Project Report

Introduction

The objective of this project is to create a Digital Logic Simulator tailored for educational use. Its purpose is to facilitate students' comprehension and exploration of digital logic circuits. Leveraging Java Swing, the software will extend an existing basic paint application framework to allow users to construct, visualize, and simulate circuits incorporating various digital logic gates (such as AND, OR, NOT, XOR, etc.). The simulator will offer an easy-to-use interface for gate placement, wire connections, and the simulation of logical operations.

Background

This project takes heavy influence from Logism. While far simpler in its aims, it hopes to utilize the many useful features and interface designs whilst also providing an alternative approach to some aspects of the interface that some users may find frustrating. Features such as wire's auto snapping to the grid will not be included such that users may better organize their designs. It will not feature the multitude of gates which Logism offers, however by including the basic logic gates it will provide users to make their own (adders, multiplexers, etc.).

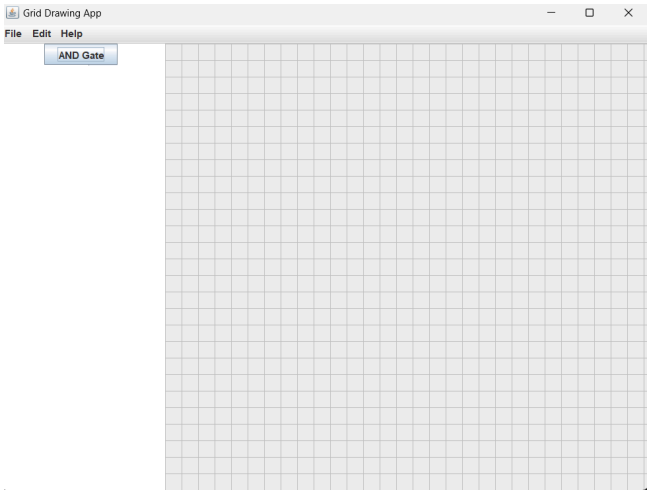
Methodology

Tbd … (at this stage it will be discussed in software design)

Implementation Details

Phase 1, UI Design:

This is an initial drawing of what the end product software should resemble. Part of usability is familiarity, such is that users should be expected to better understand software which is similar to that which they have used before. Menu options such as file and edit are included in their usual spaces. This program need not be complex with its design as its operations are fairly simple.



The functionality of the design should be simple. A user clicks their desired gate, and it is instanced in the workspace, after which, a user may drag it to their desired location. They will then likely want to add “wires” to connect any gates they may have in the workspace. These wires should follow the grid such that they can only exist along discrete coordinates. Such is that gates should only be allowed to be on the grid as well. From this design, by adding connection points to the gates and storing the end points of a wire, we can easily determine whether a wire is connected to a gate. This wire should then change color to demonstrate this or any other potential change to its state(in the case of simulation).

Let’s examine possible scenarios where this design needs to be improved upon. The grid exists in finite space, such that it may be too small for some designs. A zoom feature should be implemented that expands or deflates the grid to allow for easier editing. Additionally, if all gates are instanced in the same portion of the workspace, they may disrupt any current design; such that the addition of a designated area for them to be dragged from may be beneficial. Other problems could include a lack of a discrete field for input/output widgets (input/output widgets may behave similarly to a gate with this implementation, however they may seem that they belong to a different category to a user), the lack of any feedback to show that circuit is currently in simulation mode, or the fact that it is currently not possible to rotate gates to be in a more visually understandable position. These issues are user stories, some of which are potential issues observed in Logisim, and that this implementation corrects.

(Phase 2)

Initial challenges are instancing gates in the workspace comfortably. Images will be used for gates, such that they will need to be sized accordingly for their connection points to make sense. Wires should also snap to the grid and either only be allowed to be drawn on the grid such that it can be easily discerned whether they collide with a gate’s connection point.

To solve these challenges, the wiring tool allows for diagonal wires, however, the start and end points of these wires will automatically snap to the grid point nearest to the user's cursor when initiating and ending a mouse drag. The initial approach to this was simplistic such that wires did not carry data through connecting wires, nor did they ever remove the drag listener. This was an issue as my initial design for gate dragging utilized the same drag cursor, meaning that wires would be dragged alongside gates making the application virtually unusable.

To solve this, a wiring mode feature was added. Via the top menu, users could enter and exit a wiring mode so that they could drag gates without drawing wires. This approach didn’t feel particularly user friendly. Combined with the way gate draggin was initially implemented (it gave no visual feedback

along the drag and simply placed the gate where the user released their M1 button), an entirely new approach was required.
(Phase 3)

The solution was creating classes GateHandler and Gate. This solved a number of issues. Gate image resizing was initially problematic as I could not find a way to resize gates to a point at which they were reasonably sized and had connection points readily apparent. GateHandler addressed the issue with dragging by storing the information pertaining to it being a draggable component, as well as information regarding its input and output states. These draggable components will give the user feedback as they are both dragged and placed along the grid such that they are easily understood. Gate handles image processing and should be appended to in the future to accommodate for multi-bit wires and gates utilizing more than two inputs (as the image would require resizing).

