1), (x1 + niceness\_factor, y1), (x2 - niceness\_factor, y2), (x2, y2)]$$

The python code is very similar so I have not included it, but it can be found in the system maintenance section.

As only logic gates with 2 inputs can be created, I have created function which creates renderable components from a given expression. Unlike the function in the logic circuits module, this function throws an error if any part of the Boolean expression contains more than 2 arguments. The classes are very similar to their pure logic counterparts, and the additional code is trivial therefore I have not included it here.

logic_gates.py

This module is split into 2 sections: loading, running. The code for this module can be easily shown with a flowchart, figure 28. The loading will only take a few seconds therefore there is no need to create a loading screen. On the other hand, closing will also take a few seconds, for this reason the program will not run in full screen mode as this will require the user to sit and wait a few seconds while the program closes.

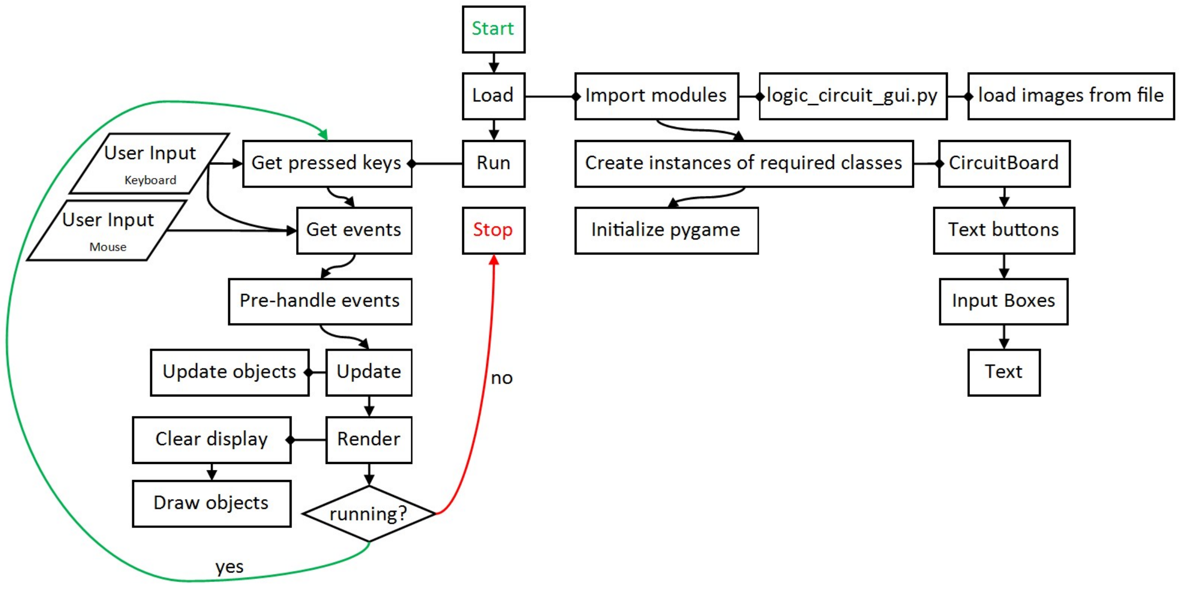


Figure 28: flowchart for the program

If the load time becomes too long, then a preload section should be added before the load section. This section will initialize pygame, create the display and print loading onto the screen before attempting to load any other modules.

If the main loop does not run fast enough and does not feel responsive enough, then either creating separate threads for rendering and updates or creating a fixed time render loop as shown in figure 29.

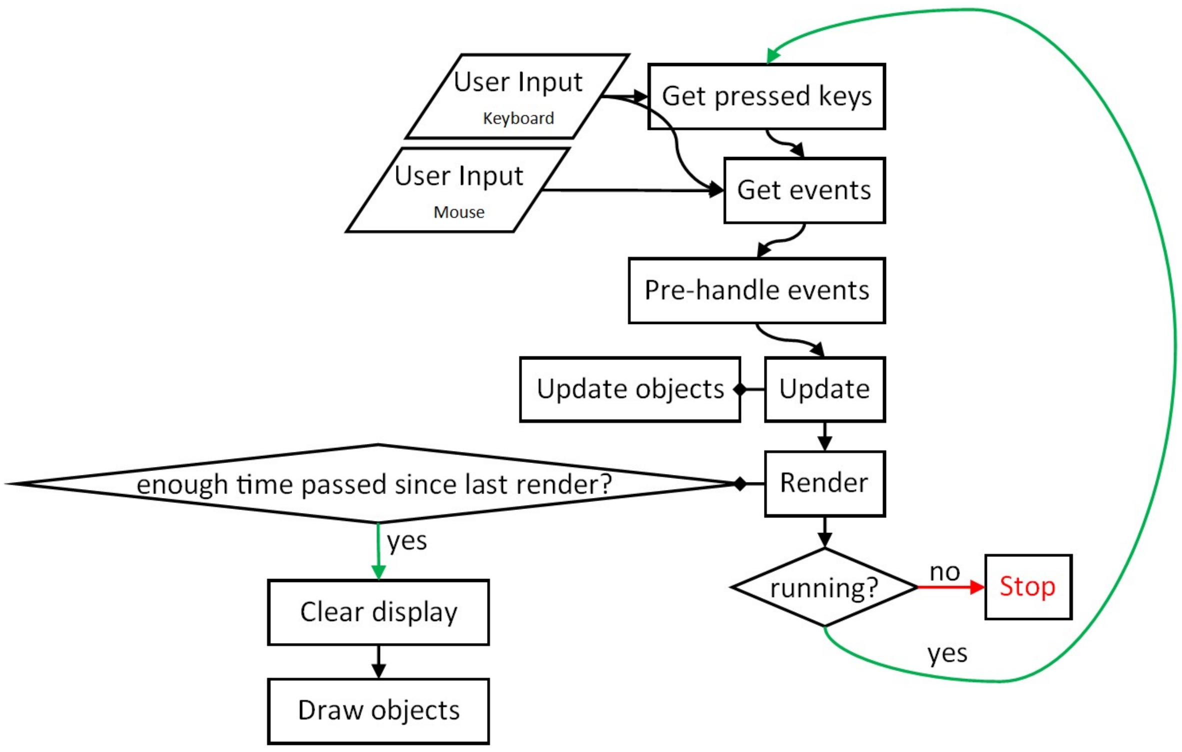


Figure 29: fixed time rendering for the program loop

Distribution

As mentioned in the modules overview, I will be using py2exe to create an executable for the user to run. The reason behind this is that although it is very easy to install python, it not as easy to install pygame. Leaving the user to do this will mean that the user will have to select the correct version of python, and the corresponding version of pygame. Installing pygame requires either compiling the program (which requires further installation of libraries) or installing a wheel file from the command line.

To avoid all of this, I decided to use py2exe to bundle the program into an executable, which is both familiar to users and easier to distribute. Unfortunately as I am loading image for the logic gates, I will not be able to create a single executable file. Instead I will have to create a zip file and distribute that via email, a memory stick or upload it to a site.

The file structure of the distribution will be:

```
Logic Gates
├───images
│   ├───and.png
│   ├───bulb.png
│   ├───bulb_off.png
│   ├───bulb_on.png
│   ├───nand.png
│   ├───nor.png
│   ├───not.png
│   ├───or.png
│   ├───repeater.png
│   ├───switch.png
│   ├───switch_off.png
│   └───switch_on.png
│
```

```
├───logic_gates.exe
└───saves
    ├───saved_image_name.png
    └───save_name.lgs
```

Figure 30: file structure for the distribution

GUI Design

Most of the modules that are render modules define how the GUI looks. As pygame does not include any GUIs itself, it is very difficult to create complex or good looking ones.

There are only 3 types of interactions that the current program uses – clicking, dragging and typing. The interfaces module defines selecting and dragging objects. The GUI module adds buttons and an input box and finally the logic circuit GUI module defines a draggable logic gates. All of the types of interactable objects have very simplistic graphics.

Keeping this in mind, the design for the interfaces should also be very simplistic.

HCI Rationale

Affordance

As all interface types are very simplistic, there will be a set colour scheme for different types of interactable objects.
Objects that cannot be interacted with will have a black or no border. This includes text and expressions. Black borders will be used to separate different blocks of text if they are too close to each other.

The only objects that can be dragged are logic gates. As they already have a district shape they will have no border. If there is ever the need of an object that does not have its own image but needs to be dragged, the purple border is reserved for it.

Visibility

The affordance section also covers the user of colour to help make it clear how the user can interact with different objects. Apart from this, all buttons contain text which tells the user what the button does.

Feedback

Objects that can be clicked on will have dark blue border. Upon selection their border will become light blue, until they are deselected. This visual indication lets the user acknowledge that the button has been pressed.

Consistency

Unfortunately there is no easy way to create GUIs in pygame that look consistent with the buttons provided by the operating system. A possible option could have been to use the pygame display as a canvas and use an external library like Qt or tkinter for the menus. This is not without its problems. By having 2 menus, it can be confusing for the user to switch between them; the code would have to be more complex as it has to manage the communication between the two windows and on smaller monitors both interfaces may not fit onto the screen at the same time.

GUI sections

The program is split into 3 GUI sections, one for logic circuit simulation, another for parsing expression, simplifying them and creating truth tables and the last one is text some basic help text and the overview of Boolean logic and logic gates.

