

# A Simple Logic Circuit Simulator

Amanda Matthes

ID:10241789

May 12, 2018

# Contents

<b>1</b>	<b>Abstract</b>	<b>3</b>
<b>2</b>	<b>Introduction</b>	<b>3</b>
<b>3</b>	<b>Code Design and Implementation</b>	<b>4</b>
3.1	GlobalVariables.h . . . . .	4
3.2	User input handling . . . . .	5
3.3	The Grid class and how rendering works . . . . .	5
3.4	The Component class hierarchy . . . . .	8
3.5	BoardCreator.h . . . . .	9
3.6	Demonstration.h . . . . .	9
<b>4</b>	<b>Results</b>	<b>9</b>
<b>5</b>	<b>Discussion and Conclusions</b>	<b>10</b>
5.1	Comparison with my original proposal . . . . .	10
5.2	C++ features that were not used . . . . .	11
5.3	Possible improvements and extensions . . . . .	11

# 1 Abstract

This project is a simulator for logical circuits consisting of 1/0 inputs, wires, gates and LEDs as Outputs. The circuits live on two dimensional *grids* (here also referred to as boards) on which the various *components* can be placed. The program then uses a simple depth first search (DFS) starting from the inputs to determine the state of the outputs. I call this *rendering the board*. For the various components I created a class hierarchy. There also is a separate class for grids which uses a vector of (base class pointers to) components.

There are several preprogrammed examples, but the program also allows the user to create their own boards. To make it possible for the user to switch between creating custom boards and requesting examples this program uses main and sub-menus.

Consult figure 1 for the symbols used in the program and this report.



-----	
True input:	1
False input:	0
LED on:	
LED off:	
AND gate:	&
OR gate:	
NOT gate:	!
Wire:	+
-----	

Figure 1: Legend

# 2 Introduction

Logical circuits are different from most circuits considered in physics in that we are only interested in whether there is a current or not rather than in a precise value of it. These two states correspond to the logical *true* and *false*.

Inputs are idealised sources for these logical values. In this project we use mock LEDs as outputs that turn on if the state of that location is true and vice versa. Have a look at figure 2 to see what that looks like in our program.



	0	1	2	3	4	5	6	7	8
0									
1			1	+	+	+	+		
2									
3			0	+	+	+	+		
4									

Figure 2: Inputs, Wires and Outputs



	0	1	2	3	4	5	6	7	8
0									
1			1	+	!	+	+		
2									
3			0	+	!	+	+		
4									

Figure 3: Not gates

Gates are also logical components. The NOT gate inverts the state (see 3). There also are gates that have two inputs like the AND gate. The output of an AND gate

is true if and only if both inputs are true (4). My program also supports OR gates. The output of an OR gate is false if and only if both inputs are false (5). In my program inputs are on the top and bottom of a gate and it outputs to the right.

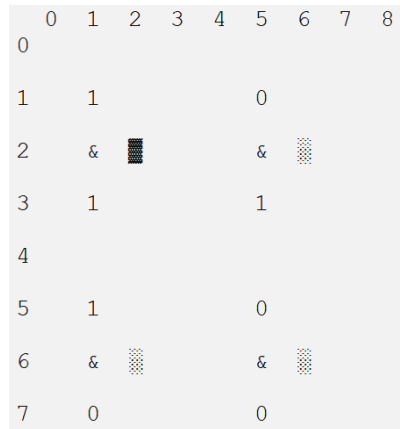


Figure 4: And gates

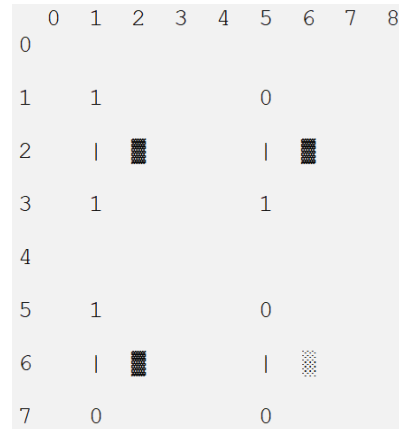


Figure 5: Or gates

The objective of my project is to simulate this behaviour and make it possible for the user to create their own circuits.