While gates felt comfortable at this point, wires faced some notable issues. They could only be dragged in such a way where after a user released their cursor the wire was set and could not be altered or expanded upon. To solve this, all wires continue to be stored in an array, however, strands of wires (those which connect and extend from some output) are traced such that data can carry through multiple strokes as long as the endpoints of wires collide.

Testing and Evaluation

As this is primarily a graphical interface, testing came primarily from interactions with the application. There exists a couple known problems that still need to be addressed or expanded upon.

Wire tracing is imperfect. As it stands, if a user draws a wire that does not extend from an output point, it will not connect with the larger strand. The current implementation stores strands in a structure loosely resembling a singly linked list in which the head is the wire extending from an output point. The result is if there does not exist an output point in the strand, data will not carry through. This might be solved by using a doubly linked list where wires are stored in structures indicating strands instead of an array containing all wires to be drawn to the board. This implementation would require enough refactoring that I was unable to implement it given the point at which I realized it existed.

While not necessarily a bug, gates must have two inputs with the current implementation. I tested methods to allow for varying inputs, however significant issues with this approach were found too close to the deadline. With the initial approach, input states are stored in the Gate class from which the subclasses that handle gate behavior perform their respective logical operations. This implementation would overwrite itself if enough connections were established (e.g. a user drags a gate to connect with wires after already having connected it). This was addressed by hard coding in a two input mandate within the code, as this was yet another issue I noticed far too late. It could be solved by better storage of this data, most likely storing information pertaining to connections in an object such that wires and connection points are better represented with respect to their shared properties.

Results and discussion

This application is a comfortable design for very simple logic gate simulations. It meets much of the initial objectives in providing a user-friendly experience which can be easily understood even by users with little knowledge of logic gates. Features such as wires which can connect with each other, gates that

are draggable along the grid, and constant simulation provide real time feedback for users which is invaluable while learning new concepts.

Key Features and their reasoning

- Constant simulation is very much a conscious choice. If the aim of the application is to create a logic circuit that can simulate to provide some sort of feedback, why waste the users time with a button to turn simulation on. It follows more naturally to have them behave as elements awaiting enough voltage to discern a 1 or default to 0. This feature does not hinder the user experience but rather helps a user predict outcomes as they make their circuit.
- Draggable gates along a grid feel the most natural way to understand this concept. This design allows for users to truly be creative with their circuits and does not limit them to a certain number of gates nor a certain number of gates logic may pass through. The only limitation is the size of the board.
- Wire snapping via dragging a wire directly from output to input feels the most natural, and allowing the opportuning to drag along the grid with multiple strokes gives the user some room for error when drawing wires. The limitations to this are the aforementioned bugs with wire tracing.
- Toggleable drop down menus on all draggable components feel natural are something most users should be used to. This makes deleting gates and toggling power on the power blocks very straight forward.
- Removing the last drawn wire is a pseudo step in the right direction for a truly user friendly experience. Ideally this button would just be “Undo”, and the wires would be draggable components with their own deletion menu. However my implementation does not do this such that this is the best that can be done. Something to work on in future iterations!

Some key features that would be ideal (aside from the one mentioned above) are file save/load functionality, the ability to resize gate images, zooming in and out of the board, multi bit wires and gates, and functionality for components such as multiplexers or adders. In the case of file save/load, this was not necessarily a choice to not implement but rather a lack of time to do so. However, these other features should mesh well with the current implementation. The Gate class contains the necessary structure to implement resizing capability for gates which would intern allow for gates accepting more than 2 inputs and multi bit wires. This functionality was left out as implementing it in the future could be done with little change to the current code.

Conclusion

This application serves as a user-friendly platform for simple logic gate simulations, catering to users with varying levels of familiarity with logic gates. Its design prioritizes ease of use and accessibility, making it suitable even for those with limited knowledge of logic gates. Overall, the combination of these features ensures that the application provides a comfortable and intuitive environment for users to explore and learn about logic gates. Whether users are beginners seeking to grasp fundamental concepts or more advanced learners experimenting with circuit designs, the application offers a valuable tool for enhancing their understanding of logic gates.

User Manual

Installation and Execution

To build from source, use the included pom.xml to build in maven. The repository should also contain an executable jar file(inside the target directory) from which you can execute immediately.

Usage

Wires

Wires can be dragged by simply clicking M1 and dragging your cursor on the workspace and then releasing your M1 button. It will automatically snap to the nearest grid point to your initial click and snap to the nearest grid point upon release. Wires may be dragged horizontally, vertically, or in diagonals. Additionally, wires can be connected as long as they originate from the output point of a power block or gate (You may release your cursor, and start a new wire and the data will carry through). Note that they MUST come from the output point. If the first wire draw does not connect to an output point you cannot connect it with other wires.

Wires can be removed by navigating to the top left of the application and opening the “Wiring Tool” menu. Interacting with the only option in this menu, the last wire drawn to the workspace can be removed.

Gates

Gates are draggable components, listed in a menu towards the left of the application. Gates are instanced randomly, but within a certain field, in the middle of the workspace. They are then draggable along the grid and can be dropped at any point along the grid.