Every section will have the same basic layout, as shown in figure 31. As these buttons are in a consistent place, it helps make the interface familiar. The interface is also somewhat similar to that of Microsoft Office 2007 (or newer) where there is a ribbon at the top.

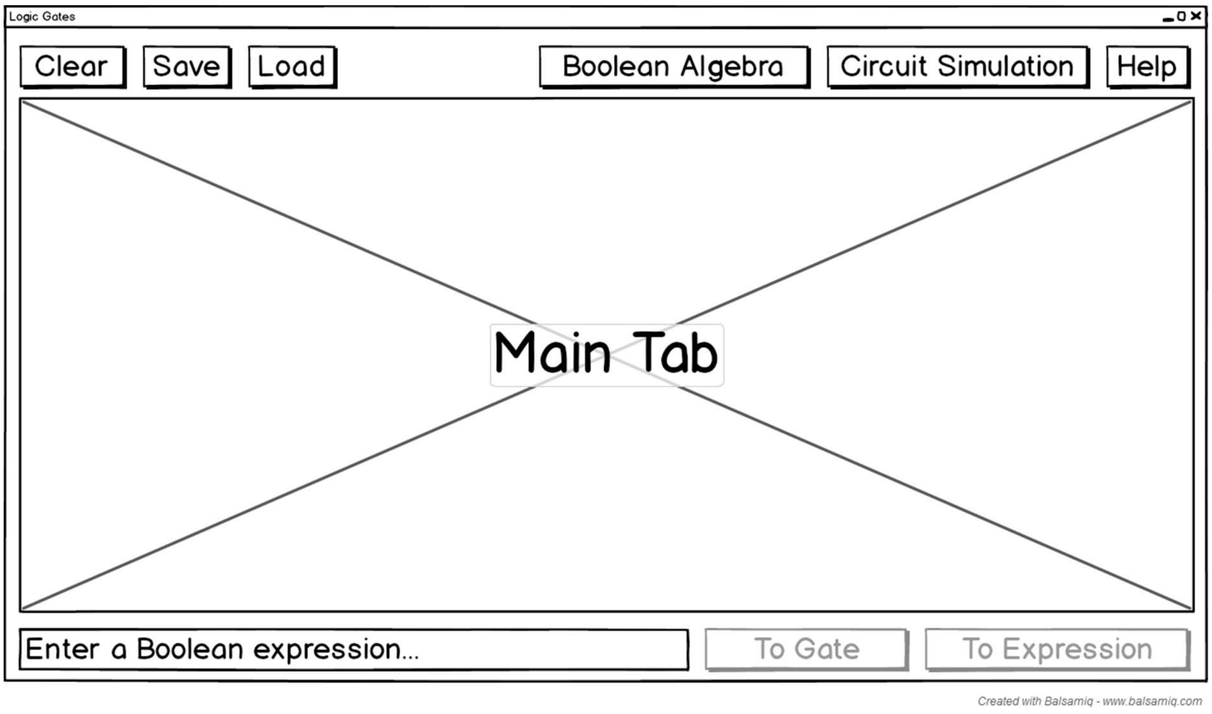


Figure 31: The basic layout for the program, showing the consistent navigation buttons

This suits well with the design of the program; I can use a container for the main tab and switch between them by making it hidden. Unfortunately I do not any way to disable buttons and still render them like I have done in this mock-up but this is trivial and could be easily implemented at a later stage.

When the user starts the program, they will be greeted with the circuit simulation tab. They can use navigation buttons on the top right to enter the different tabs. The main tab will contain the buttons for creating a Boolean circuit. The bottom ribbon allows the user to enter a Boolean expression to convert to a logic gate or they can press the convert to button to attempt to convert their logic circuit into a Boolean expression.

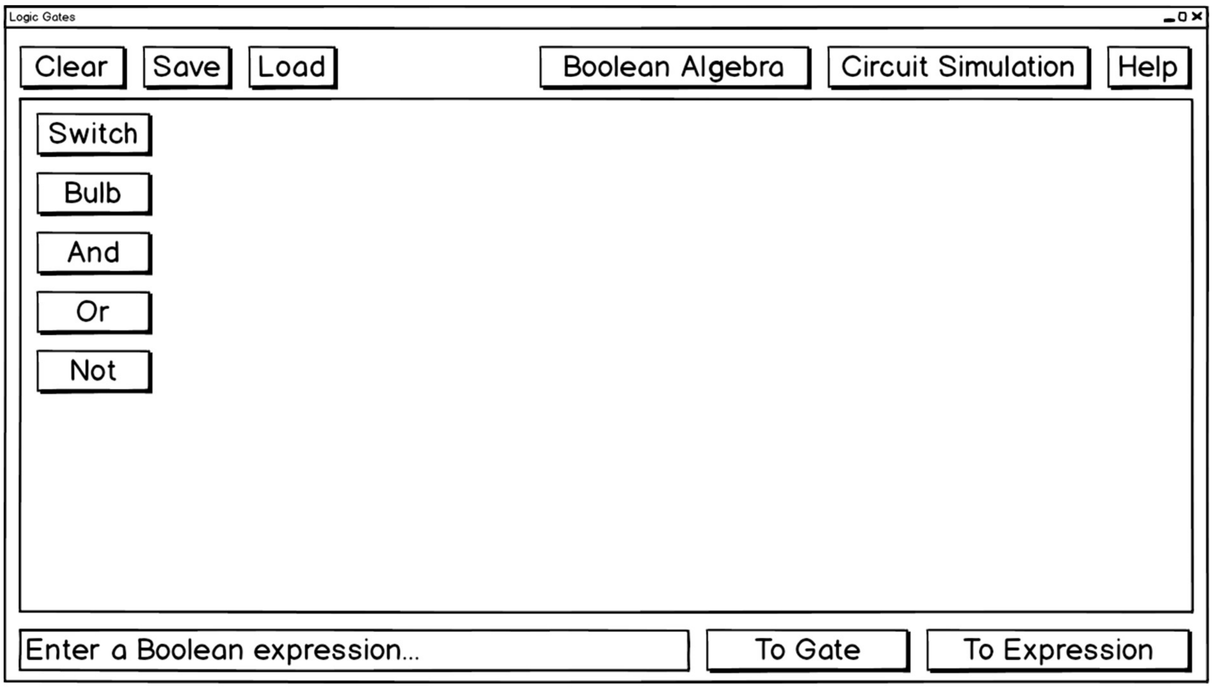


Figure 32: The window the user will be greeted with upon opening the program

Clicking the "To Expression" button will change the tab into the Boolean algebra tab, where they will be able to simplify the logic gate. If there is the need to notify the user of an error, a standard error message will be used instead of the program crashing. In this section, the clear button will remove all of the logic gates from the screen, the save button will save the current logic circuit and the load button will load the saved logic circuit.

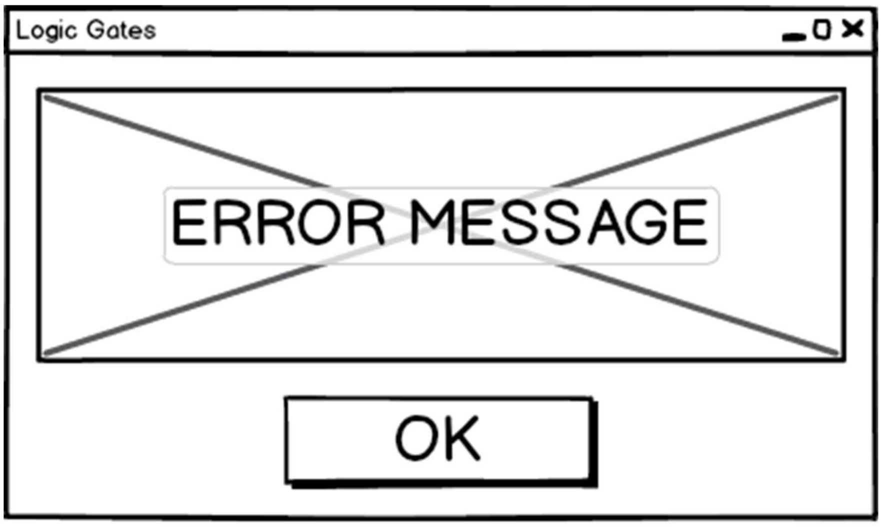


Figure 33: The standard error message

The other tab, the Boolean algebra section will allow the user to create a truth table or simplify a Boolean expression. In this section the clear button will clear the truth table and expression entered, the save button will save the truth table as an image and the load button is disabled. This is because there is no easy way to recover the expression entered from an image of the truth table.

