Circuit Board Designer

Authors: Kenneth Shipley, Joseph Spear, and Jason Rivas

Design Description

1. Graphical User Interface

The Circuit Board Designer's first primary feature is going to be it's look. We want the program to have a simple and easy to understand interface, yet provide enough features to accommodate most circuit needs. The GUI will be comprised of three main portions: the workspace, a pullout menu or moveable pallet wheel (depending on time constraints) containing circuit design features (i.e. wire, wire snipper, add component, delete component, label, comment), hamburger menu and pull out file management features. PyQt will be used to develop the GUI elements of the Circuit Board Designer.

The implementation of the GUI revolves around three classes: Schematic, Component, and Comment. These three classes, (shown in figure 1), will allow the user to develop the board in a versatile way. The Schematic class will contain an array of component objects and an array of comment objects. The Schematic class will be the primary function used for organizing the board data as a whole. It will save the component data and will have each pallet button clickable from this class. The file management options will also be stored in this schematic class (save, load, open new, etc.) The next class, Component, will be used to store each individual type of circuit element. The component class contains various members to include the label, position data, and orientation data. The different types of circuit elements are: resistor, capacitor, inductor, NPN transistor, PNP transistor, switch, diode, voltage source, ground, and LED. These elements will be stored as individual subclasses, each with the ability to add a label and will call a draw function to actually implement it into the schematic. The last class, comment, is a small class to allow for annotation by creating a savable text box on the workspace. This textbox can be closed by clicking away, and it will turn into a small comment icon on one of the grid points of the workspace. PyQt will allow for the creation of the actual components by pulling from pre-stored images of each component.

The workspace itself will be plain white, while the default color of wires, components, and text boxes will be black. The structure of the workspace will be created using a cartesian grid system, with each grid point corresponding to a point on a 2-dimensional array. This array will allow the position of different components, wires, and comments to be stored once placed. There will be faint, gray lines connecting the grid points depending on the zoom level (i.e. at one zoom level, there will be lines for every 5 grid points, and at another zoom level, there will be lines at every grid point). When a zoom input is acquired, the workspace window will quickly

zoom in to a certain distance around the cursor, or the center depending on the type of input. There will be a home zoom button located in the top right menu (along with the hamburger menu) which will auto-scale back to the original viewing ratio. When an element is placed onto the workspace, a simple user prompt will be given to name the component, storing it as that component's "Label" member.These features will be implemented using modules from the PyQt library.

The Hamburger Menu and pullout Component list will act in similar ways with the hamburger menu being on the left side, and the features on the right. When either the file management button is pushed in the top left corner, or the features button on the top right corner, the options for the selected list will slide out in a menu tab covering a third of the page. When this is done, the option buttons will be clickable, and doing so will activate the function for that button (i.e. clicking the resistor button will turn the cursor icon into a resistor that can be dragged around to the desired grid point). If the desired tab is needed to be closed without actually selecting one of the options from the tab, clicking the workspace or the file management/feature button will close the tab without activating one of the buttons. When a workspace element is desired to be selected, either for drag/drop purposes or deleting purposes, the selected element will be highlighted with a faint blue outline.

The final desired GUI application is the pallet wheel. This is more of an advanced feature, which is an alternative to the feature list pull out menu, and will only be implemented in the event that time is available after creating the pull out menu. The pallet wheel will include "Component", "Wire", "Wire Snipper", "Remove", "Comment", "Color" (the relabel function will not be on the pallet wheel, but it will be in the feature pullout menu). When one of the options are selected, a subset of options will pop up in circular "bubbles" floating around the wheel if necessary. Once a final feature is selected, it will operate. For "Wire", "Wire Snipper", "Remove", and "Comment", the functions associated with these buttons will begin. For "Component", bubbles containing all 10 components will pop up around the main wheel, and will be equally spaced in a circle. When this is happening, the "Component" button will be highlighted yellow. When the "Color" button is clicked, bubbles containing 10 colors ("Black", "Red", "Blue", "Green", "Purple", "Pink", "Brown", "Yellow", "Gold", "Silver"). When this is happening, the "Color" button will be highlighted yellow. One of the major design features of the Pallet wheel, is its ability to be moved around. At the center of the wheel will be a small icon which will be able to be clicked and dragged moving the whole pallet wheel. This will allow for a desired "cleanness" of the workspace for the user. If the central icon is clicked without being dragged, the entire wheel will collapse into a small circle, which can also be dragged around to the desired location for ease of work.