## 3 Code Design and Implementation

### 3.1 GlobalVariables.h

Global Variables.h contains the headers that are used in (almost) all files of the project as well as some important simple functions, enums and structs. This header is included in all other files (directly or indirectly). That way every other file does not have to include all these headers explicitly. This keeps the code short. Figure 6 shows this header.

The first two and the last line guard this header to make sure that it is not included more than once. The *location* enum is used to tag where the search is coming from when we render the board. I will be coming back to this. The *type* enum allows us to tag component objects to make them easy to identify. When taking user input we often want to take two integers. To make encapsulation possible we need to have a return type for this. The *intPair* struct makes this possible.

*endProgram(string message)* is used to exit the program if errors occur or if the user exits the program. It displays the *message* and ends in an infinite while loop to keep the window open.

```

#ifndef GLOBALVAR_H
#define GLOBALVAR_H

#include <iostream>
#include <iomanip> // included to use setw for nicely formatted output
#include <string>
#include <vector> // rather than traditional arrays

using namespace std;

// location is used for the graph search to indicate where the search is coming from:
enum location { UP, DOWN, LEFT, RIGHT, START };

// type is used to tag objects so that the board can be rendered efficiently:
enum type { INPUT, OUTPUT, AND, OR, NOT, WIRE, EMPTY };

struct intPair { int first; int second; };

char const on = 178; // symbol used to indicate that an LED is on
char const off = 176; // symbol used to indicate that an LED is off

bool isGate(type t); // returns true if t is a gate

void endProgram(string message);

#endif

```

Figure 6: Global Variables.h

## 3.2 User input handling

Figure 7 is an example for what user input handling looks like. It was taken from *void menu()* in Main.cpp but the structure is always the same. The while loop stays active until the user exits it. Invalid input keeps the user in the loop until they provides valid input. The code lists the possibilities and assigns them letters to keep it simple. *getline(cin, input)* comes from the string header. It takes the entire line of user input and saves it in the string *input* which gets overwritten in each iteration.

## 3.3 The Grid class and how rendering works

There is a class for grids (Grid.h and Grid.cpp). Its member variables are the integers *width* and *height*, the base class pointer vector for the components on the board *data* and finally the *visited* vector of the same length. The latter will become important later, when we start looking at how rendering works.

To switch between the index used in *data* and the more descriptive (row, column)

```

string input;                // temporarily stores the input
while (true) {
    // list possibilities:
    cout << "What do you want to do? \n";
    cout << "A: Show me some examples \n";
    cout << "B: Show me the legend \n";
    cout << "C: I want to make my own circuit \n";
    cout << "D: Exit \n";
    getline(cin, input);
    // handle invalid input:
    if (input != "A" && input != "B" && input != "C" && input != "D" && input != "E") {
        cout << "Invalid input" << "\n";
    }
    if (input == "A") {
        demonstration();
    }
    if (input == "B") {
        printLegend();
    }
    if (input == "C") {
        boardCreator();
    }
    if (input == "D") {
        break;
    }
}

```

Figure 7: Input Handling (taken from menu() in Main.cpp)

pairs, my program uses

$$x = c * h + r$$

$$r = x \bmod h$$

$$c = (x - r) / h$$

where  $h$  stands for the height of the grid,  $x$  stands for the index and  $r$  and  $c$  for row and column.

Apart from the standard constructors<sup>1</sup>, the destructor, the getter methods and a  $\ll$  operator we have the following member functions:

*void add(int row, int column, type t, bool state = false)* and *void remove(int row, int column)* respectively add or delete components at the specified location.

*void render()* and *void visit(int index, location comingFrom, bool state)* are the two functions that implement the grid rendering by running a DFS starting from the inputs. The code is relatively long, so it is not included in this report but the idea is very simple. Recall that the *visited* vector has the same length as *data*. It functions

<sup>1</sup>For all classes in this project, the compiler generated assignment operator, move assignment operator, copy constructor and move constructor are either adequate or never needed.

as a property map in this graph search. The value of *visited* at index *x* is true if the corresponding component in *data* has already been visited in the search.