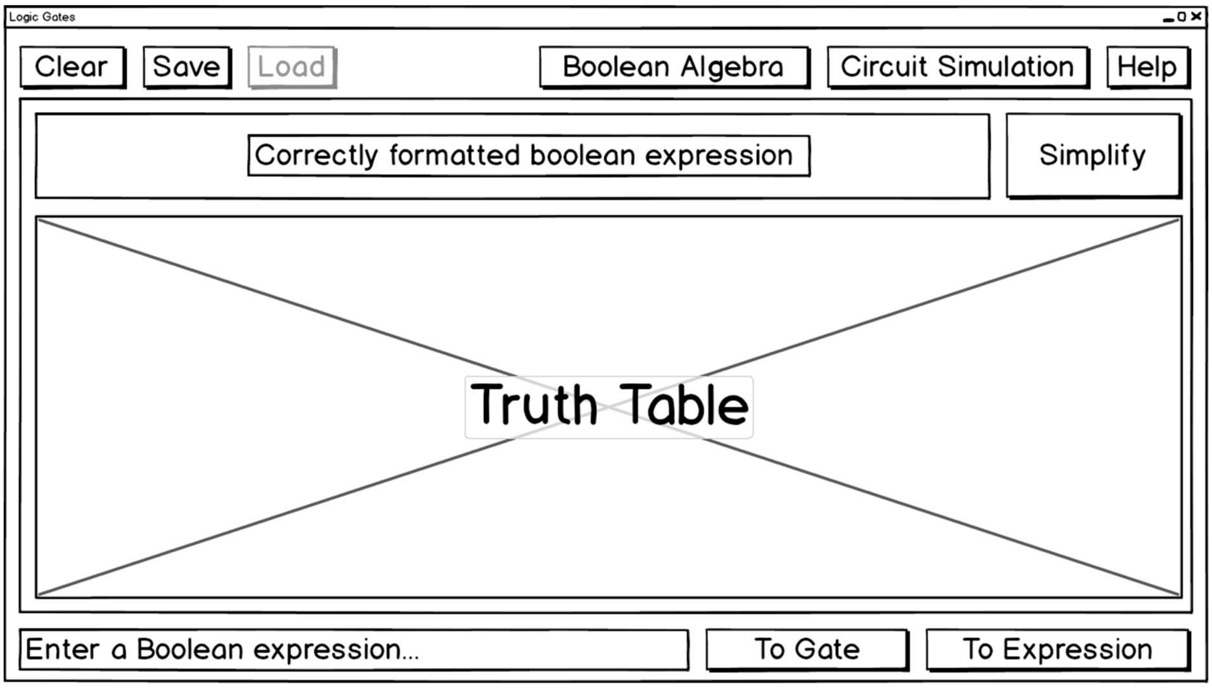


Figure 34: The Boolean algebra tab

The user still uses the bottom bar to input their Boolean expression, except this is copied into the main tab and formatted with the over bars. When they press the simplify button, the simplified expression will also be copied into the input box at the bottom so they can easily convert it into a gate.

Test Plan

Each module will have its own unit test, a form of white box testing. This is a piece of code that checks every part of the module by setting up a scenario and then inputting data. This will include both valid and invalid data. Unit testing ensures that the code works correctly as a single unit, when disassembled. This will help when attempting to refactor functions, as if the refactored code does not pass the unit test then the programmer knows he has make a mistake.

For some modules (especially those which render things), it will be easier to check the display by hand. For these modules, the test code will create a series of image that I have to look through, and verify that they are correct. For these tests, I will first create a description or rough drawing of the intended result to ensure that the module works are intended.

I will also need tests for the program when it is linked together by the logic_gates.py module. As this is more complex and requires testing multiple areas, it is best to manually use the program to test that the features work. This could include black box testing by a beta tester but should mainly alpha testing by the developer.

It is always possible that a bug will occur in the program even after extensive testing; therefore I will leave an email in the user manual for those who need assistance or find a bug.

Testing

As planned in the design section, I have written unit tests for most of the modules. The only two modules that I have not written unit tests for are logic_circuit_gui.py and logic_gates.py, which both rely on a high amount of user interaction.

Test Plan

Before I began testing I made a list for each module of the things that need to be tested.

boolean.py

Parse

- Correctly parses a given expression
- Raises an error if the expression contains uneven amount of brackets
- Raises an error if the expression contains two functions next to each other

Truth Table

- Raises an error if the expression is not of the correct type
- Correctly creates a truth table
- All possible combinations are created

Simplification

- Associativity
- Annihilation
- Impotence
- Identity
- Complementation
- Elimination
- Absorption
- Commutativity
- De Morgan's Law

logic_circuit.py

Correct simulation

- Non recursive circuit

- Recursive circuit

Converting to an expression

- Converts correctly without unnecessary extra symbols
- Converts symbols into readable ones
- Raises an error if given a recursive circuit

Creating a circuit board from an expression

- Raises an error if the expression is not of the correct type
- Creates a correct circuit board with inputs and wires

interfaces.py

Alignment

- Align to position
- Align to rectangle

Saving

- Saving an IRenderable
- Loading an IRenderable
- Saving an IContainer
- Loading an IContainer

Interaction

- Can Interact
- Only one object is interacted with
- Dragging
- Interacting with things inside of a container

text.py

Text

- Text renders correctly
- Newline actually creates a new line

Expression

- Expressions render with over bars for NOT
- Bounding box includes over bar

Table

- Row and column size fits every expression in the table

gui.py

Input Box

- All normal characters can be entered
- Does not allow unrenderable characters like tab
- Text can be deleted
- Text aligns to centre left

Text Button

- Calls function when pressed

logic_circuit_gui.py

- Raises error if files are not found

Gates

- Gates render their output colour

Wires

- Wires render their output colour
- Wires connect from gate outputs
- Wires connect into gate inputs
- Wires can exist with their output end unconnected but not their input

Circuit board

- Contains the relevant buttons to create gates
- All objects can be moved by dragging the background

Renderable components

- Renderable components can be created from an expression

logic_gates.py

- Alpha and Beta testing
- Fixing any bugs reports

Unit Test Code

test_boolean.py

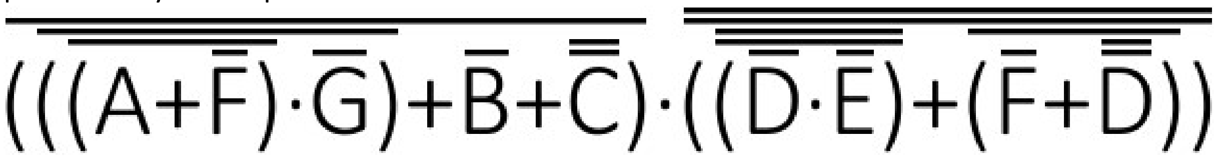
source in separate file

test_interfaces.py

source in separate file

text.py

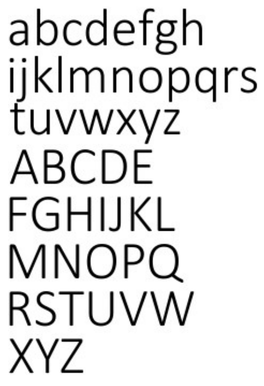
This module required me to view some image that were created by the code. The image produced by text.Expression is:



$$\overline{\overline{\overline{(A+\overline{\overline{F}})}\cdot\overline{G}}+\overline{B}+\overline{\overline{C}}}\cdot\overline{\overline{\overline{\overline{(\overline{\overline{D}}\cdot\overline{E})}}+\overline{(\overline{F}+\overline{\overline{D}})}}}$$

Figure 35: An image produced by test_text.py

The image produced by text.Test is:



abcdefgh
ijklmnopqrs
tuvwxyz
ABCDE
FGHIJKL
MNOPQ
RSTUVW
XYZ

Figure 36: An image produced by test_text.py

And finally the image produced by text.TruthTable is:

Figure 37: An image produced by test_text.py

You may not be able to see this one clearly but it does indeed create the correct truth table with the correct ordering of row headings.

gui.py

source code in separate file

logic_circuit_gui.py

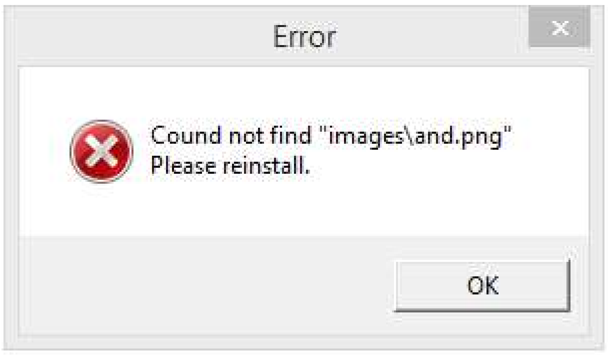
To test this module, I simply tested the circuit simulation tab in logic_gates.py.

logic_gates.py

For some of the planned testing point for logic_circuit_gui, I added some code to it to ensure that the program didn't crash when being ran. This meant that I had to update the testing plan for these two modules.

logic_circuit_gui.py

Instead of raising an error if files are not found, a helpful error message should be displayed to the user. Unfortunately there is no way except for the user to reinstall to get the files required.


Error

Cound not find "images\and.png"
Please reinstall.

Figure 38: Test Success

Renderable components can connect to others and will change colour depending on their output. Wires cannot be connected to wires and can only connect outputs of components to inputs of others.