2. Component Database

The "database" of components will not be a standard database as there will only be 10 components in the system for the user to choose from so the "database" will be internal, in a .json file that is loaded on startup. There will be a resistor, capacitor, inductor, NpN BJT (Bipolar Junction Transistor), PnP BJT, switch, diode, LED, voltage source, and ground. The database itself will just be two files: a .json file where the ten components are each objects that have some descriptive fields and a sprite sheet that has the sprites for every component. The fields for the components: the central sprite position for the component; the positions of the pins, for connecting wires in the schematic builder; the sprite index, for getting the sprite for this component from the sprite sheet; the positions of the solder pads relative to the center of the component; and the size of the solder pad (dimensions of a square). This is enough for editing a schematic using sprites using PyQT and for drawing solder pads using PIL as described in (5).

3. Saving and Loading a Schematic

Schematics are saved by using a to_dict() function in the Component, Comment, and Schematic classes that creates a dictionary for the schematic that is then converted to a JSON object and written to a file using the JSON module. Specifically, the schematic class is converted to a dictionary with the keys: "components", "comments", "paths", "iteration_num", "last_run_score", "this_run_score", and "n_grid_spaces". The first two keys, components and comments, correspond to lists of dictionaries that represent the components and the comments respectively. Each component has the keys: "label", "id", "schem_position", "schem_orientation", "pcb_position", "pcb_orientation", and "connections". Each comment has the keys: "text" and "position". The schematic dictionary is then fed to the json.dump(dictionary, filename) method where it creates a .json file which we can look at for debugging and for rudimentary interaction with the program.

Schematics are loaded using a reverse of the save function. Using json.load(filename) we get back the schematic dictionary the user saved previously and then use a constructor for the Schematic class to reassign the values to a new schematic thereby recreating the schematic saved.

4. Monte Carlo and A*

Metropolis' Monte Carlo method is an algorithm based on pseudo-randomness. Basically it randomly places components in a gridspace, checks how well it worked out, perturbs the layout, restarts with the perturbed layout.

The method first uses numpy's random number generator to randomly choose grid spaces for a components' pins. Then it initializes a set of heuristic matrices to be used for finding paths for each connected component. The heuristic matrix (h(i, j)) we will use matches up to the grid that the components are on where each element (h(i, j)) is the distance from that element, (i, j), to the goal, (a, b) for example. A* takes H, the set of h's, and the connections list, which are the starting points and the goals. For each h, start, and goal, it calculates two things: a g-score and an f-score; these are used to figure out which (i, j)'s are a step in the "right" direction. As a side note, it keeps track of components and other paths created by treating them as obstacles or points with infinite distance to the goal; this way, it won't overlap anything. Once the algorithm finds the path for a given h, start, and goal, it retraces the path and puts each grid point in a list that is stored in a list called "paths", a field of the Schematic class. A* does this for the entire set of connections. Next, the Monte Carlo method chooses to accept or reject the layout based on a calculated score for the layout. This calculation takes into account the total path length and the area of a board that would fit the components. The score is weighted based on what we believe, or find, produces a more "optimized" layout, which will take some testing. Monte Carlo accepts the layout if the score is better than the previous iteration, and it rejects otherwise. If the layout is rejected, we take the previous layout as the input for Monte Carlo's next iteration; if the layout is accepted, we take the new layout as the input for Monte Carlo's next iteration. After the layout is decided, we lock the shortest path and the components it connects and then input the layout to the Monte Carlo method. The components and path that are locked will simply be obstacles for A* to avoid in the next iteration. Components that are locked for one iteration may be unlocked and randomly placed again if another set of components' path is shorter. The stopping conditions for Monte Carlo are either a "no possible solution" case or a "failure to improve" case (by a certain amount). The output of the algorithm is stored in the paths field