This is what *render()* does:

- Reset the board:
  - Reset the property map  
Set all elements of *visited* to false
  - Reset all gates  
As we will see, gates have member variables that store whether the output is already defined. We may want to render the same board several times, so we need to reset these to false
  - Reset all outputs
- Iterate over the grid and for every component of type *INPUT* call *visit(x, START, state)*, where *x* is the index of the component and *state* is its state.

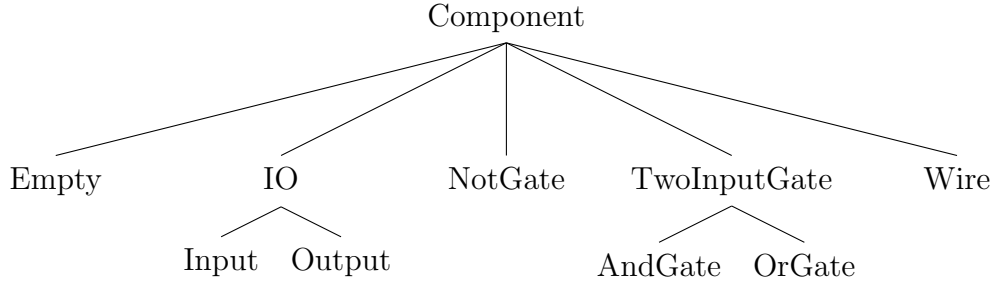
So what does *visit(int index, location comingFrom, bool state)* look like?

- If *visited[index] == true* then we can just return, because we have already been here. The exception to this rule is when we are at a gate. These can (and usually should) be visited several times
- This step depends on the component type we encountered
  - OUTPUT: Turn the LED on or off depending on *state*
  - EMPTY: Return
  - AND or OR: Set the lower or upper input depending on where we are *comingFrom*. Check if the output is now defined and if so continue the search to the right with that output (by calling *visit(...)* recursively for that neighbour with *comingFrom = LEFT* and the output as *state*). Return
  - NOT: flip the state
- Recursively visit all neighbours (*UP, DOWN, LEFT, RIGHT*) with the possibly altered *state*

The search ends if all neighbours have been visited or are empty.

### 3.4 The Component class hierarchy

For the components I created a class hierarchy like this:



They all have their own .h and .cpp files. An empty cell here is also formally a component. This makes it possible to use a vector of components to represent a grid. All objects of *Component* and its derived classes have the two members *char c* and *type componentType*. The *componentType* tags the objects with their type. This might seem unnecessary, since obviously an object of the class *Wire* will be a wire. However, we know that grids are represented by a vector of *Component* base class pointers and C++ does not have an easy way of determining which constructor was called when a specific object that a pointer points to was created. This tag makes it possible to easily identify a *Component* pointer. *c* holds the character that is used to display this component on the console.

The derived class *TwoInputGate* has four other members: *bool upperInputSet* and *bool lowerInputSet* which are true if the respective input is defined, and *bool upperInput* and *bool lowerInput* which hold the respective states. The derived class *IO* has the additional member *bool state*.

*Component* has the following (virtual) member functions:

- *bool getState() const* which for *IO* components gets overridden with a getter for the state of the component
- *void flip()* which for *Input* components flips the state ( $1 \rightarrow 0$  and  $0 \rightarrow 1$ )
- *void setLED(bool newState)* which for *Output* components allows the user to set the *state*
- *bool upperInputDefined()* and *bool lowerInputDefined()* which act as getters for *TwoInputGate* objects
- *void setUpperInput(bool newValue)* and *void setLowerInput(bool newValue)* which are the corresponding setters.



- *void resetGate()* also gets overridden by *TwoInputGate* and sets *upperInputSet* and *lowerInputSet* to false
- *bool outputDefined()* returns true if both *upperInputDefined* and *lowerInputDefined* are true (for objects of *TwoInputGate*)
- *bool getOutput()* which returns the output (of a *TwoInputGate* object) if it is defined, otherwise it calls *endProgram()*. So we always need to check *outputDefined()* first

For all components for which these function should never be called (like *flip()* for a wire for example) the overriding function displays an error message and ends the program by calling *endProgram()*.