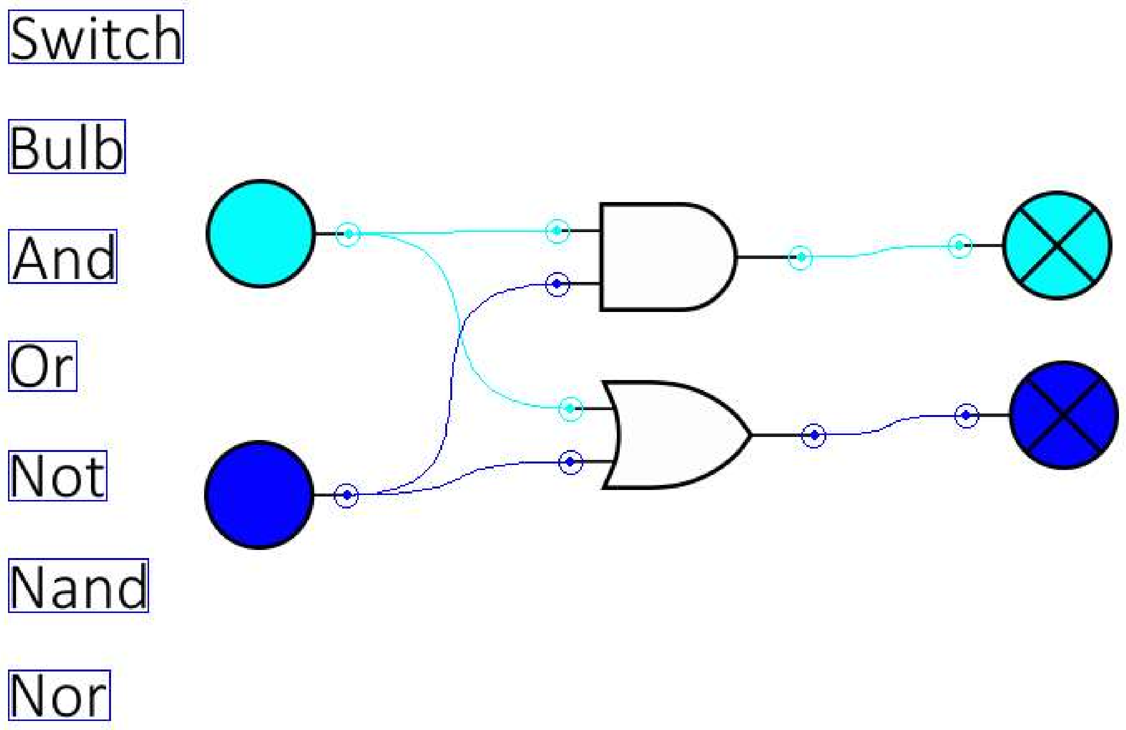


Figure 39: Works as intended

There is no real way to show this but by dragging the background, all objects can be moved around.

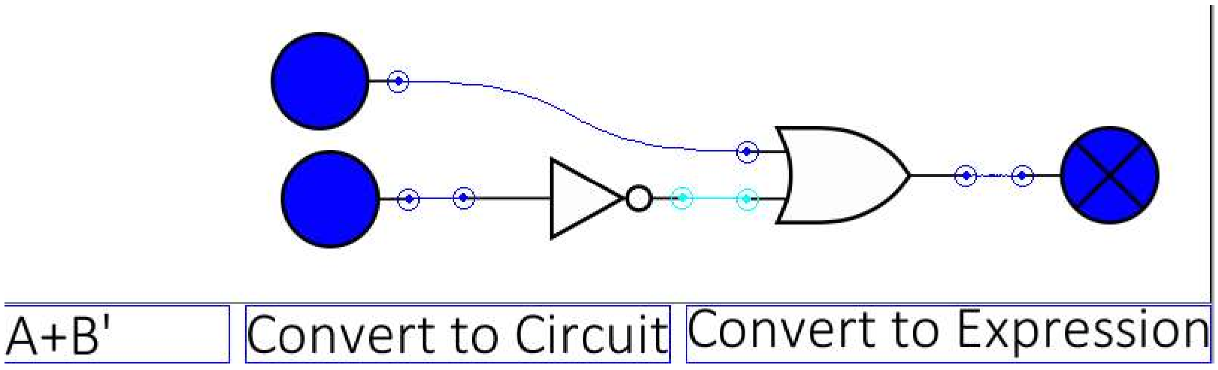
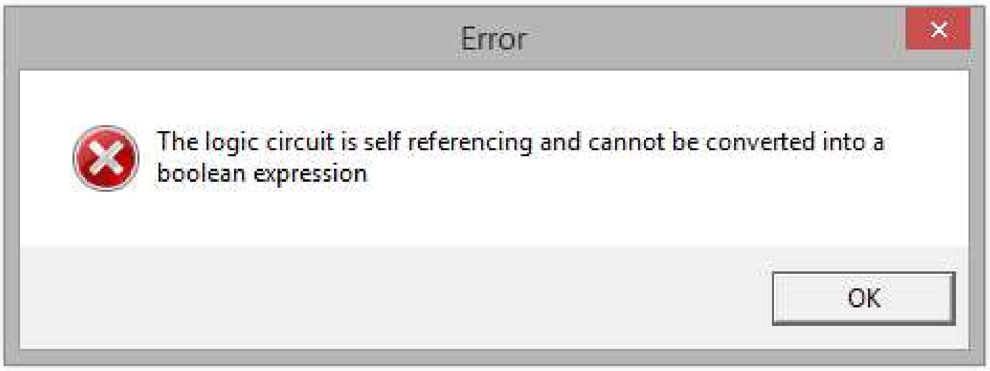


Figure 39: The logic circuit was turned into the expression on the left

Converting a logic circuit to an expression also works, but more importantly, when trying to convert a circuit that has no expression (recursive circuits), the program gives a nice error message instead of crashing.



The logic circuit is self referencing and cannot be converted into a boolean expression

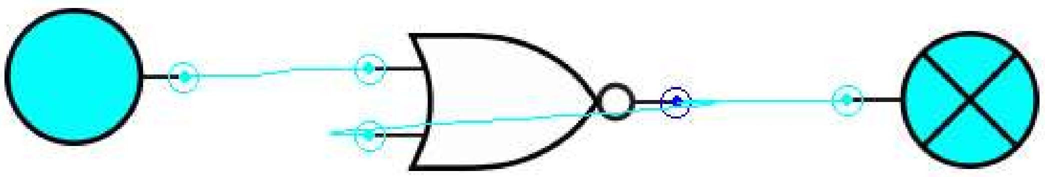


Figure 40: An error message is displayed when trying to convert this circuit

Unit Test Outputs

Here is the output the unit test for Boolean. All of the other unit test outputs look like this. I have not included them but if you wish you can run the code yourself.

```
----------------------------------------------------------------------
Ran 52 tests in 0.246s

OK
```

Performance

The program has been tested on my Windows 8 machine, 2 different school computers both running Windows 7 (different screen sizes), an old laptop running Windows XP (and only 512MB or ram) and an Ubuntu Laptop.

To get a grasp of how well they were running, I uncapped the FPS and calculated the average FPS while idle, constantly updating (I connected the output of a NOT gate to itself, and then connected this to a bunch of different gates) and while dragging the table containing the expression A+B*C+C (This is the least optimised part of the code as on every update the Boolean Algebra tab actually recalculates the position of the expression and button.

All FPS is rounded to the nearest FPS and taken over a period of 30 seconds with no other programs running.

	Windows 8	Windows 7	Windows 7	Windows XP	Ubuntu
	My Machine	School Computer	School Computer	Old Laptop	Old Laptop
	1920x1080	1280x720	1280x600	1280x720	1280x768
idle	538	212	228	27	37
constant update	198	57	64	9	16
optimised code	48	23	25	3	7

Figure god knows what: Raw data of FPS over 30 second period

Although the old XP laptop only ran at 9FPS when simulating a logic circuit and the response was very poor, the program was still usable.

One thing that was noticed is that if multiple instances of the program is started on the same computer, then the FPS for each running instance would drop by roughly the number of instances running (so 3 instances = 1/3 of the fps). This is a very curious fact and further testing could be done on it.

System Maintenance

Setting up the development environment and Notes

1. Install Python 3.4+
2. Either visit http://www.lfd.uci.edu/~gohlke/pythonlibs/ to download precompiled binaries (windows only) or visit the respective websites to download and install py2exe, pygame and (optionally) Cython.
3. Exit Python34\Lib\tkiner_fix.py, adding or True to the 57th, 67th and 73rd line, all which contain if statements. This is documented in setup.py.
4. Download the source from https://hostr.co/aC8o0LbhlNQf and unzip it.

In the design section I covered the proposal for the file structure. While programming some changes have been made to it and new file structure is:

```
Logic Gates
│   alignment.py
│   boolean.py
│   gui.py
│   interfaces.py
│   logic_circuit.py
│   logic_circuit_gui.py
│   logic_gates.py
│   setup.py
│
├───csetup
│     csetup.py
│
├───images
│       and.png
│       bulb.png
│       bulb_off.png
│       bulb_on.png
│       favicon.ico
│       nand.png
│       nor.png
│       not.png
│       or.png
│       repeater.png
│       switch.png
│       switch_off.png
│       switch_on.png
│       vector_gates.svg
│
├───Logic Gates 64bit
│
├───Logic Gates 32bit
│
└───test
        test_boolean.py
        test_gui.py
        test_interfaces.py
        test_logic_circuit.py
        text_text.py
```

Figure 41: File structure for the development

When refactoring code, please ensure that the code passes all unit tests before committing the change. My current unit tests are at a minimum, and more extensive testing is required if a change is made. If an additional module is required, then this should be added to the top most directory. If required a unit test should be written and added to the test folder.

boolean.py

In the design I talked about using an open source boolean.py which covered most of the needs of the project. To cover the rest of them I had to edit the code. While doing so I also found some problems with it and submitted a pull request which the author merged.

To simplify using de Morgan's law, I editing the eval function in the NOT class: Instead of cancelling its terms when trying to evaluate itself, instances now literalizes its terms.

While trying to fix printing using dot and prime notation, I found that the program prints an unnecessary number of parentheses around some arguments. I fixed this and added an extra case for printing prime notation in the NOT class.