5. Diagram Image Converter

The end result of the image converter is an image of a pcb layout similar to that of Figure 2.This will be done using an imaging library named Pillow. This function will take in the data output from the Monte Carlo function, the paths list, to begin drawing. The location of each part in Monte Carlo will be important as it will allow the converter to map the parts onto a physical space. Once all the components are mapped to a physical location, all the traces, lead pads, component outlines, and component labels will be drawn. The function will start with a black rectangle representing the board. Any traces will be drawn with lines in red. These lines will only be able to turn at right angles for the sake of simplicity so there will be no diagonal lines. The lead pads will be a yellow square with a hole, this indicates where

there is a physical hole in the board that would allow a component to be soldered. While it is not expected to have any component outlines on the board, if it is necessary green outlines of shapes like rectangles will be used to indicate the direction at which a particular component should be installed like the op amp in figure 1 below labeled U1. The component labels will be in green as well and placed nearby the pads for the particular component. If the component has a horizontal placement, the label will be placed above the pads, if the component has a vertical placement, the label will be placed to the right of the pads. In the case of a diode, electrolytic capacitor, and LED a + symbol will be placed next to the appropriate pad as those components are directional.

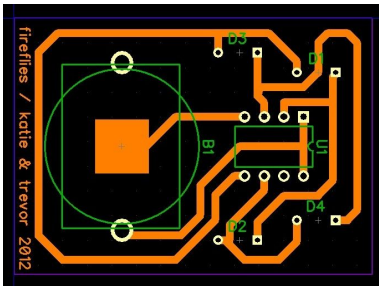

Figure 1: PCB layout example
Source: http://www.trevorshp.com/creations/fireflies.htm "firefly PCB layout"

6. The Output

The output is a .jpeg image that one could print out. This has its uses in DIY manufacturing of printed circuit boards. If you print out the image onto a ripped out page of a magazine with a lot of ink on it, you can use an iron to transfer that image, pcb layout, onto a copper substrate to then etch out traces on the board. This step is where figure 2 is at. The next step would be to use a chemical dissolving agent to dissolve/etch away the copper not covered by the magazine ink. Then you have an actual circuit board that you need only solder components onto.

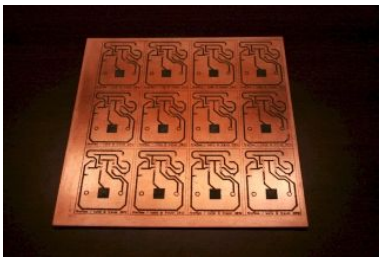

Figure 2: PCB layout on copper substrate
Source: http://www.trevorshp.com/creations/fireflies.htm "toner transferred on copper board"

Requirements:

- **A Circuit Builder:**
  - **A.1** GUI
    - **A.1.1** Implemented using PyQT
    - **A.1.2** Canvas/Workspace
      - **A.1.2.1** The workspace will have a canvas that represents the 2D grid which contains points for position data that is needed. There will be a grid pattern to show the points a component can be placed.
        - **A.1.2.1.1** Infinite/Finite option.
          - **A.1.2.1.1.1** Canvas dynamically change size to enclose only the components that are in view and some amount of padding (space around the outside).
        - **A.1.2.1.2** Zoom in/out
          - **A.1.2.1.2.1** [ctrl] + [+] and [ctrl] + [-]
          - **A.1.2.1.2.2** [ctrl] + [scroll up] and [ctrl] + [scroll down].
          - **A.1.2.1.2.3** Zoom option
            - **A.1.2.1.2.3.1** Zoom relative to mouse
            - **A.1.2.1.2.3.2** Zoom relative to center of canvas
            - **A.1.2.1.2.3.3** Home zoom button that sets the zoom to the default zoom setting
        - **A.1.2.1.3** Component Highlighting
          - **A.1.2.1.3.1** When a component is selected for an action, it will be highlighted in a light blue color to show the user a visual cue of what is being interacted with.
      - **A.1.2.2** Drag/Drop
        - **A.1.2.2.1** When a component is dragged around, if it is connected to another component via a wire, only the component being dragged will move, and its location will be updated. The connected components will remain where they and the connections will update wherever needed.
    - **A.1.3** Pallet Wheel Widget/Tools(Buttons)
      - **A.1.3.1** Can be expanded and collapsed
        - **A.1.3.1.1** When the center of the widget is clicked, it can expand to show the tools available or collapse to keep workspace minimal.