Gates have two inputs at the left of each gate. The input points are the top left corner of the image and the bottom left corner of the image. Dragging a wire to these points will result in a connection (denoted by a filled circle).

Gates can be removed by left clicking them and interacting with the drop down menu.

Simulation

This application does not have a simulation mode. Instead it elects to be permanently simulating. If gates have no connected inputs, their input states are by default zero. Such is that they adopt a false until proven true behavior. Power blocks can be used to toggle different inputs to test different outcomes for effective simulation.

Power blocks can be toggled by left clicking them and interacting with the drop down menu.

Software Design

Architecture Overview:

Most data in this application flows from interactions with the class Board. It maintains nearly all of the data and methods to process it. This class should be refactored into smaller parts. App Interface initializes it, and from there Board handles everything from drawing the wires to instancing the gates, as well as many helper methods to make sure proper handling of these components. It stores all instances of gates and wires in respective ArrayLists and iterates through these arrays, constantly painting them to the board. Methods for identifying successful connections also exist here by checking whether the connection points identified in class GateHelper that belong to each object in the gate array collide with the start or end points of any of the wires in the wire array. This information is then passed to their respective classes to either change the output state of a gate, or the appearance of a wire to then be passed back to board for display. Further explanation of the interactions between the classes board references can be found below.

Component Descriptions:

AppInterface

This is self explanatory from its name. It does little else than create the menus users interact with initially. It initializes the top menu, and the buttons for instancing gates. It is the first class to instance Board, from which all of the data pertaining to the actual simulator flows from.

Board

As this class need not be instanced more than once, it is implemented as a singleton. It contains the majority of all the data. It is broad in scope and should be whittled down in refactoring. It interacts with every component/class with the exception of AppInterface. Gates are instanced here as well as the logic for their instancing. Wires are also instanced here in addition to their logic. Wire tracing, board snapping, most of the major features of the application happen in this class. Connections between wires and wires between gates are identified here and their logic is passed into their respective classes for further processing. Additionally, this is where the grid itself is defined and drawn. It contains methods to locate the nearest grid point and is passed as a class attribute to many of the other objects.

GateHandler

This is the super class for all things gate related. It is the primary interaction between the board and gates. Class Gate extends from this, and xGate(s) extends from Gate making this class the holder of many of the important functionality pertaining to gates. This holds the data defining where a gate has a connection point for input and output (differentiating between the two) and contains methods to process input data to output data. Connections identified in class Board are passed to this class which xGate(s) then use to process the logic into output and then send the data back to Board where it can be displayed.

Gate

Gate is primarily a handler of image data. It is almost confusingly called gate at this stage as when it is inevitably expanded upon (and it should be expanded upon), it should contain the data pertaining to resizing and thus the methods that define where input and output points are should be moved (or just initially located) in here. Board instances this class rather than GateHandler as it was intended to be the more versatile of the two, however I never got to making images resizable.

Wire
Is essentially a data structure to store all the data pertaining to each wire. It contains information on the position, color, and state of each individual wire. This information is then passed back to Board to be painted to the workspace.

xGate(s)
These do little other than contain the associated image for each gate and provide a function for each gate's respective output logic. It's method overrides the method in the superclass GateHandler it inherits from. This logic is then passed to board so that a change in the state of the wire attached to the output point defined in GateHandler can be accurately painted to Board.

References and Appendices

Conversation 1:
The conversation begins with the presentation of a Java Swing application allowing users to draw lines along grid lines. The lines snap to the grid, with grid lines drawn in light gray and user-drawn lines in blue. Then, a suggestion is made to add a window on the left side for dragging in widgets representing logic gates. Following this suggestion, modifications are made to the code to include a side panel with draggable logic gate widgets, represented by JLabels. Much of the code pertaining to logic gate widgets was unusable such that I had to seek an entirely different approach.
Conversation 2:
The conversation is extremely similar to the one above with modifications to my language and more specific requests in the hopes of getting more usable code chunks. This was to no avail.

Conversation 3:
Again trying to rephrase to find an implementation of draggable widgets. Took some learning on my own. I don't think I even knew the right questions to ask at this stage.

Conversation 4:
The conversation begins with an anonymous user asking how to save a JPanel and all its components as a file, providing a Java class representing a board with wires and gates. ChatGPT responded by suggesting the use of serialization and file I/O operations and provided functions to save and load the board. Unfortunately due to the general structure of my program I was unable to implement all classes and subclasses as serializable. This would have been a helpful stage in refactoring but I never got there unfortunately.

Conversation 5:
Next, the user asks for Java Swing code that displays a right-click menu when a component is right-clicked. ChatGPT provides an example of how to achieve this. The user then specifies that the

right-click menu should only appear when hovering over a specific component. ChatGPT modifies the code accordingly.Following that, the user asks for guidance on updating the colors of connected lines in a Java Swift application. ChatGPT provides a simplified example demonstrating how to achieve this functionality. Each request is addressed promptly and with code examples provided to assist the user in implementing their desired features.

Conversation 6:
The conversation just revolves around me rewording different parts of this report that say the same thing in different fields.