To parse other Boolean notations, I tweaked the parse function to accept "¬" and "!" for NOT, "·", "∧" and "^" for AND, "∨" (logical or, not capital v) for OR. Then I wrote a function that converts prime notation to tilde notation to allow it to be parsed by the function without rewriting it entirely. Pseudo code for this function is as follows:

```
$expr \leftarrow INPUT$
$DO$
 $tilde\_index \leftarrow expr.rfind(“’”)$
 $depth \leftarrow 0$
 $FOR\ I \leftarrow tilde\_index - 1\ TO\ -1\ STEP\ -1$
  $IF\ expr[i]\ ==\ “)”$
   $depth \leftarrow depth + 1$
  $ELSE\ IF\ expr[1]\ ==\ “(“$
   $depth \leftarrow depth - 1$
  $IF\ depth\ ==\ 0\ and\ expr[i] != “”$
   $expr = expr[up\ to\ i] + “\sim” + expr[i\ to\ tilde\_index] + expr[tilde\_index$
             $+ 1\ to\ end]$
   $BREAK$
 $IF\ depth != 0$
  $RAISE\ ERROR$
 $ELSE$
  $tilde\_index \leftarrow expr.rfind(“’”)$
$RETUR\quad expr$
```

To create a function that returns a truth table, I had to create a format for tables. I could have chosen to create a new class or use SQL but I settled on an array of dictionaries with the column name as the dictionary key each index of the array is a different row. Using python syntax, the code is like so:

```
[
 {<column_name>: <value>, <column_name>: <value>, ...},
 {<column_name>: <value>, <column_name>: <value>, ...},
 ...
]
```

As functions (which are expressions) can contain expressions are their arguments, the natural solution to this problem is to use recursion. The python code to create a truth table is as follows:

```
def truth_table(expr, format_str=False):
    """
    Returns a truth table from an expression, which may be a string or
Expression.

    The table is in the format:
    [
        {<column_name>:<value>, <column_name>:<value>, ...},
        {<column_name>:<value>, <column_name>:<value>, ...},
        ...
    ]
    Note that if format_str is True, then you can only sort them as
strings.
    """
    if isinstance(expr, str):
        expr = parse(expr, eval=False)
    if not isinstance(expr, Expression):
        raise TypeError(
            "Argument must be str or Expression but it is %s" %
expr.__class__)
    if isinstance(expr, BaseElement):
        return [{expr:expr}]

    def truth_table_permutations(expr, symbols, values, format_str=True):
        def replace(expr, symbols, values, format_str=True):
            sub = {symbols[i]: values[i]
                   for € in range(len(symbols)) if symbols[i] in
expr.symbols}
            if format_str:
                return {str(expr): str(expr.subs(sub))}
            else:
                return {expr: expr.subs(sub)}

        if isinstance(expr, Symbol):
            return replace(expr, symbols, values, format_str)
        return_dict = {}
        if isinstance(expr, Function):
            return_dict = replace(expr, symbols, values, format_str)
            for args in expr.args:
                return_dict.update(
                    truth_table_permutations(args, symbols, values,
format_str))
            return return_dict

    # Make the rows look slightly nicer
    symbols = sorted(expr.symbols, key=lambda e: str€)
    rows = []
    for values in itertools.product((FALSE, TRUE), repeat=len(symbols)):
        rows.append(
            truth_table_permutations(expr, symbols, values, format_str))
    return rows
```

Figure 19: python code to create a truth table

The idea behind it is:

Take an expression as an input.
Sort the symbols that are used in the expression into alphabetical order and save this into symbols.
Create an array called rows.
For every permutation of true and false possible for every symbol, add a new row to rows and put into it the result from calling the recursive function giving it the expression, the symbols and the current permutation.
Return rows.

The recursive function:

For every possible sub-expression in expr (this is the recursive bit), substitute the values of the permutations into the symbols, resulting in true or false. Return a dictionary containing the sub-expression and the evaluated value of this sub-expression with the current permutation.

Stand alone expression rendering

If for some reason the license for boolean.py does not match your requirements and you need to code your own, there is an alternative to rendering Boolean expressions that does not depend on a large library. In fact the module itself is only 285 lines without the demo code.

Although this depends on pygame and pygame.freetype, one could easily port to use another text library or even another programming language. The requirements are that the library supports getting the bounding box of text and has some functions for drawing onto a display.

Modules at a glance

All modules have been extensity commented and contain helpful doc code. This can be accessed by importing said module and using the help function on it. The design section covers the difficult function for each module.

User Manual

Installation

Windows

1. Download the 64bit version (https://hostr.co/6TvjGQt3hKP4) or the 32bit version (https://hostr.co/46v0DBLhDk1m). If you are unsure which version of windows you have, download the 32 bit version.
2. Unzip the folder into a directory you will remember. If you current unzipping program doesn't support .7z formats, then either install 7zip (http://www.7-zip.org/) or PeaZip (http://peazip.sourceforge.net/)
3. To start the program, open the Logic Gates folder and run double click "logic_gates.exe"

Place the "Logic Gates" folder somewhere where you can remember, preferably inside of your documents or downloads folder.

If you want to create a shortcut then you can right click on logic_gates.exe and select "Create Shortcut". You can move the shortcut anywhere you want as long as you don't change the location of the program.

Alternatively you can install python and pygame to your system and download the developer version of the program. The instructions are similar to those on Mac OS X.

Mac OS X

1. Download Python 3.4 (https://www.python.org/downloads/release/python-343/) and install it.
2. Download and install pygame 1.9.1 (http://www.pygame.org/download.shtml)
3. Download the developer version of Logic Gates (https://hostr.co/aC8o0LbhlNQf)
4. Unzip the folder into a directory you will remember. If you current unzipping program doesn't support .7z formats, then either install 7zip (http://www.7-zip.org/ under the unofficial mac builds)
5. Start the program. If it prompts you to select a default application, find and select PythonLauncher. Alternatively you can open your console any run the program by typing "pythonw logic_gates.py" when you are in the Logic Gates folder.

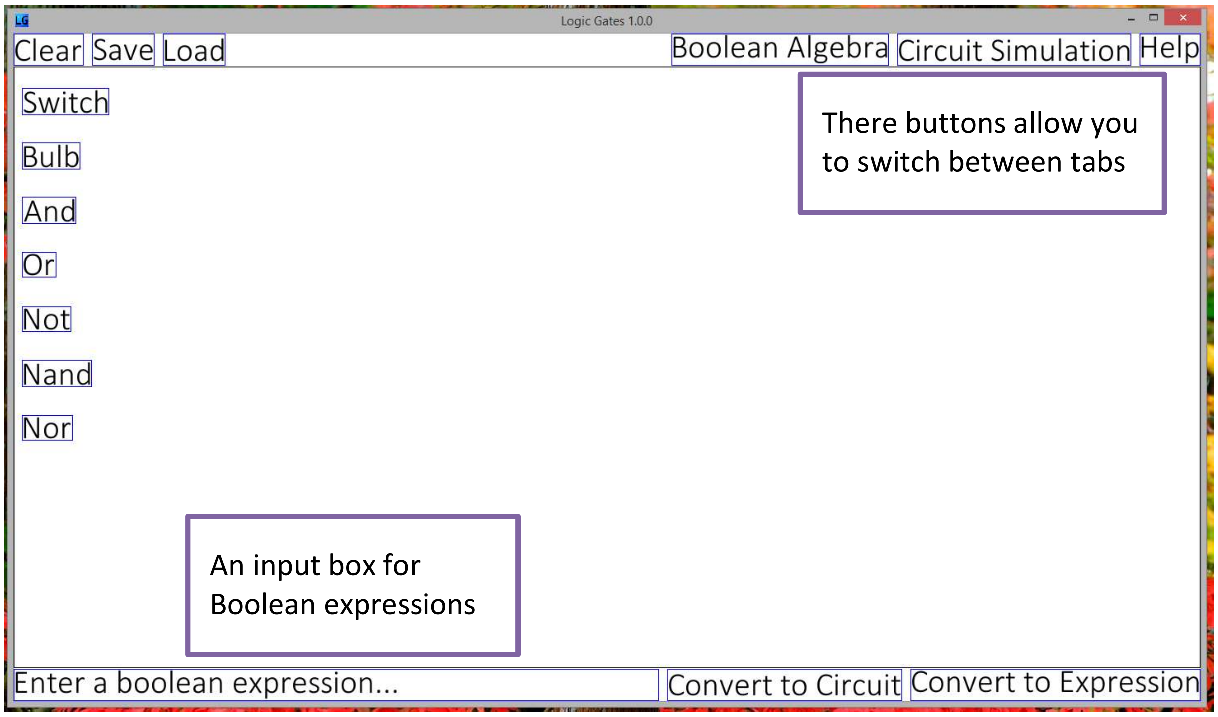
On Mac OS X, your window borders may be slightly different from the image shows. The buttons on the top border should be the same as on a mac but may be flipped.

Linux

1. Install python 3 if it is not already installed on your distribution. On Ubuntu you can do this by typing "sudo apt-get install python3"
2. Install pygame, you must ensure that the version of pygame you install is for python3 and not python2. If pygame for python3 is not in the packages of your distribution, then you can build it from source by following the instructions here: http://www.pygame.org/wiki/CompileUbuntu?parent=Compilation
3. Download the developer version of Logic Gates (https://hostr.co/aC8o0LbhlNQf)
4. Unzip the folder. If you current unzipping program doesn't support .7z formats, then you can install p7zip from your package manager.
5. Start the program by running "python logic_gates.py" or from your file explorer.