### 3.5 BoardCreator.h

Everything related to creating boards is in the file *BoardCreator.h*. The function *void boardCreator()* acts as a sub menu for this part. We will come back to this function in section 4. The three other functions in this file are *intPair askForTwoInts(string message)*, *type askForType()* and *bool askForBool()* which ask the user for two integers, a type or a bool respectively and return those.

### 3.6 Demonstration.h

Everything related to examples is in the file *Demonstration.h*. Here the function *void demonstration()* is the submenu from which the user can navigate examples. The examples are simple hard-coded grids that are encapsulated in their own void functions and get called if the user requests them.

## 4 Results

This section is about how we use all this code to make an interactive simulator. To keep it short I will not explicitly mention every line of input checking.

The program starts in *Main.h*. It displays a welcome message (*welcome()*) and then enters the main menu (*menu()*):

What do you want to do?

- A: Show me some examples  
Calls the menu *demonstration()* in *Demonstration.h*.

- B: Show me the legend  
Calls *printLegend()*
- C: I want to make my own circuit  
Calls the menu *boardCreator()* in Board Creator.h
- D: Exit  
Leaves the menu, then *main()* calls *endProgram()* in Global Variables.h to smoothly end the program and keep the window open.

If the user chooses option A they will enter the demonstration menu which lets the user choose between a few examples and exit back to the main menu. If the user chooses option C they will enter the board creator menu (*boardCreator()* in BoardCreator.h). This will first ask the user for the width and the height of the board they wants to create by calling *askForTwoInts(...)* and creates a grid with those dimensions. It then it gives the following options:

What do you want to do?

- A: Add a component
- B: Delete a component
- C: Flip an input
- D: Show me the legend again
- E: Exit to main menu (Deletes the board)

Options A, B and C use *askForTwoInts(...)* to get the location on the grid that the component is to be placed/ removed from/ flipped. Option A will also ask the user for the component type with *askForType()* and if the user wants to place an *INPUT* will use *askForBool()* to ask for the state. The program then calls *add(...)/remove(...)/flip()* with the respective arguments. Option D calls *printLegend()* as before and Option E returns and exits *boardCreator()*. By doing that, the created grid goes out of scope and the Grid destructor gets called to free the memory.

## 5 Discussion and Conclusions

### 5.1 Comparison with my original proposal

This solution implements all minimum features that I included in my original proposal for this project. Under “ideas for additional functionality” I listed “other

gates” (than AND and OR) and “a demonstration of one or more simple circuits”. I did include the NOT gate but to keep it simple I did not implement more exotic gates like XOR. I included several examples that demonstrate all the components but I found it very hard to think of more interesting circuits that are actually used in computers and could be implemented in my simulation. This is not due to missing components but due to the problem of wire crossing. This could only be achieved by implementing a proper GUI.

## 5.2 C++ features that were not used

There was no real way to make use of static data since there is no information that all components or all grids need to be able to modify.

I also refrained from using templates. The only time it felt useful to make use of generic functions was when I implemented the three functions *intPair askForTwoInts(...)*, *type askForType()* and *bool askForBool()* which all take input of some kind. However, they require very different implementations and more importantly in C++ the return type must be known at compile time. So it is not possible to use templates for the return type.



























I could have replaced the Component class hierarchy with one template class but then my program would have no subtyping which seemed like a more important part of this course.

## 5.3 Possible improvements and extensions

As mentioned above, one improvement would be to implement a more sophisticated GUI to make wire crossing possible. Unfortunately, this goes beyond the scope of this kind of project.

A second extension could be the support to save boards as simple text files. To do that one could either simply use the visual representation already in use or a more abstract format. Those files could then be loaded from the main menu, altered and then saved again.

Another more substantial addition would be to output timing diagrams at labelled locations. This would allow the demonstration of more practical circuits like flip-flops.

	0	1	2	3	4	5	6	7	8	9	10	11
0												
1												
2				+	+				+			
3			+	+	+	+	+	+				
4			+		+	+	+	+				
5			1		+			+				
6												
7	