A.1.3.1.1.1 When expanded, each option will also expand with its options in another sub-wheel.

A.1.3.1.1.1.1 Sub-wheel is concentric to the main wheel.

A.1.3.1.1.2 Option expanded will be highlighted.

A.1.3.2 Can be dragged around

A.1.3.2.1 Can be dragged by pressing and holding while moving mouse

A.1.3.3 Tools/Functions

A.1.3.3.1 Wire

A.1.3.3.1.1 Allow the user to draw a line connecting two components together.

A.1.3.3.1.1.1 Wires can be colored to differentiate from other wires.

A.1.3.3.1.1.2 The pin the user first clicks gets the IDs of the following pins the user clicks on which adds these ids to the list of connections in **D.1.1.2.5**

A.1.3.3.1.1.3 All that is needed to draw a wire the location of the two components that are being connected.

A.1.3.3.2 Wire Snipper

A.1.3.3.2.1 Allow the user to cut a wire that they had previously drawn, removing the connection they made.

A.1.3.3.2.1.1 This will remove connections from the data of each component in **D.1.1.2.5**

A.1.3.3.3 Add Component

A.1.3.3.3.1 Pop up another menu of components available to add

A.1.3.3.3.2 Allow the user to pick which component they want to add, then drag it to the workspace

A.1.3.3.3.2.1 Will create a new component with the selected one's type in its "Type" field

A.1.3.3.3.2.1.1 Adds it to the list of components

A.1.3.3.3.2.2 Will create a sprite for the component selected that will follow the mouse until the user lets go.

A.1.3.3.3.2.3 Will place the component on the grid point using it as its location

A.1.3.3.4 Delete Component

A.1.3.3.4.1 Allow the user to pick which component they want to remove from their schematic

A.1.3.3.4.1.1 Will remove any connections the component had

A.1.3.3.4.1.2 Will remove the component from the list of components used

A.1.3.3.4.1.3 A component that gets its connection deleted will have that wire removed and the corresponding pin will no longer be connected.

A.1.3.3.5 Label

A.1.3.3.5.1 Allow user to label/relabel a particular component

A.1.3.3.5.2 When a component is added, the program will prompt the user to add a label to the component

A.1.3.3.6 Comments

A.1.3.3.6.1 Comments will be used in order to annotate the board.

A.1.4 Hamburger Menu/File Management (Buttons)

A.1.4.1 Save Project

A.1.4.1.1 This will save the project as described in **D.1**

A.1.4.2 Load Project

A.1.4.2.1 This will load the project as described in **D.2**

A.1.4.3 Create New Project

A.1.4.3.1 Ask user if they want to save then save or don't save (if current instance is not blank)

A.1.4.3.2 Delete current instance of the schematic class (**D.1**)

A.1.4.3.3 Instantiate a new schematic

A.2 Database of components (Surface Mount Device only)

A.2.1 Will have 10 types of components that the user can choose from

A.2.1.1 Resistor, Capacitor, Inductor, Bipolar PNP/NPN Transistor, Switch, Diode (rectifying and light emitting), Voltage Source, and Ground.