On Linux, your windows borders may look different from those in the images shown.

General



There buttons allow you to switch between tabs

An input box for Boolean expressions

When you start the program, it should start at a size slightly smaller than your screen resolution. If you find that the buttons and bundled together or the text is too big, then please send a bug report (this is explained in the bug reports tab).

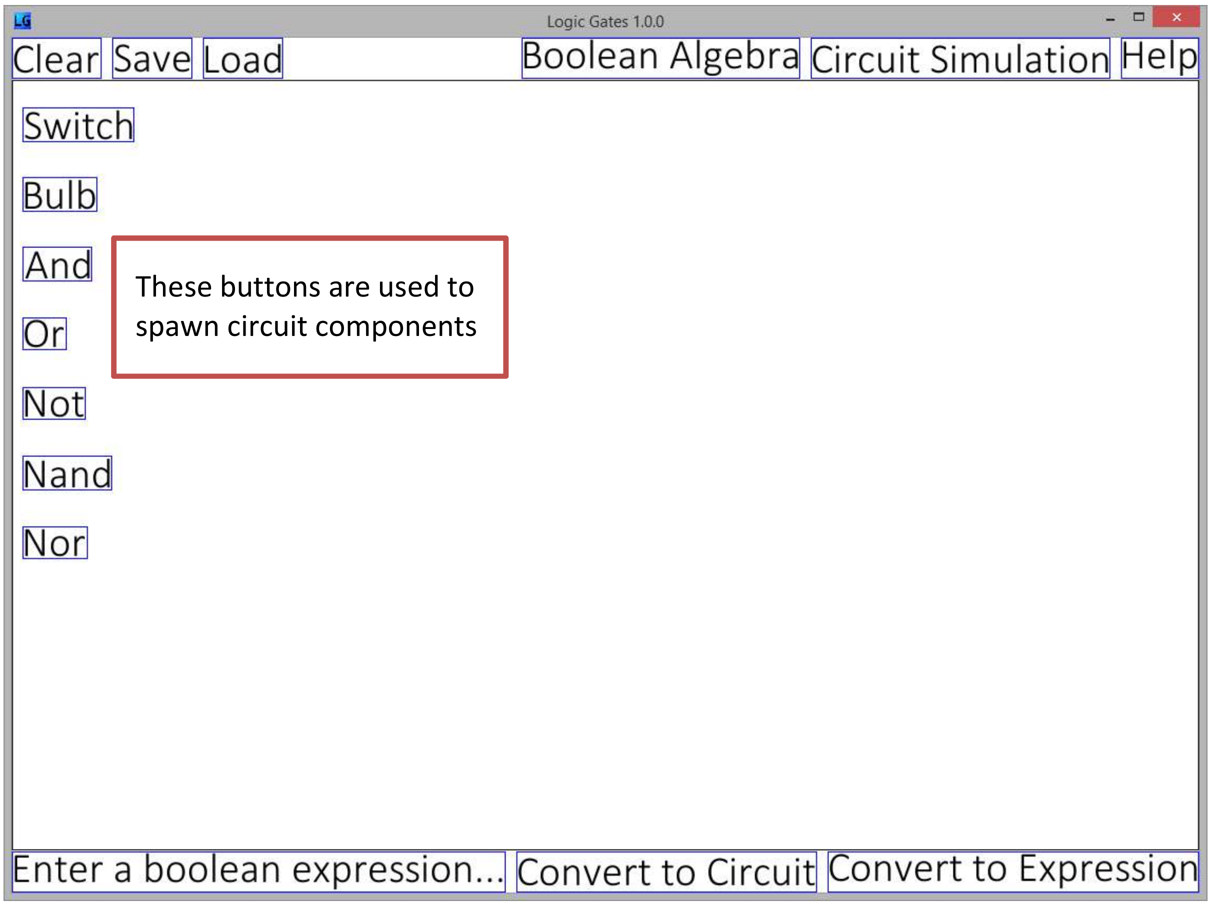
You can resize the program however you want but you may find that if the overall width of the program window is too small, not all of the text may fit inside. If so, then just make the window larger.

If you receive an error message, then please read it and read the Errors section before submitting a bug report. If the program closes for no apparent reason without an error, then please submit a bug report. This is explained below in the Reporting Bugs section.

Tabs

This program has 3 main tabs, “Boolean Algebra”, “Circuit Simulation” and “Help”. You can navigate between them by pressing the text on the top right.

Circuit Simulation



When you start the program, you should be greeted with this tab. In this tab, you can create logic circuits. On the right hand side, you should see some buttons that are labelled with words. By left clicking the button you can spawn the described circuit component.

Circuit components can be dragged by left clicking (and holding) on the component and moving the mouse. At the right of every component except the bulb, there are small filled circle which is inside of a hollow one. By clicking inside of the hollow circle, you can create a wire.

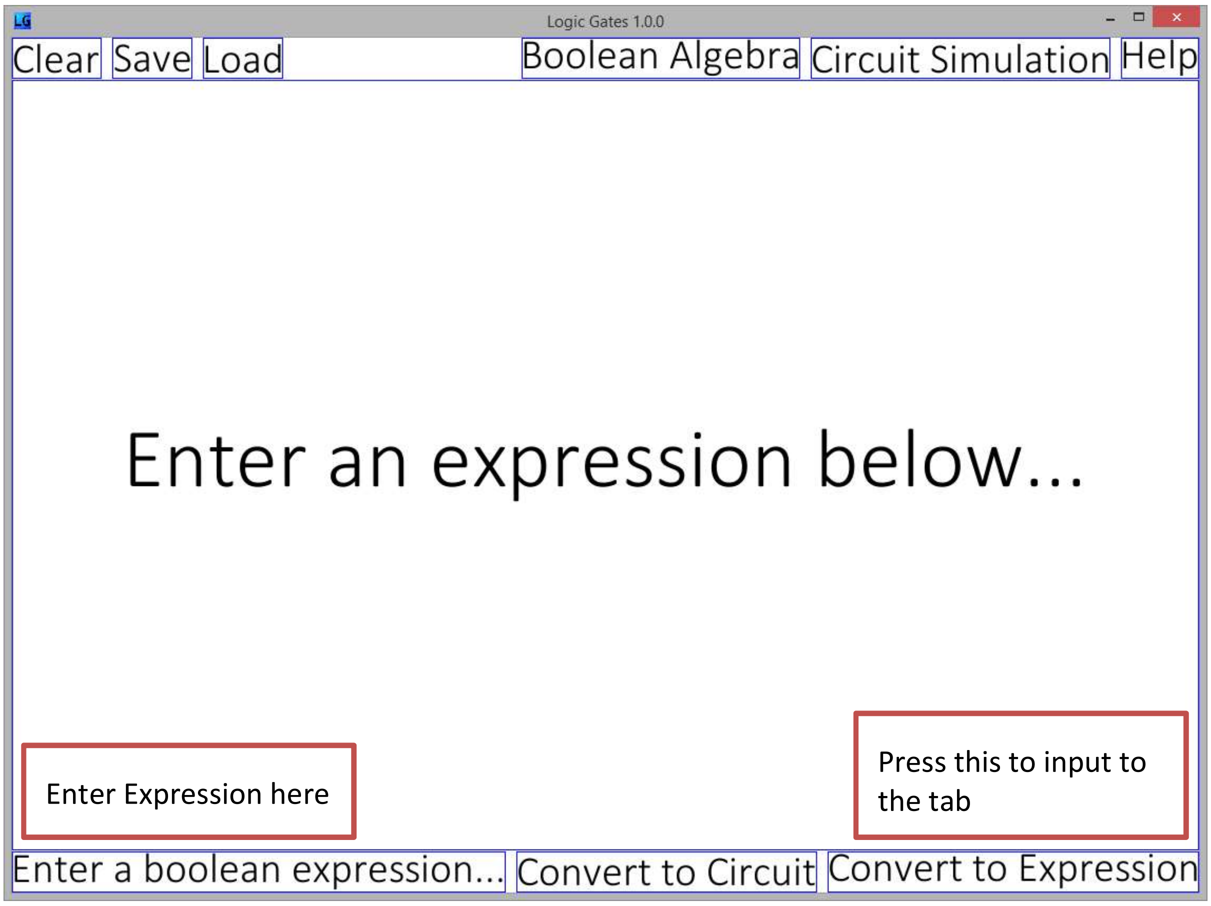
Wires can be dragged by left clicking their output. If when you release the wire it is close enough to the input of another component, it will automatically connect. If you want to remove a wire from a component, simply click inside of the wire output and drag the wire away.

You can right click all components to delete them, and middle click to copy them. Copies of wires will be connected to the same output as the wire you are copying from while copying a gate will make a brand new gate that is not connected to anything else.

If you begin dragging the background, then this will move all of the components.

In this mode, the convert to expression button will attempt to create a Boolean expression from your logic circuit starting at the last bulb that you selected. This button will sometimes show an error message if something went wrong.

Boolean algebra



The Boolean algebra by default tells you to ender an expression below. The buttons below are part of the input bar. The left most box labelled "Enter an expression..." is a text box. You can select a text box by clicking inside of the button. When the text box it selected, you can press keys to enter letters or symbols.

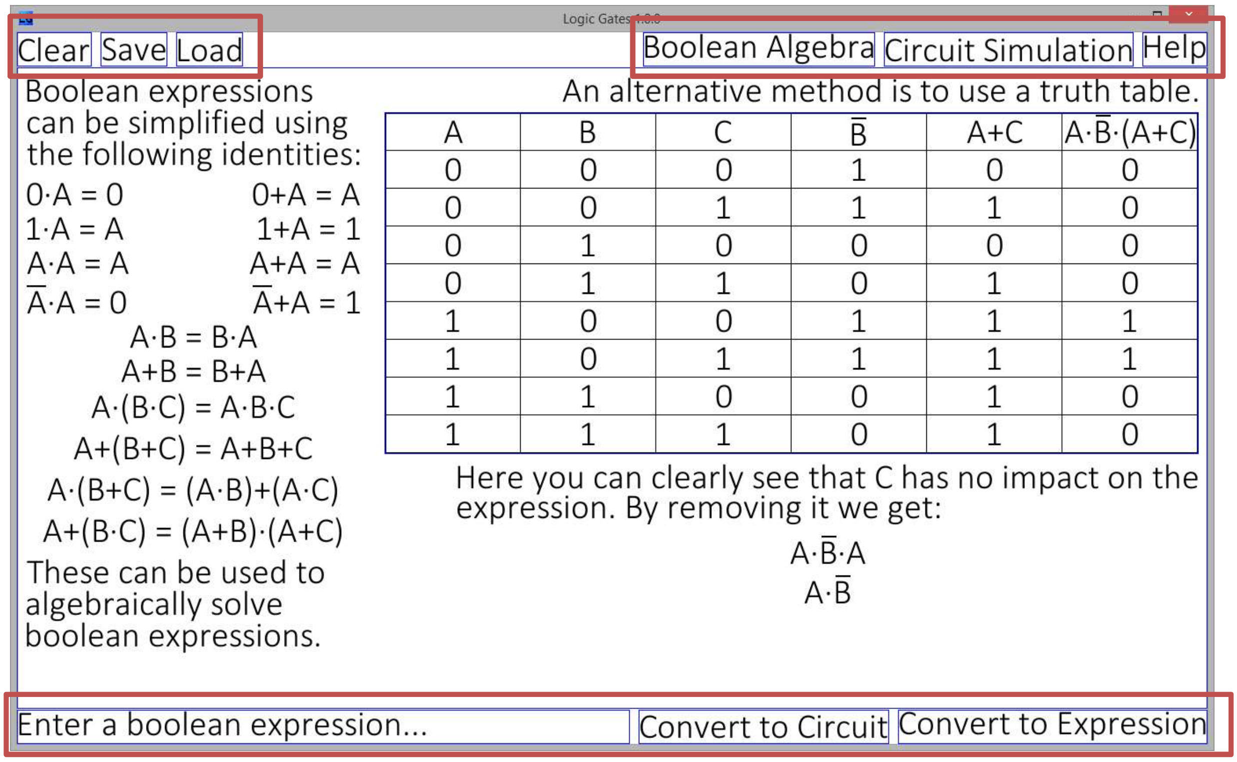
After you have entered a Boolean expression, you can click the "Convert to Expression" button to take the expression you entered and put it into the Boolean algebra tab.

When there is an expression in this tab, then at the top it will display the expression rendered with over bars instead of NOTs. On the right of the tab is the simplify button. This button attempts to simplify the Boolean expression. At the bottom is a truth table for the expression. If the truth table is too large or too small, you can use your mouse wheel to zoom in or out. You can also use your mouse to drag the table around.

Help

The last tab is the help tab. This tab includes a list of Boolean identities and an example of simplifying using a truth table.

Buttons and Inputs



Boolean expressions can be simplified using the following identities:

$0 \cdot A = 0$ $\quad 0+A = A$

$1 \cdot A = A$ $\quad 1+A = 1$

$A \cdot A = A$ $\quad A+A = A$

$\overline{A} \cdot A = 0$ $\quad \overline{A}+A = 1$

$A \cdot B = B \cdot A$

$A+B = B+A$

$A \cdot (B \cdot C) = A \cdot B \cdot C$

$A+(B+C) = A+B+C$

$A \cdot (B+C) = (A \cdot B)+(A \cdot C)$

$A+(B \cdot C) = (A+B) \cdot (A+C)$

These can be used to algebraically solve boolean expressions.

An alternative method is to use a truth table.

A	B	C	$\overline{B}$	A+C	$A \cdot \overline{B} \cdot (A+C)$
0	0	0	1	0	0
0	0	1	1	1	0
0	1	0	0	0	0
0	1	1	0	1	0
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	0	1	0
1	1	1	0	1	0

Here you can clearly see that C has no impact on the expression. By removing it we get:

$A \cdot \overline{B} \cdot A$

$A \cdot \overline{B}$

Things that can be interacted with are highlighted by the red box.

Clear

The clear button on the top left removes all logic gates if you are in the circuit simulation tab or removes the current expression if you are in the Boolean algebra tab.

Save

In the Boolean algebra tab, the save button saves the expression as a ".les" (Logic Expression Save). It will also save the Boolean expression (with the correct overbars) and truth table as png files.

In the Circuit Simulation tab, the save button attempts to save your circuit simulation to a ".lgs" (Logic Gate Save). Currently this feature is in alpha so may not work with all logic circuits.

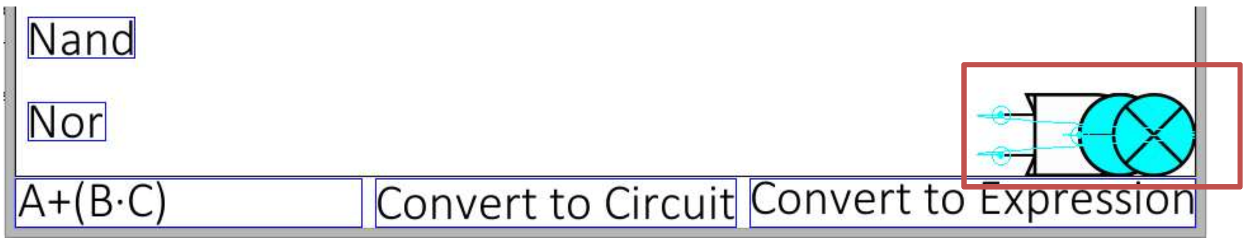
Load

Loading does the exact opposite of saving. You are asked to get a file to load either an expression or logic circuit into the program.

Boolean algebra, Circuit Simulation and Help

These buttons allow you to change between the different tabs.

Convert To Circuit



This button converts any expression entered in the input box into a series of logic gates. Currently all logic gates are dumped to the bottom right of the tab. This is because this feature is in alpha. You will have to drag them out manually for now.

Convert to Expression

When in the Boolean algebra tab, this simplify sets the expression for the Boolean algebra tab to the same expression as you have entered into the input box.

When in the circuit simulation tab, this button will attempt to create a Boolean expression from your logic circuit, starting at the last bulb you clicked. An error message will appear if the circuit you are trying to convert has no Boolean algebra equivalent.

Enter a Boolean expression…

This is an input box which you can enter text into. This text is parsed into a Boolean expression when you click Convert to Expression or Convert to Circuit.

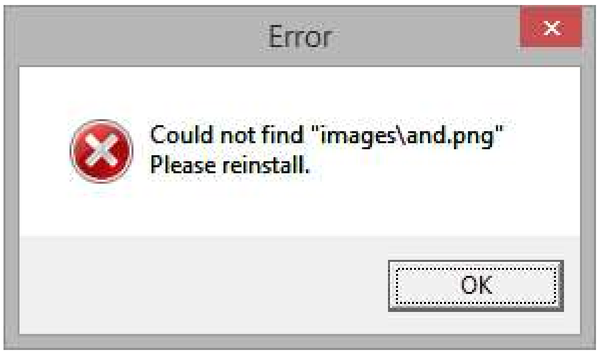
Left clicking on the input box selects it and allows you to enter text into it. Clicking away deselects the input box.

You can copy from the input box by either middle clicking on the input box or using CTRL+C. You can erase the text in the input box by right clicking on it or pressing the DELETE key when the input box is selected. Pressing the ENTER key while the text box is selected also deselects the text box.

Errors

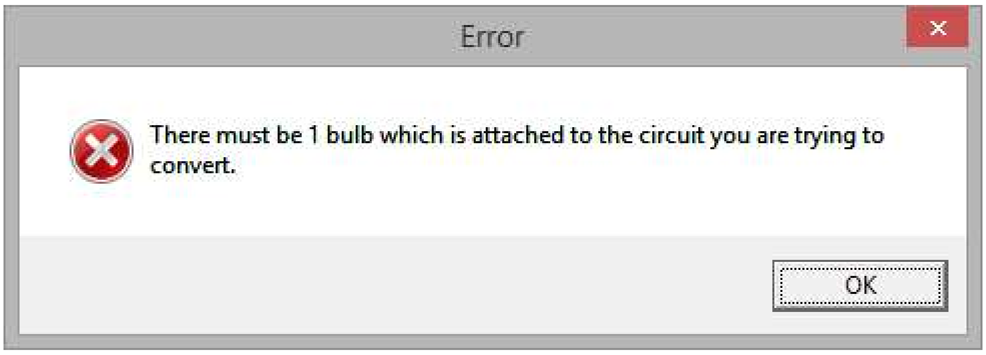
The following errors may popup when you are using the program. Find your error and read the help text below for more information.