A.3 Data Management

A.3.1 All components created will create instances of objects with the following fields/member variables, all of which are described in sub-numbers of **D.1.1.2**

A.3.1.1 Label, Type, ID, Position, Connections, and Paths

A.3.2 All components are added to a list (or dictionary) as they are created, and removed from the list as they are deleted.

B **Optimization of PCB Layout:**

B.1 Grid System

B.1.1 The layout for the PCB will be made on a grid (3D grid if we get time)

B.1.1.1 All components will be laid on the vertices of the grid

B.1.1.2 All component connections will run along the grid connecting the components together

B.1.1.3 2D vector (or 3D if we have time)

B.2 Monte Carlo

B.2.1 Input Arguments

B.2.1.1 List of Components

B.2.1.1.1 List of dictionaries described in **D.1.1.2**

B.2.2 Output Data

B.2.2.1 List of Components now with path data

B.2.2.1.1 List of dictionaries described in **D.1.1.2**

B.3 A-Star (A*)

B.3.1 Input Arguments

B.3.1.1 List of components

B.3.1.1.1 List of dictionaries described in **D.1.1.2**

B.3.2 Output Data

B.3.2.1 List of components with updated path data

B.3.2.1.1 List of dictionaries described in **D.1.1.2**

C **Diagram-to-Image Exporter:**

C.1 Produce an image in either JPEG/PNG

C.1.1 The exporting software will take in the resulting position data from the PCB Optimization.

C.1.1.1 Uses PIL to draw rectangles for solder pads and lines for traces

C.1.2 The program will save the data using functions described in **D.1** before exporting to an image and save the output from **B**

D **Saving/Loading a Circuit Project:**

D.1 Save to .json

D.1.1 This will save the current schematic and all its components to a .json text file that will allow the program to reload it in another session.

D.1.1.1 Every project will have a base of a Schematic class

D.1.1.2 Schematic will be made from Components, components have

D.1.1.2.1 Label

D.1.1.2.1.1 Name given to a component by the user, only there for the user

D.1.1.2.2 Type

D.1.1.2.2.1 Type of component as described in **A.2.1.1**

D.1.1.2.3 ID

D.1.1.2.3.1 Number given to the component for the program to reference

D.1.1.2.3.2 Each of its pins will be: ID_#

D.1.1.2.4 Position

D.1.1.2.4.1 Circuit Position

D.1.1.2.4.1.1 Relative to top-left of screen and corresponds to the center of the sprite drawn

D.1.1.2.4.1.2 Component has a location on the workspace showing where the user placed the component

D.1.1.2.4.1.2.1 Just for redrawing the circuit when loading

D.1.1.2.4.2 PCB Position

D.1.1.2.4.2.1 Component has a location on a grid showing where the component is located on the pcb

D.1.1.2.4.2.1.1 Relative to top-left of screen and corresponds to the central location of the component's solder pads

D.1.1.2.4.2.2 Sub-Positions

D.1.1.2.4.2.2.1 Location of the solder pads relative to the PCB position

D.1.1.2.4.2.3 Will be null if the circuit has not been converted to a pcb

D.1.1.2.5 Connections

D.1.1.2.5.1 Contains dictionary of connections where the key is the starting pin and the values are the connecting pin(s)

D.1.1.2.6 Paths

D.1.1.2.6.1 Starts out null.

D.1.1.2.6.2 List of points describing path for the connections described in **D.1.1.2.5.1**

D.1.1.3 Schematics will also have comments, comments have

D.1.1.3.1 Comment

D.1.1.3.1.1 A string for written by the user for the comment

D.1.1.3.2 Position

D.1.1.3.2.1 Location of the comment on the workspace

D.1.2 If a user tries to exit the program without saving, the program will warn the user of this and ask if they would like to save

D.2 Load project from .obj

D.2.1 This will load in the .obj file that saved a previous Schematic instance

D.2.1.1 The loaded schematic object will be assigned to a new instance of the Schematic class