The program may crash without an error message. If this occurs, please refer to the bug reporting section. If you get an error that does not appear on this list, please also report it as a bug.


Error

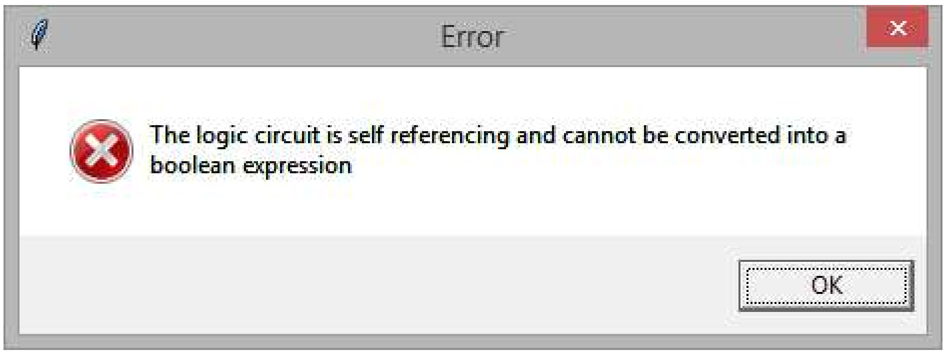
Could not find "images\and.png"
Please reinstall.

The error can occur at start-up. If you get this error, the best choice is to re-download the file. If you have moved the images folder or renamed items inside of it, this can occur.


Error

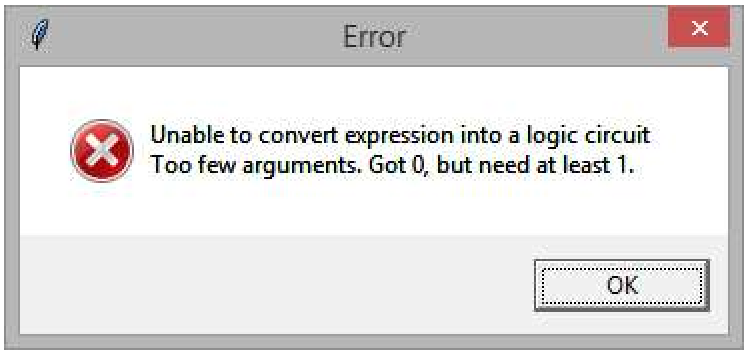
There must be 1 bulb which is attached to the circuit you are trying to convert.

This error occurs when you click the “Convert to Expression” button when in the Circuit Simulation tab but have no bulbs on the screen. Drab a bulb into the screen and connect it to your circuit.


Error

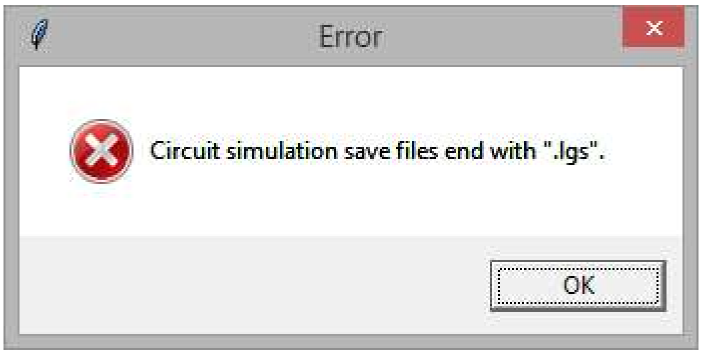
The logic circuit is self referencing and cannot be converted into a boolean expression

This error occurs when you click the “Convert to Expression” button when in the Circuit Simulation tab but your logic circuit is has a self-referencing section, for example if you connect the output of a gate to the input of the same gate. These kinds of logic circuits cannot be converted into expression.


Error

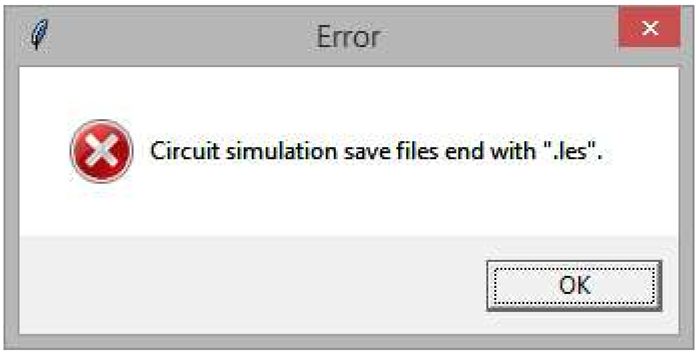
Unable to convert expression into a logic circuit
Too few arguments. Got 0, but need at least 1.

If you receive this error, or any error that begins with “Unable to convert expression into logic circuit”, then you have entered an incorrect expression. The text below is sometimes helpful, for example to get this error message I entered “A*~”. The help message is indicating to the fact that there must be a symbol after the ~ but it has not found any.


Error

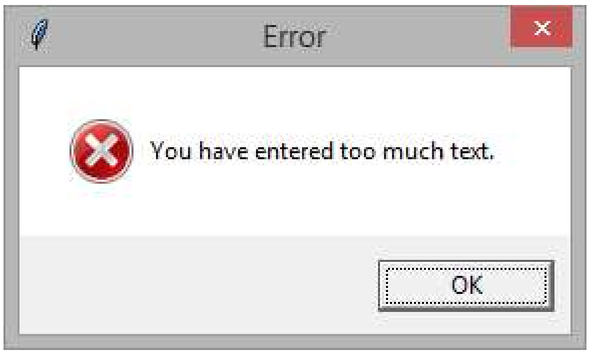
Circuit simulation save files end with ".lgs".

This occurs when you are trying to load a save from the Circuit Simulation window you have selected a file that does not end in “.lgs”. Currently only “.lgs” files can be loaded therefore you must select a “.lgs” file. “.lgs” files are created when saving circuit simulations.


Error

Circuit simulation save files end with ".les".

This error occurs for the same reason as above but for when you are in the Boolean algebra tab. The program can only load for “.les” files when loading a Boolean expression.


Error

You have entered too much text.

This may occurs when you enter lots of text into the input box. Upon pressing enter the input box is cleared. The limit of characters that you can enter into the input box should normally never be exceeded. If you have this error but have only entered a few characters, please submit a bug report.

Reporting Bugs

If you have found a bug, then please email me at Qi-rui.Chen@hotmail.co.uk with the subject “Logic Gates Bug”. Please provide the following information when submitting bug reports:

- OS Version (e.g. Windows 7, Mac OSX 10.10, Red Hat) and architecture (32bit, 64bit)
- Python Version (3.X.X)
- Pygame Version (1.9.X)
- System Memory (aka RAM)
- Detailed report of what happened or how to reproduce the bug

Appraisal

Feedback from Stathem and Starkings

Currently the following objectives have been met:

- The system must work at school

The program runs smoothly on school computers. The program seems to have no problems even running on low end computers with only 0.5GB of ram.

- Boolean expressions should be formatted correctly using over bars, dots and plus signs

The program can correctly format all expressions.

- Boolean expressions that are inputted must be able to be simplified as much as possible

The program can simplify any Boolean expression into Boolean literals and then apply identities on them. The result is often not as pretty as simplifying by hand.

- The system must have a list of Boolean identities that can be shown to the user

The help tab contains a list of Boolean identities.

- Boolean expressions must be able to be converted into truth tables, with all the steps

Boolean expressions inputted into the program are automatically converted into truth tables.

- The system must be able to simulate a logic circuit

The program can correctly simulate the logic circuit, including the propagation delay between components.

- Logic gates must be of the ANSI/IEEE Standard 91a-1991: "Distinctive Shape"

The logic gate images are taken from the IEEE Standard symbols.

- The system should be able to create a Boolean expression from a logic circuit and vice versa

The system can do this, but when converting from a logic circuit to a Boolean expression, a bulb is required to signify the end point.

- Boolean expressions should be able to be saved

Boolean expressions are saved as ".lgs" files.

- Logic circuits should be able to be saved

Logic circuits can be saved, but only if they can be converted into expressions. This is because unpicking boolean.py classes causes problems.

- Truth table should be able to be saved

Truth tables can be saved as images, align with correctly formatted expressions.

The following objectives have not been met:

- Given a Boolean expression and logic circuit, the system should be able to verify if they are equivalent

Given a Boolean expression and logic circuit, the system is currently able to verify if they are equivalent if the user follows a precise procedure. The current code has the ability to do this but it has not been added to the final system.

- The system must have an example for De Morgan's Law

This objective has not been met due to time pressure. Adding an example for De Morgan's Law is easily possible with the current code. More communications with the user should have been done to communicate exactly what the user wanted

- The system could create questions for the user to answer

This objective was listed as a "could", and would have only been implemented if there was enough time. The current system cannot randomly generate logic expressions or gates.

- The system could have a tutorial section which teaches the basics of Logic Gates
- The system could have a tutorial section which teaches the basics of Boolean algebra

These sections were eventually deemed unfeasible due to time constraints. For these objectives to be achieved more communications with the end user should have occurred about the details of how the tutorial section works.

Possible Extensions

Aside from completing the objectives that have not been fully met and ones that have not been met at all, the program could also contains a system that allows teachers to create problems for students. This would be easier than creating a problem by hand as the program can easily verify the solution