Appendix

Class Diagram

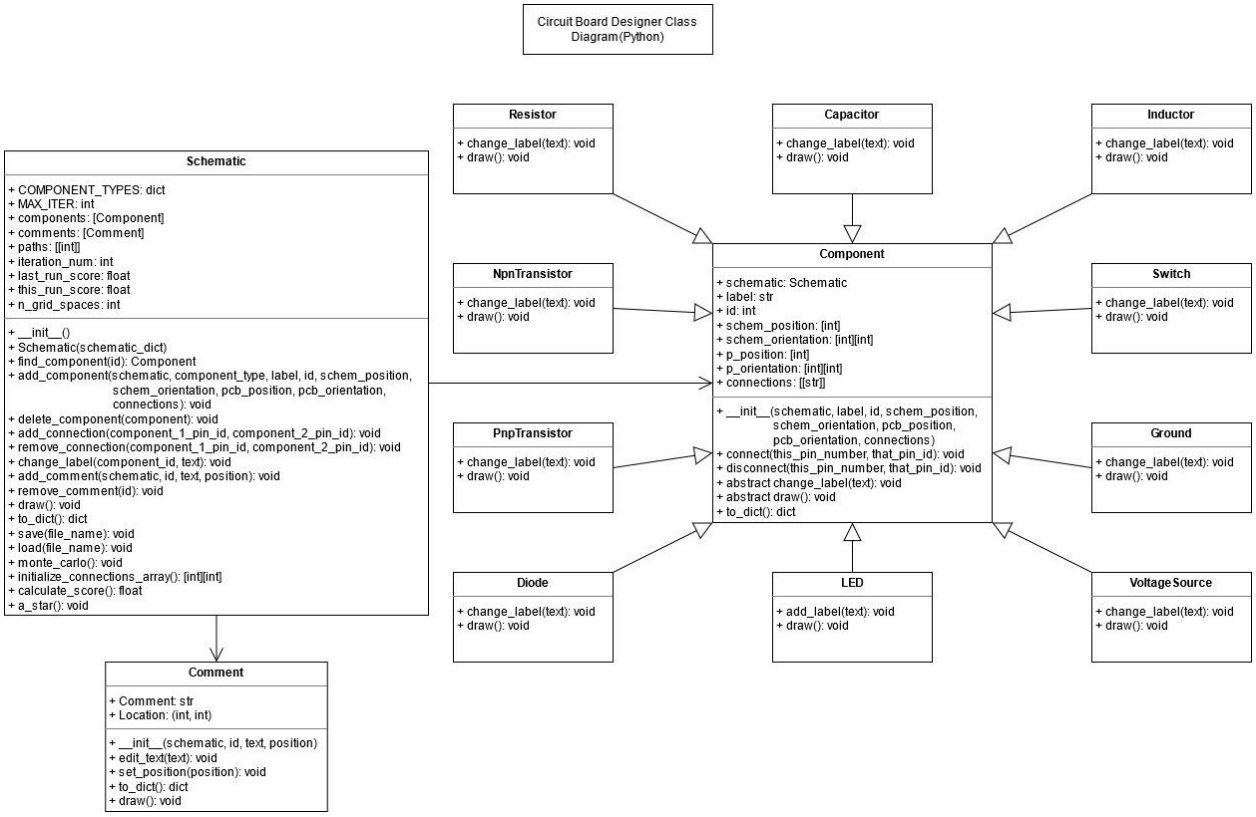

Figure 3: Class Diagram

Storage

When the user decides to save the project it will save the state of the schematic into a .json file. An example json file can be found in the github repository under **src/test.json**. The json file will consist of a list of objects that include things such as components, comments, paths, iteration numbers, previous run score, current run score, and the number of grid spaces for the schematic. The components, comments, and paths will be a list and other objects will be numbers. The component list will contain the components used in the circuit diagram where it will store the member variable of the given component. These member variables are as explained in **D.1.1.2.1** through **D1.1.2.6**. The comment list will contain comments that have fields as described in **D.1.1.3.1** through **D.1.1.3.2**.